John P. Doran

Unity 2017 Mobile Game Development

Build, deploy, and monetize games for Android and iOS with Unity

Packt

Unity 2017 Mobile Game Development

Build, deploy, and monetize games for Android and iOS with Unity

John P. Doran

BIRMINGHAM - MUMBAI

Unity 2017 Mobile Game Development

First published: November 2017

Production reference: 1281117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham

B3 2PB, UK.

ISBN 978-1-78728-871-3

www.packtpub.com

Credits

Author John P. Doran	**Copy Editor** Dhanya Baburaj
Reviewer Francesco Sapio	**Project Coordinator** Ritika Manoj
Commissioning Editor Smeet Thakkar	**Proofreader** Safis Editing

Acquisition Editor Larissa Pinto	**Indexer** Rekha Nair
Content Development Editor Arun Nadar	**Graphics** Jason Monteiro
Technical Editor Harshal Kadam	**Production Coordinator** Nilesh Mohite

About the Author

John P. Doran is a passionate and seasoned Technical Game Designer, Software Engineer, and Author who is based in Redmond, Washington. His passion for game development began at an early age. He later graduated from DigiPen Institute of Technology with a Bachelor of Science in Game Design.

For over a decade, John has gained extensive hands-on expertise in game development working in various roles ranging from game designer to lead UI programmer working in teams consisting of just himself to over 70 people in student, mod, and professional game projects including working at LucasArts on *Star Wars: 1313*. Additionally, John has worked in game development education teaching in Singapore, South Korea, and the United States. To date, he has authored over 10 books pertaining to game development.

In addition to teaching, John is also a part of DigiPen’s Research and Development team. Prior to his present ventures, he was an award-winning videographer.

About the Reviewer

Francesco Sapio received his Master of Science in Engineering in Artificial Intelligence and Robotics degree from Sapienza University of Rome, Italy, a couple of semesters in advance, graduating summa cum laude; he is currently a PhD researcher at the same university.

He is a Unity 3D and Unreal expert, skilled game designer, and experienced user of major graphics programs. He developed *Game@School* (Sapienza University of Rome), an educational game for high-school students to learn concepts of physics, and the Sticker Book series (Dataware Games), a cross-platform series of games for kids. In addition, he worked as a consultant for the (successfully funded by Kickstarter) game Prosperity – Italy 1434 (Entertainment Game Apps, Inc.) and for an open online collaborative ideation system, titled Innovoice (Sapienza University of Rome). He has also been involved in different research projects such as Belief-Driven Pathfinding (Sapienza University of Rome), which is a new technique for path-finding in video games that was presented as a paper at the DiGRAFDG Conference 2016, and perfekt.ID (Royal Melbourne Institute of Technology), which included developing a recommendation system for games.

Francesco is an active writer on the topic of game development. Recently, Recently, he authored the book *Getting Started with Unity 5.x 2D Game Development* (Packt Publishing) that takes your hand and guide you through the amazing journey of game development, the successful *Unity UI Cookbook* (Packt Publishing), which has been translated also in other languages, that teaches readers how to develop exciting and practical user interfaces for games within Unity, and a short e-guide *What do you need to know about Unity* (Packt Publishing). In addition, he co-authored the book *Unity 5.x 2D Game Development Blueprints* (Packt Publishing), and the video course *Unity 5.x Game Development Projects* (Packt Publishing). Furthermore, he has also been a reviewer for the following books: *Game Audio Development with Unity 5.x* (Packt Publishing), *Game Development Patterns and Best Practices* (Packt Publishing), *Game Physics Cookbook* (Packt Publishing), *Mastering Unity 5.x* (Packt Publishing), *Unity 5.x by Example* (Packt Publishing), and *Unity Game*

Development Scripting (Packt Publishing); as well as for the following video courses: *Building an FPS Game with Unity and UFPS* (Packt Publishing), *Enhancement with Unity UI Advanced* (Packt Publishing), and *Making Sense of Data with Java* (Packt Publishing).

Francesco is also a musician and a composer, especially of soundtracks for short films and video games. For several years, he worked as an actor and dancer, where he was a guest of honor at the Teatro Brancaccio in Rome. In addition, he has volunteered as a children's entertainer at the Associazione Culturale Torraccia in Rome. Finally, Francesco loves math, philosophy, logic, and puzzle solving, but most of all, creating video games—thanks to his passion for game designing and programming.

I'm deeply thankful to my parents for their infinite patience, enthusiasm and support throughout my life. Moreover, I'm thankful to the rest of my family, in particular to my grandparents, since they have always encouraged me to do better in my life with the Latin expressions "Ad maiora" and "Per aspera ad astra".

www.PacktPub.com

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://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

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

Table of Contents

Preface

As an indie or AAA game developer, you want to have your games where your customers are. More and more people buy mobile devices every year and there's no sign of this stopping any time soon. One of the big advantages of the Unity game engine is that it is cross-platform, making it easy to write your game once and then port it to other consoles with minimal changes. However, there are certain features unique to working with mobile devices, which is what this book is about.

Unity 2017 Mobile Game Development will take readers on an exploration of how to use Unity when trying to deploy your content to mobile devices. Over the course of the book, we will see how to create a mobile game and then see how to deploy it to both iOS and Android. We will explore how to add input for mobile devices and have the interface adapt to the many different screen sizes that phones have. We'll then see some ways to monetize our game by discussing Unity's in-app purchase and advertisement systems. Then, we will see how we can share our game with the world by enabling us to use Twitter and Facebook's SDK. Afterward, we will see how to work with Unity's analytics system and then polish our title in a number of different ways, before putting it on the Google Play and iOS App Stores.

What this book covers

Chapter 1, *Building Your Game*, covers the creation of a simple project in Unity, which we will be modifying over the course of this book to make use of features commonly seen in mobile games. This chapter will also serve as a refresher for some fundamental concepts when working in Unity.

Chapter 2, *Setup for Android and iOS Development*, will show the setup required to deploy a project to both iOS and Android mobile devices, by installing the Java and Android SDKs for Android and configuring Xcode for iOS.

Chapter 3, *Mobile Input/Touch Controls*, shows a number of ways in which input can work on mobile devices. Starting off with mouse events, we will dive into recognizing touch events and gestures, as well as how to use the accelerometer and accessing information using the Touch class.

Chapter 4, *Resolution Independent UI*, discusses how to build the user interface for our game, starting with a title screen, and then build the other menus that we will want to use for our future chapters.

Chapter 5, *Advertising Using Unity Ads*, shows how to integrate Unity's Ad framework into our project and learn how to create both simple and complex versions of advertisements.

Chapter 6, *Implementing In-App Purchases*, talks about how to integrate Unity's In-App Purchase (IAP) system into our project and take a look at how to create an IAP that is used for consumable content as well as permanent unlocks.

Chapter 7, *Getting Social*, shows how to integrate social media into your projects, starting off with sharing high scores using Twitter and then taking a look at how we can use the Facebook SDK in order to display our player's name and profile picture while inside our game.

Chapter 8, *Using Unity Analytics*, covers some of the different ways that we can integrate Unity's Analytics tools into our projects, tracking custom events as well as using remote settings to allow us to tweak gameplay without having people

redownload the game from the store.

Chapter 9, *Making Your Title Juicy*, introduces the concept of making games *juicy* with different ways that you can integrate features of juiciness into our projects, including tweening animations, materials, post-processing effects, and adding particle effects.

Chapter 10, *Game Build and Submission*, goes over the process of submitting our game to the Google Play or iOS App Store, with tips and tricks to help the process go smoother.

What you need for this book

Throughout this book, we will work within the Unity 3D game engine, which you can download from http://unity3d.com/unity/download/. The projects were created using Unity 2017.2.0f3, but the project should work with minimal changes in future versions of the engine.

For the sake of simplicity, we will assume that you are working on a Windows-powered computer when developing for Android and a Macintosh computer when developing for iOS. Though Unity allows you to code in C#, Boo, or UnityScript, for this book we will be using C#.

Who this book is for

If you are a Unity game developer and want to build mobile games for iOS and Android, then this is the book for you. Previous knowledge of C# is helpful, but not required.

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: "Rename the sphere to `Player` and set the Transform component's Position to `(0, 1, -4)`."

A block of code is set as follows:

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update ()
{
   // Check if target is a valid object
   if (target != null)
   {
      // Set our position to an offset of our target
      transform.position = target.position + offset;

      // Change the rotation to face target
      transform.LookAt(target);
   }
}
```

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

```
[Tooltip("How fast the ball moves forwards automatically")]
    [Range(0, 10)]
    public float rollSpeed = 5;
```

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: "To fix this, go to Window | Lighting | Settings."

Warnings or important notes appear 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 for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.

5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Unity-2017-Mobile-Game-Development. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

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.

Building Your Game

As we start off on our journey building mobile games using the Unity game engine, it's important that readers are familiar with the engine itself before we dive into the specifics of building things for mobile platforms. Although there is a chance that you've already built a game and want to transition it to mobile, there will also be readers who haven't touched Unity before, or may have not used it in a long time. This chapter will act as an introduction to newcomers, a refresher for those coming back, and will provide some best practices for those who are already familiar with Unity.

In this chapter, we will build a 3-D endless runner game in the same vein as Imangi Studios, LLC's *Temple Run* series. In our case, we will have a player who will run continuously in a certain direction, and will dodge obstacles that come in their way. We can also add additional features to the game easily, as the game will endlessly have new things added to it.

Chapter overview

Over the course of this chapter, we will create a simple project in Unity, which we will be modifying over the course of this book to make use of features commonly seen in mobile games. While you may skip this chapter if you're already familiar with Unity, I find it's also a good idea to go through the project so that you know the thought processes behind why the project is made in the way that it is, so you can keep it in mind for your own future titles.

Your objectives

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

- Project setup
- Creating the player
- Improving scripts using attributes
- Having the camera follow the player
- Creating a basic tile
- Making the game endless
- Creating obstacles

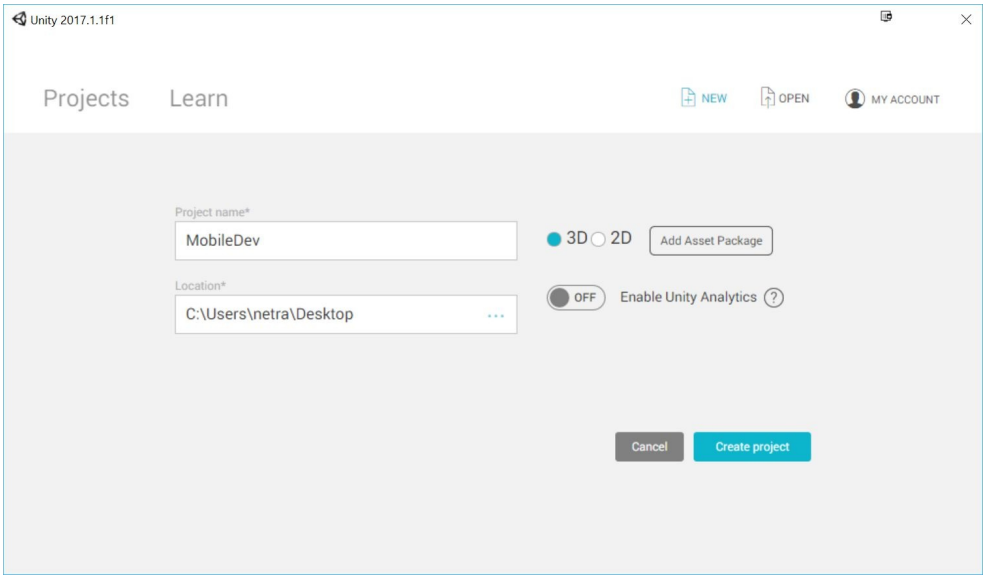
Setting up the project

Now that we have our goals in mind, let's start building our project:

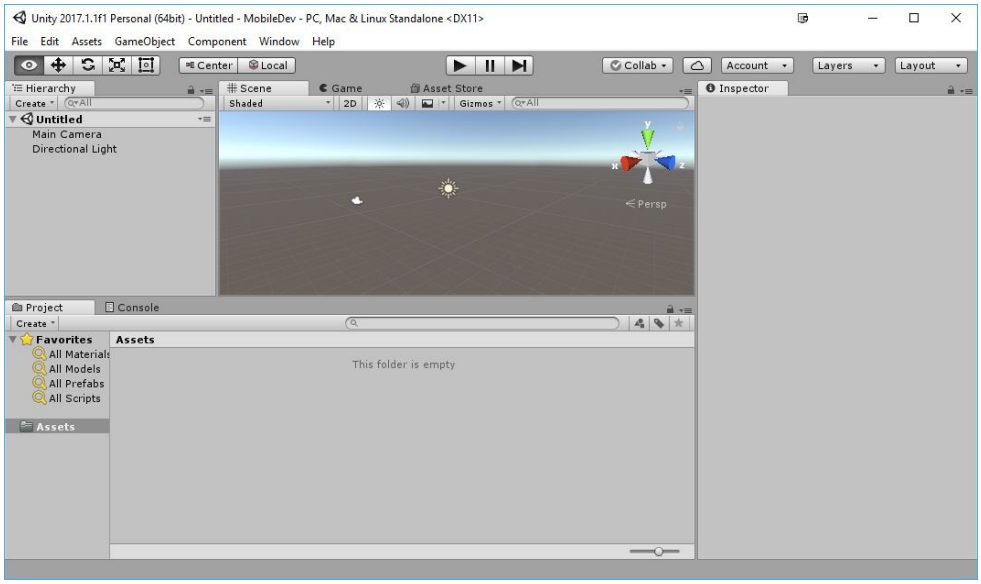
1. To get started, open Unity on your computer. For the purpose of this book, we will use Unity 2017.2.0f3, but the steps should work with minimal changes in future versions.

If you would like to download the exact version used in this book, and there is a new version out, you can visit Unity's download archive at https://unity3d.com/get-unity/download/archive.

2. From startup, we'll opt to create a new project by clicking on the New button.
3. Next, under Project name* put in a name (I have chosen `MobileDev`) and make sure that 3D is selected. If Enable Unity Analytics is enabled (the check to the left of it says On), click on the Enable Unity Analytics button again in order to disable it for the time being; we will add it ourselves later on when we go through Chapter 5, *Advertising with Unity Ads*. Afterwards, click on Create project and wait for Unity to load up:



4. After it's finished, you'll see the Unity Editor pop up for the first time:



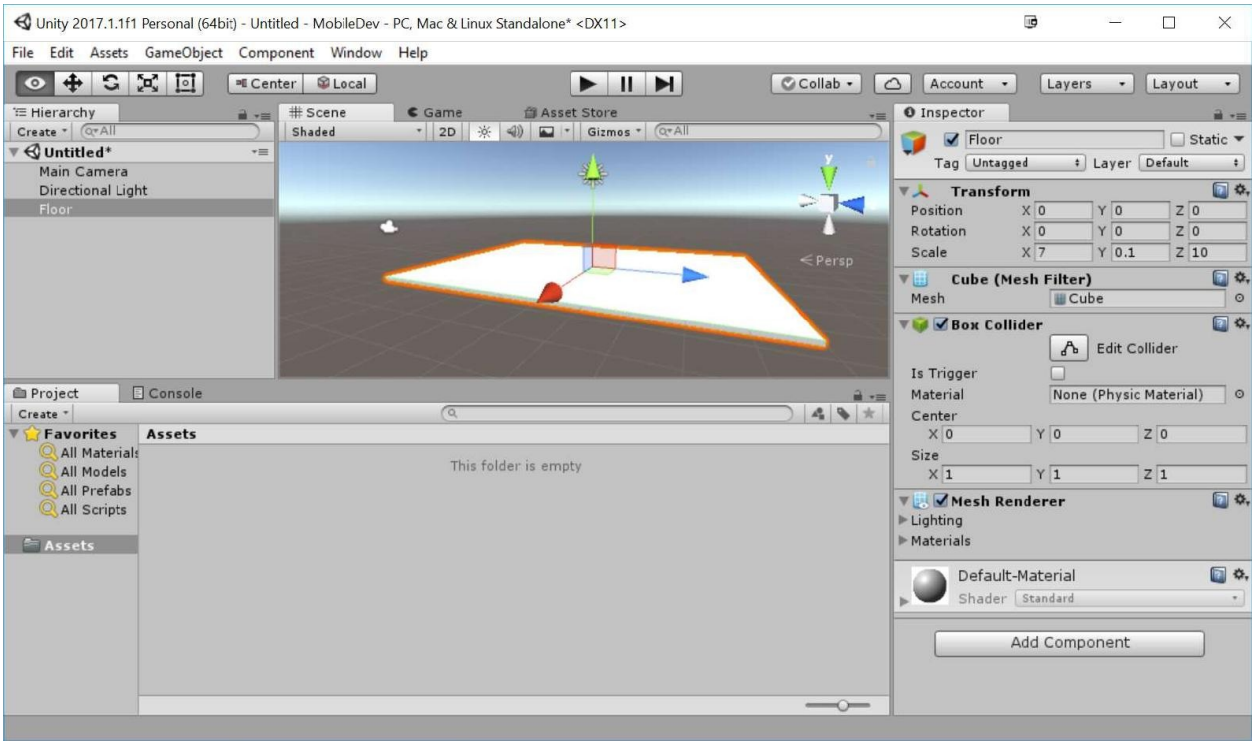
5. If your layout doesn't look the same as in the preceding screenshot, you may go to the top-right section of the toolbar and select the drop-down menu there that reads Layers. From there, select Default from the options presented.

If this is your first time working with Unity, then I highly suggest that you read the Learning the Interface section of the Unity Manual, which you can access at https://docs.unity3d.com/Manual/LearningtheInterface.html.

Creating the player

Now that we have Unity opened up, we can actually start building our project. To get started, let's build a player that will always move forward. Let's start with that now:

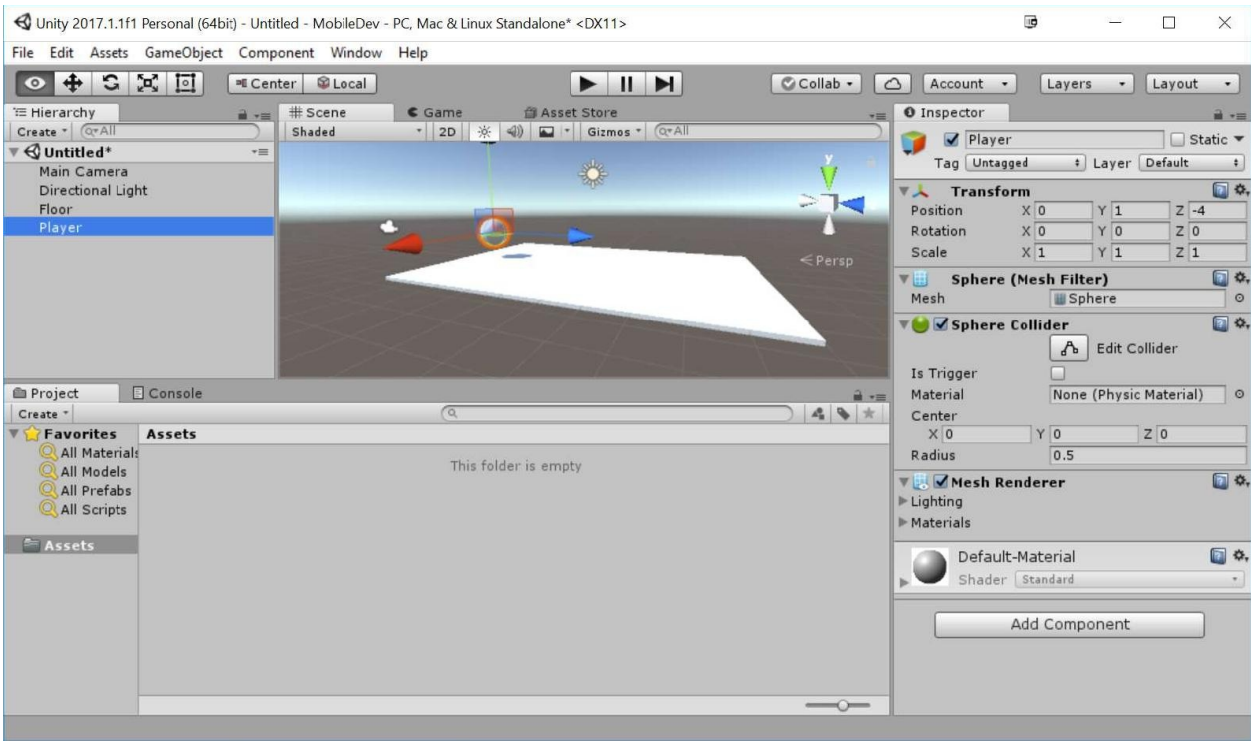
1. Let's create some ground for our player to walk on. To do that, let's go to the top menu and select GameObject | 3D Object | Cube.
2. From there, let's move over to the Inspector window and change the name of the object to `Floor`. Then, on the Transform component, set the Position to `(0, 0, 0)`, which we can either type in, or we can right-click on the Transform component and then select the Reset Position option.
3. Then, we will set the Scale to `(7, 0.1, 10)`:



In Unity, by default, 1 unit of space in Unity is representative of 1 meter in real life. This will make the floor longer than it is wide (X and Z), and we have some size on the ground (Y), so the player will collide and land on it because we have a Box Collider component attached to it.

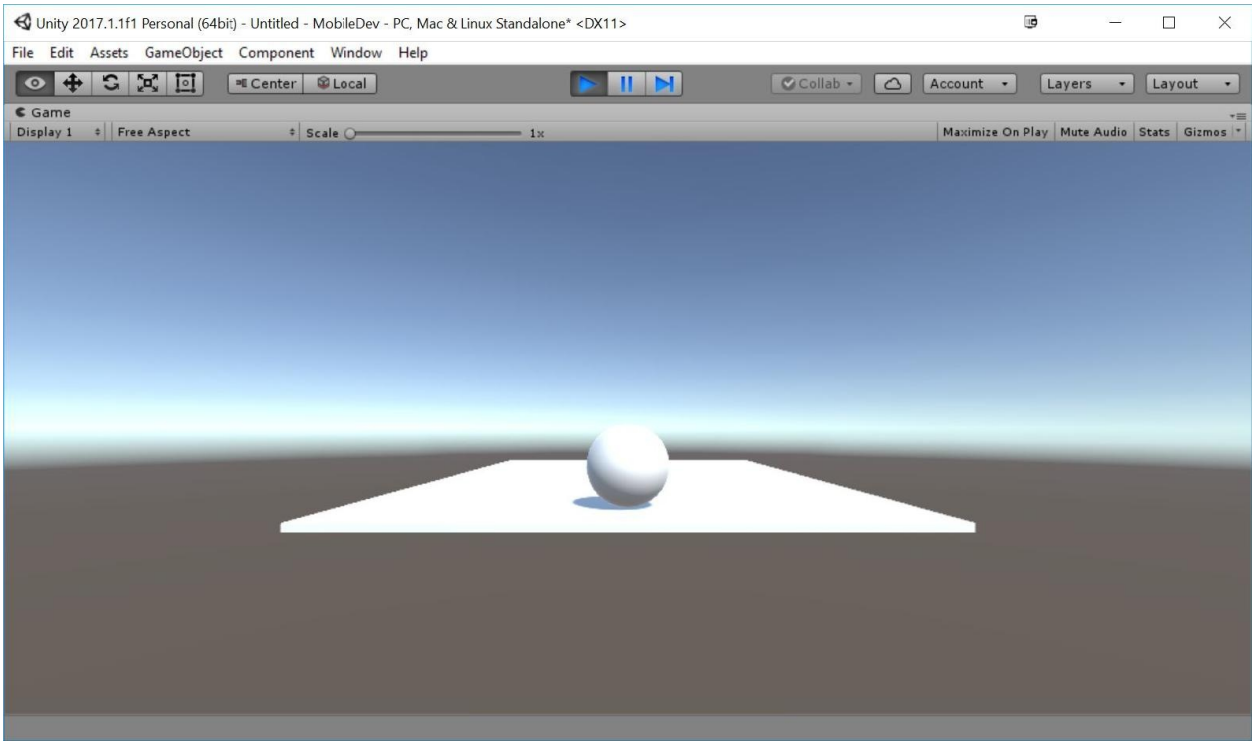
4. Next, we will create our player, which will be a sphere. To do this, we will go to Game Object | 3D Object | Sphere.

5. Rename the sphere to `Player` and set the Transform component's Position to `(0, 1, -4)`:



This will place the ball slightly above the ground, and shifts it back to near the starting point. Note that the camera object (you can see a camera icon to point it out) is pointing toward the ball by default because it is positioned at `0`, `1`, `-10`.

6. We want the ball to move, so we will need to tell the physics engine that we want to have this object react to forces, so we will need to add a Rigidbody component. To do so, go to the menu and select Component | Physics | Rigidbody. To see what happens now, let's click on the Play button that can be seen in the middle of the top toolbar:



As you can see in the preceding screenshot, you should see the ball fall down onto the ground when we play the game.

You can disable/enable having the Game tab take the entire screen when being played by clicking on the Maximize On Play button at the top, or by right-clicking on the Game tab and then selecting Maximize.

7. Click on the Play button again to turn the game off and go back to the Scene tab, if it doesn't happen automatically.

   We want to have the player move, so in order to do that, we will create our own piece of functionality in a script, effectively creating our own custom component in the process.

8. To create a script, we will go to the Project window and select Create | Folder on the top-left corner of the menu. From there, we'll name this folder `Scripts`. It's always a good idea to organize our projects, so this will help with that.

If you happen to misspell the name, go ahead and select the object

and then single-click on the name and it'll let you rename it.

9. Double-click on the folder to enter it, and now you can create a script by going to Create | C# Script and renaming this to `PlayerBehaviour` (no spaces).

The reason I'm using the "behaviour", "spelling" instead of "behavior" is that all components in Unity are children of another class called `MonoBehaviour`, and I'm following Unity's lead in that regard.

10. Double-click on the script to open up the script editor (IDE) of your choice and add the following code to it:

```
using UnityEngine;

public class PlayerBehaviour : MonoBehaviour
{
    // A reference to the Rigidbody component
    private Rigidbody rb;

    // How fast the ball moves left/right
    public float dodgeSpeed = 5;

    // How fast the ball moves forwards automatically
    public float rollSpeed = 5;

   // Use this for initialization
   void Start ()
   {
        // Get access to our Rigidbody component
        rb = GetComponent<Rigidbody>();
   }

   // Update is called once per frame
   void Update ()
   {
        // Check if we're moving to the side
        var horizontalSpeed = Input.GetAxis("Horizontal") * dodgeSpeed;
        rb.AddForce(horizontalSpeed, 0, rollSpeed);
   }
}
```

In the preceding code, we have a couple of variables that we will be working with. The `rb` variable is a reference to the game object's `Rigidbody` component that we added previously. It gives us the ability to make the object move, which we will use in the `Update` function. We also have two variables `dodgeSpeed` and `rollSpeed`, which dictates how quickly the player will move when moving left/right, or when moving forward, respectively.

Since our object has only one `Rigidbody` component, we assign `rb` once in the `Start` function, which is called when the game starts, as long as the game object attached to this script is enabled.

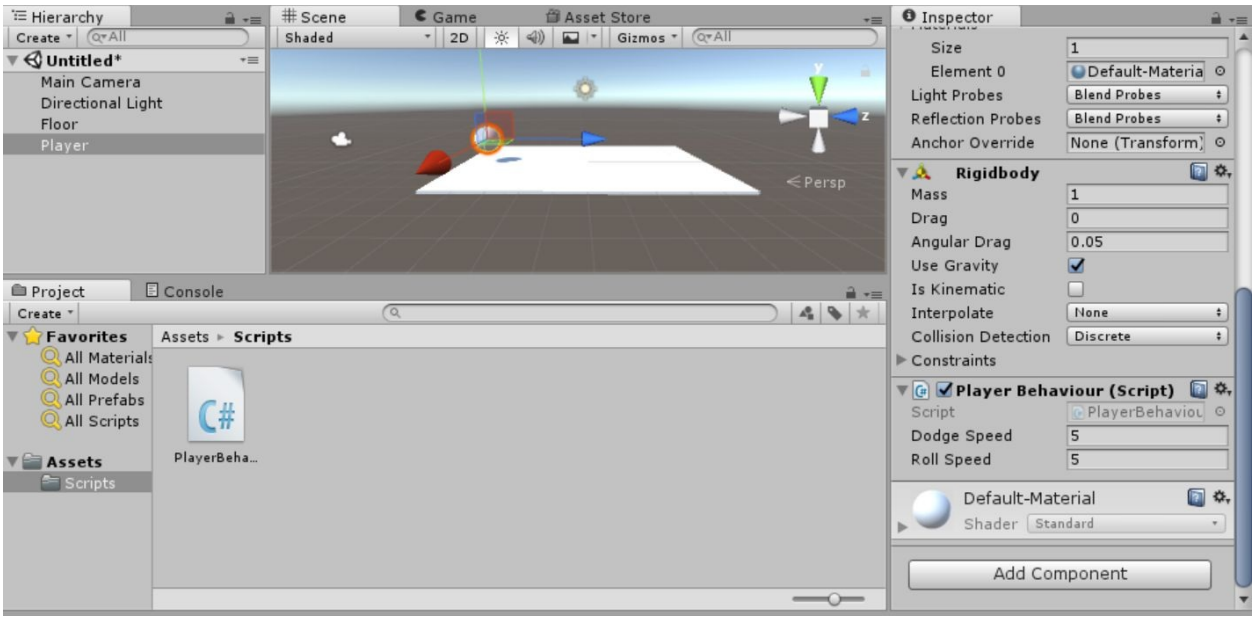
Then, we use the `Update` function to check whether our player is pressing keys to move left or right as based on Unity's Input Manager system. By default, the `Input.GetAxis` function will return to us a negative value moving to `-1` if we press *A* or the left arrow. If we press the right arrow or *D*, we will get a positive value up to `1` returned to us, and the input will move toward a `0` if nothing is pressed. We then multiply this by `dodgeSpeed` in order to increase the speed so that it is easier to be seen.

For more information on the Input Manager, check out https://docs.unity3d.com/Manual/class-InputManager.html.

Finally, once we have that value, we will apply a force to our ball's `horizontalSpeed` units on the X-axis and `rollSpeed` in the Z-axis.

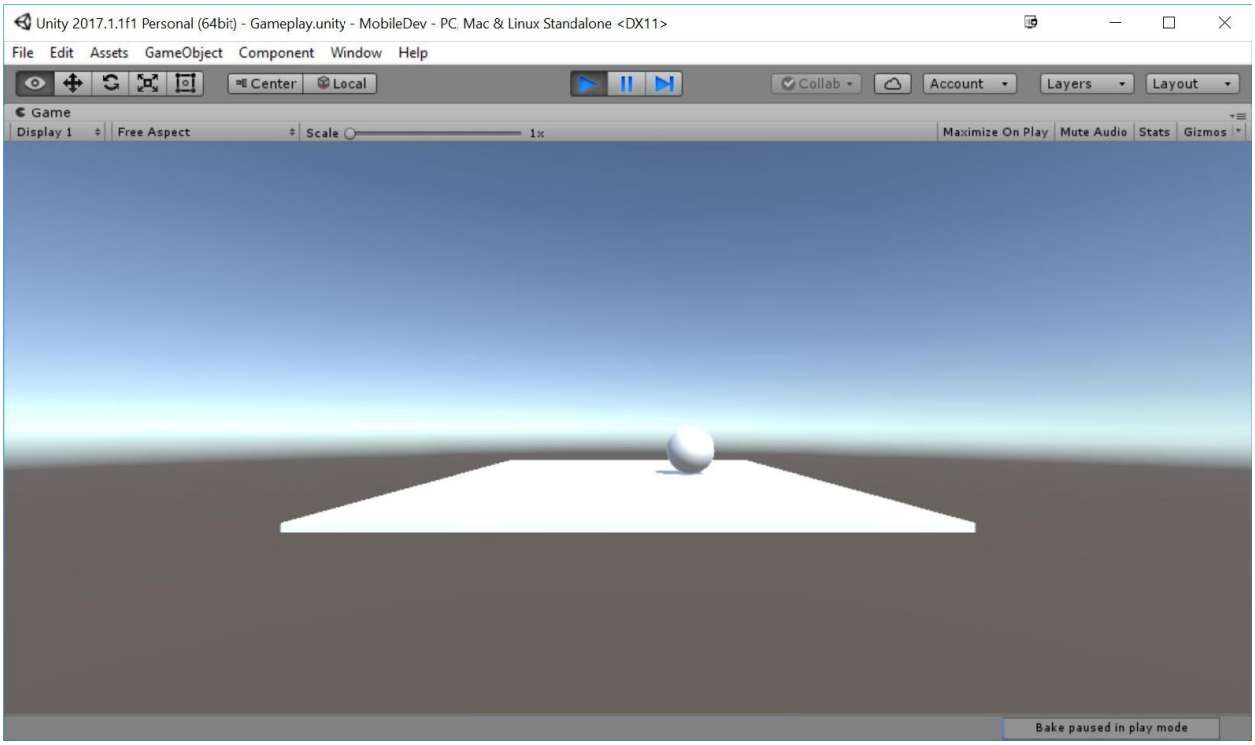
11. Save your script, and return to Unity.

12. We will now need to assign this script to our player by selecting the `Player` object in the Hierarchy window, and then in the Inspector window, drag and drop the `PlayerBehaviour` script from the Project window on top of the `Player` object. If all goes well, we should see the script appear on our object, as follows:



Note that when writing scripts if we declare a variable as `public`, it will show up in the Inspector window for us to be able to set it. We typically set a variable as public when we want designers to tweak the values for gameplay purposes.

13. Save your scene by going to File | Save Scene. Create a new folder called `Scenes` and save your scene as `Gameplay`. Afterward, play the game and use the left and right arrows to see the player moving according to your input, but no matter what, moving forward by default:


Unity 2017.1.1f1 Personal (64bit) - Gameplay.unity - MobileDev - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Window Help
Center Local
Collab
Account
Layers
Layout
Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
Bake paused in play mode

Improving our scripts with attributes and XML comments

We could stop working with the `PlayerBehaviour` class script here, but I want to touch on a couple of things that we can use in order to improve the quality and style of our code. This becomes especially useful when you start building projects in teams, as you'll be working with other people--some of them will be working on code with you, and then there are designers and artists who will not be working on code with you, but will still need to use the things that you've programmed.

When writing scripts, we want them to be as error-proof as possible. Making the `rb` variable `private` starts that process, as now the user will not be able to modify that anywhere outside of this class. We want our teammates to modify `dodgeSpeed` and `rollSpeed`, but we may want to give them some advice as to what it is and/or how it will be used. To do this in the Inspector window, we can make use of something called an *attribute*.

Using attributes

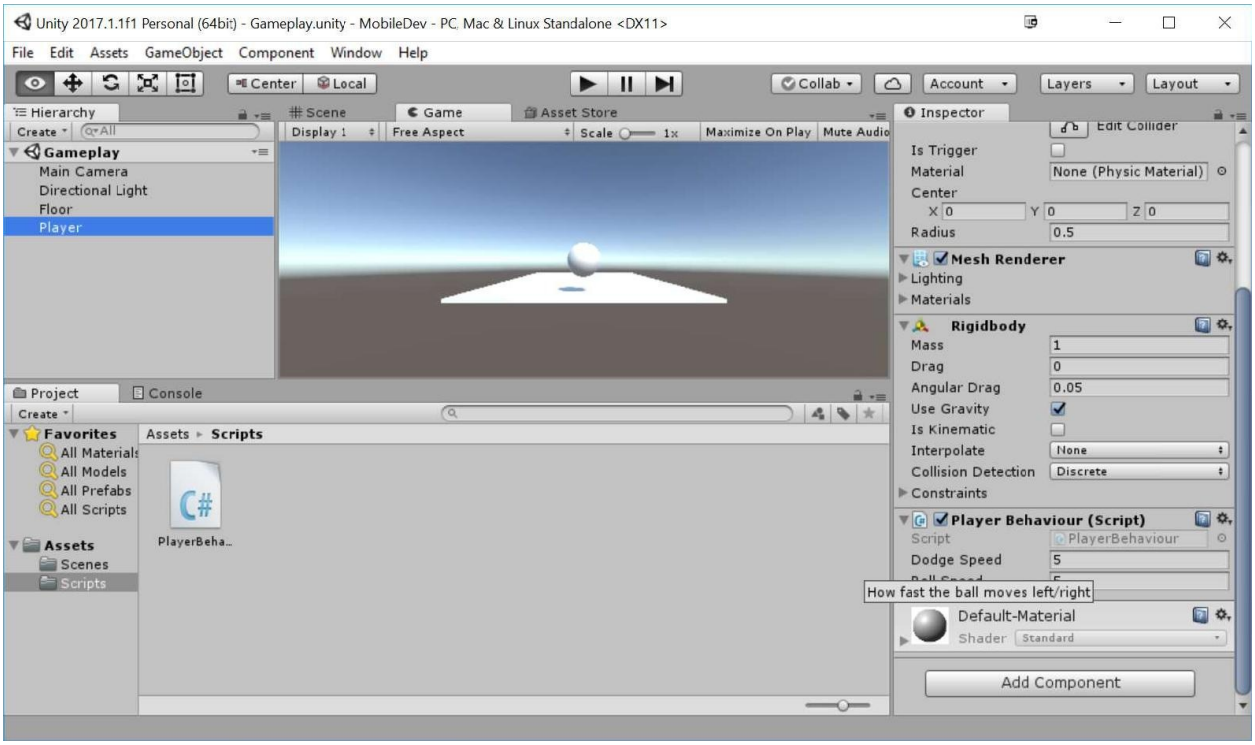
Attributes are things we can add to the beginning of a variable, class, or function declaration, which allow us to attach an additional functionality to them. There are many of them that exist inside Unity, and you can write your very own as well, but, right now, we'll talk about the ones that I use most often.

The Tooltip attribute

If you've used Unity for a period of time, you may have noted that some components in the Inspector window, such as the `Rigidbody`, have a nice feature--if you move your mouse over a variable name, you'll see a description of what the variables are and/or how to use them. The first thing you'll learn is how we can get the same effect in our own components by making use of the `Tooltip` attribute. If we do this for the `dodgeSpeed` and `rollSpeed` variables, it will look something like this:

```
[Tooltip("How fast the ball moves left/right")]
public float dodgeSpeed = 5;

[Tooltip("How fast the ball moves forwards automatically")]
public float rollSpeed = 5;
```

Save the preceding script and return to the editor:



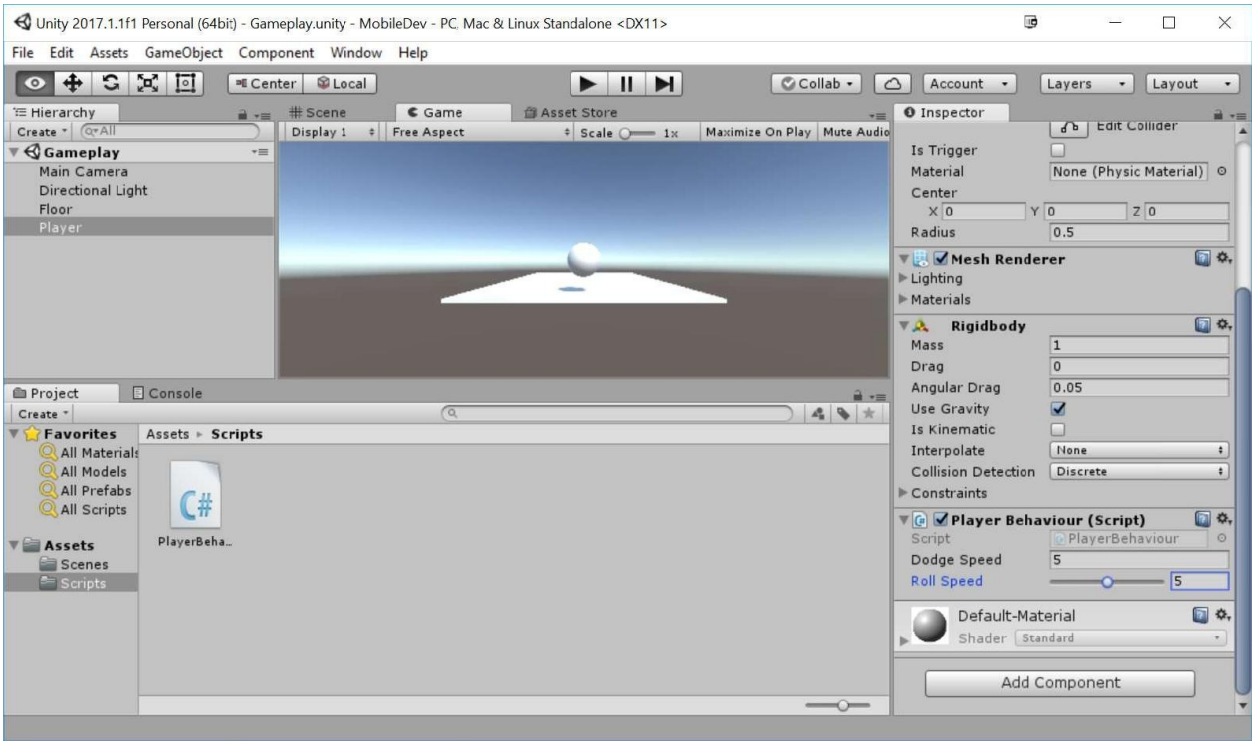
Now, when we highlight the variable using the mouse and leave it there, the text we placed will be displayed. This is a great habit to get into, as your teammates can always tell what it is that your variables are being used for.

For more information on the `Tooltip` attribute, check out, https://docs.unity3d.com/ScriptReference/TooltipAttribute.html.

The Range attribute

Another thing that we can use to protect our code is the `Range` attribute. This will allow us to specify a minimum and maximum value for a variable. Since we want the player to always be moving forward, we may want to restrict the player from moving backward. To do that, we can add the following highlighted line of code: [Tooltip("How fast the ball moves forwards automatically")]
[Range(0, 10)]
public float rollSpeed = 5;

Save your script, and return to the editor:



We have now added a slider beside our value, and we can drag it to adjust between our minimum and maximum values. Not only does this protect our variable, it also makes it so our designers can tweak things easily by just dragging them around.

The RequireComponent attribute

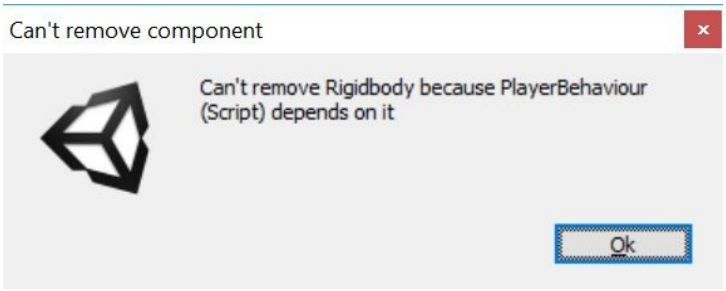
Currently, we are using the `Rigidbody` component in order to create our script. When working as a team member, others may not be reading your scripts, but are still expected to use them when creating gameplay. Unfortunately, this means that they may do things that have unintended results, such as removing the `Rigidbody` component, which will cause errors when our script is run. Thankfully, we also have the `RequireComponent` attribute, which we can use to fix this.

It looks something like this:

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody))]
public class PlayerBehaviour : MonoBehaviour
```

By adding this attribute, we state that when we add this component to a game object and it doesn't have a `Rigidbody` attached to its game object, the component will be added automatically. It also makes it so that if we were to try to remove the `Rigidbody` from this object, the editor will warn us that we can't, unless we remove the PlayerBehaviour component first. Note that this works for any class extended from `MonoBehaviour`; just replace `Rigidbody` with whatever it is that you wish to keep.

Now, if we go into the Unity editor and try to remove the `Rigidbody` component by right-clicking on it in the Inspector and selecting Remove Component, the following message will be seen:



This is exactly what we want, and this ensures that the component will be there, allowing us not to have to include if-checks every time we want to use a

component.

XML comments

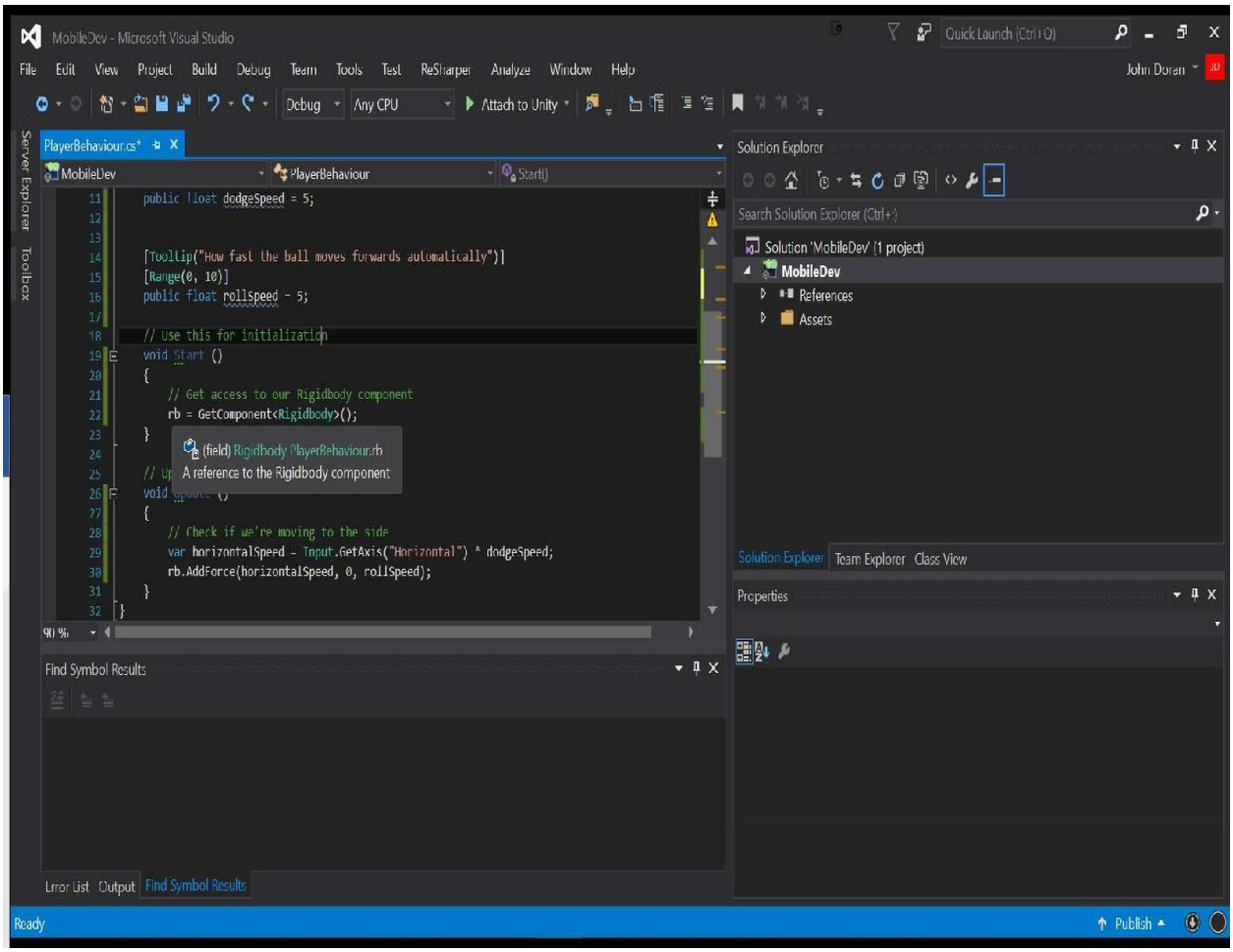
Note that previously we did not use a `Tooltip` attribute on the private `rb` variable. Since it's not being displayed in the editor, it's not really needed. However, there is a way that we can enhance that as well, making use of XML comments. XML comments have a couple of nice things that we get when using them instead of traditional comments, which we were using previously. When using the variables/functions instead of code in Visual Studio, we will now see a comment about it. This will help other coders on your team have additional information and details to ensure that they are using your code correctly.

XML comments look something like this:

```
/// <summary>
/// A reference to the Rigidbody component
/// </summary>
private Rigidbody rb;
```

It may appear to be a lot more writing is needed to use this format, but I did not actually type the entire thing out. XML comments are a fairly standard C# feature, so if you are using MonoDevelop or Visual Studio and type `///`, it will automatically generate the summary blocks for you (and the `param` tags needed, if there are parameters needed for something like a function).

Now, why would we want to do this? Well, now, if you select the variable in Intellisense, it will display the following information to us:



This is a great help for when other people are trying to use your code and it is how Unity's staff write their code. We can also extend this to functions and classes to ensure that our code is more self-documented.

Unfortunately, XML comments do not show up in the Inspector, and the `Tooltip` attribute doesn't show info in the editor. With that in mind, I used `Tooltips` for public instructions and/or things that will show up in the Inspector window, and XML comments for everything else.

If you're interested in looking into XML comments more, feel free to check out: *https://msdn.microsoft.com/en-us/library/b2s063f7.aspx*.

Putting it all together

With all of the stuff we've been talking about, we can now have the final version of the script, which looks like the following: using UnityEngine;

```
/// <summary>
/// Responsible for moving the player automatically and
/// reciving input.
/// </summary>
[RequireComponent(typeof(Rigidbody))]

public class PlayerBehaviour : MonoBehaviour
{
/// <summary>
/// A reference to the Rigidbody component
/// </summary>
private Rigidbody rb;

[Tooltip("How fast the ball moves left/right")]
public float dodgeSpeed = 5;


[Tooltip("How fast the ball moves forwards automatically")]
[Range(0, 10)]
public float rollSpeed = 5;

/// <summary>
/// Use this for initialization
/// </summary>
void Start ()
{
// Get access to our Rigidbody component
rb = GetComponent<Rigidbody>();
```

```
}

/// <summary>
/// Update is called once per frame
/// </summary>
void Update ()
{
// Check if we're moving to the side
var horizontalSpeed = Input.GetAxis("Horizontal") *
dodgeSpeed;

// Apply our auto-moving and movement forces
rb.AddForce(horizontalSpeed, 0, rollSpeed);
}
}
```

I hope that you also agree that this makes the code easier to understand and better to work with.

Having the camera following our player

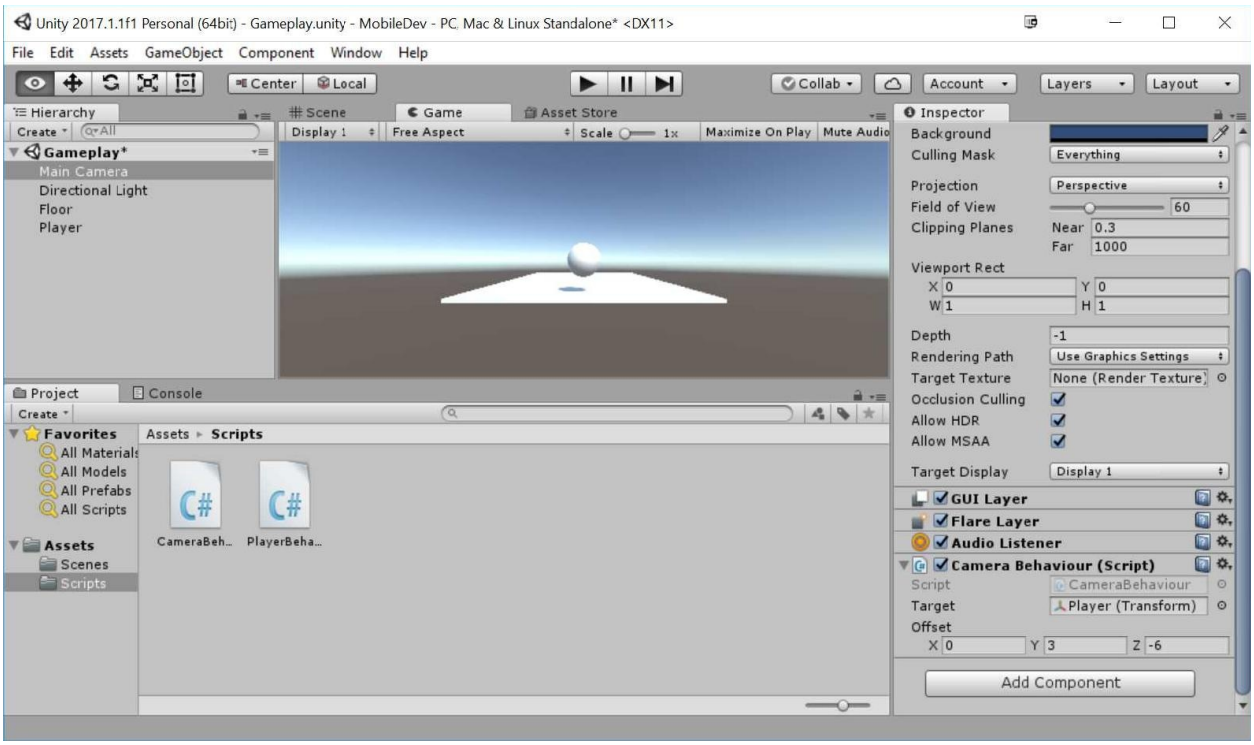
Currently, our camera stays in the same spot while the game is going on. This does not work very well for this game, as the player will be moving more the longer the game is going on. There are two main ways that we can move our camera. We can just move the camera and make it a child of the player, but that will not work due to the ball's rotation. Due to that, we will likely want to use a script instead. Thankfully, we can modify how our camera looks at things fairly easily, so let's go ahead and fix that next:

1. Go to the Project window and create a new C# script called `CameraBehaviour`. From there, use the following code:

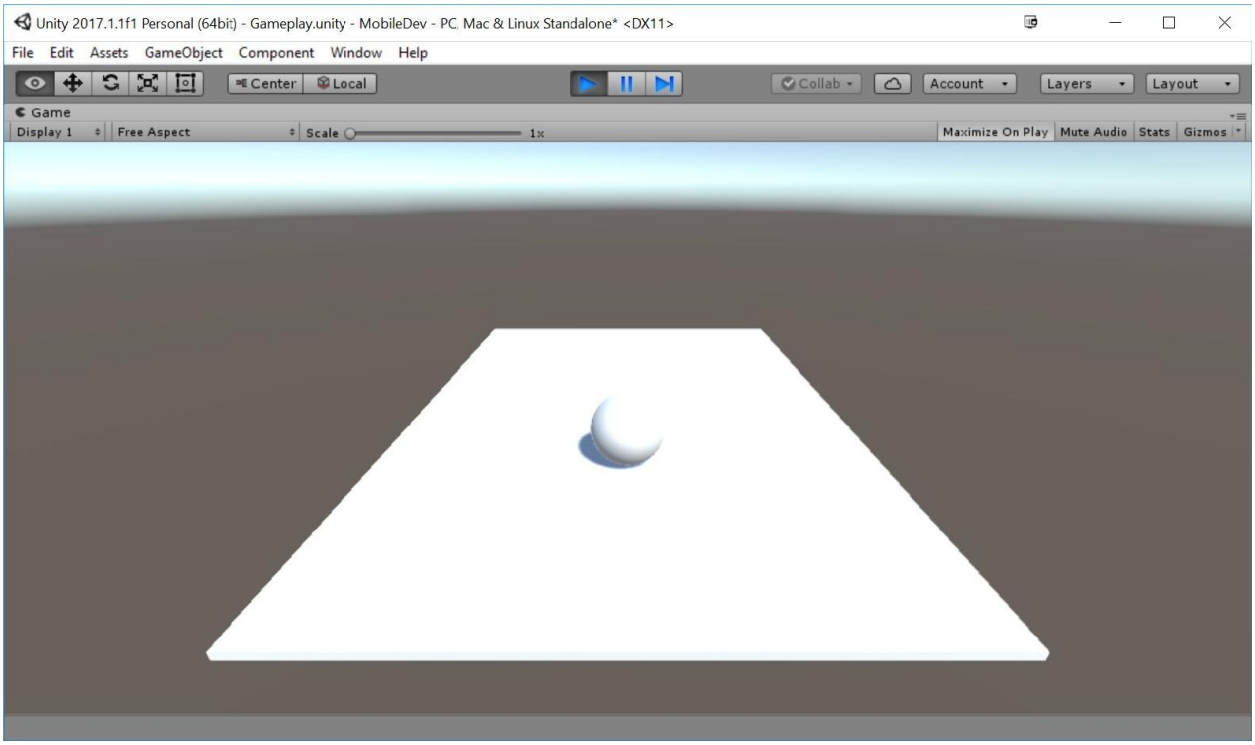
```
using UnityEngine;

/// <summary>
/// Will adjust the camera to follow and face a target
/// </summary>
public class CameraBehaviour : MonoBehaviour
{
    [Tooltip("What object should the camera be looking at")]
    public Transform target;

    [Tooltip("How offset will the camera be to the target")]
    public Vector3 offset = new Vector3(0, 3, -6);

    /// <summary>
    /// Update is called once per frame
    /// </summary>
    void Update ()
   {
        // Check if target is a valid object
        if (target != null)
        {
            // Set our position to an offset of our target
            transform.position = target.position + offset;

            // Change the rotation to face target
            transform.LookAt(target);
        }

   }
}
```

2. Save the script and dive back into the Unity Editor. Select the `Main Camera` object in the Hierarchy window. Then, go to the Inspector window and add the `CameraBehaviour` component to it. You may do this by dragging and dropping the script from the Project window onto the game object or by clicking on the Add Component button at the bottom of the Inspector window, typing in the name of our component, and then clicking on *Enter* to confirm once it is highlighted.

3. Afterward, drag and drop the `Player` object from the Hierarchy window into the Target property of the script in the Inspector window:



4. Save the scene, and play the game:



The camera now follows the player as it moves. Feel free to tweak the variables and see how it effects the look of the camera to get the feel you'd like best for the project.

Creating a basic tile

We want our game to be endless, but in order to do so, we will need to have pieces that we can spawn to build our environment; let's do that now:

1. To get started, we will first need to create a single piece for our runner game. To do that, let's first add some walls to the floor we already have. From the Hierarchy window, select the `Floor` object and duplicate it by pressing *Ctrl + D* in Windows or *command + D* on Mac. Rename this new object as `Left Wall`.

2. Change the `Left Wall` object's Transform component by adjusting the Scale to `(1, 2, 10)`. From there, select the Move tool by clicking on the button with arrows on the toolbar or by pressing the *W* key.

For more information on Unity's built-in hotkeys, check out: https://docs.unity3d.com/Manual/UnityHotkeys.html.

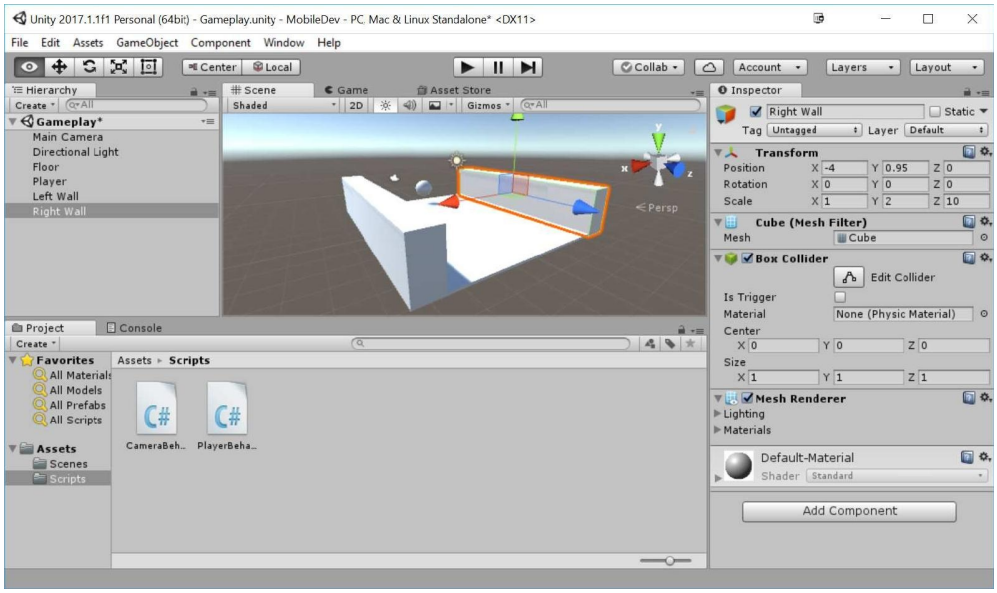
3. We want this wall to match up with the floor, so hold down the *V* key to enter the **Vertex Snap** mode.

> In the Vertex Snap mode, we can select any of the vertices on a mesh and move it to the same position of another vertex on a different object. This is really useful for making sure that objects don't have holes between them.

4. With the Vertex Snap mode on, select the inner edge and drag it until it hits the edge of the floor.

For more info on moving objects through the scene, including more details on Vertex Snap mode, check out https://docs.unity3d.com/Manual/PositioningGameObjects.html.

5. Then, duplicate this wall and put an other on the other side, naming it `Right Wall`:



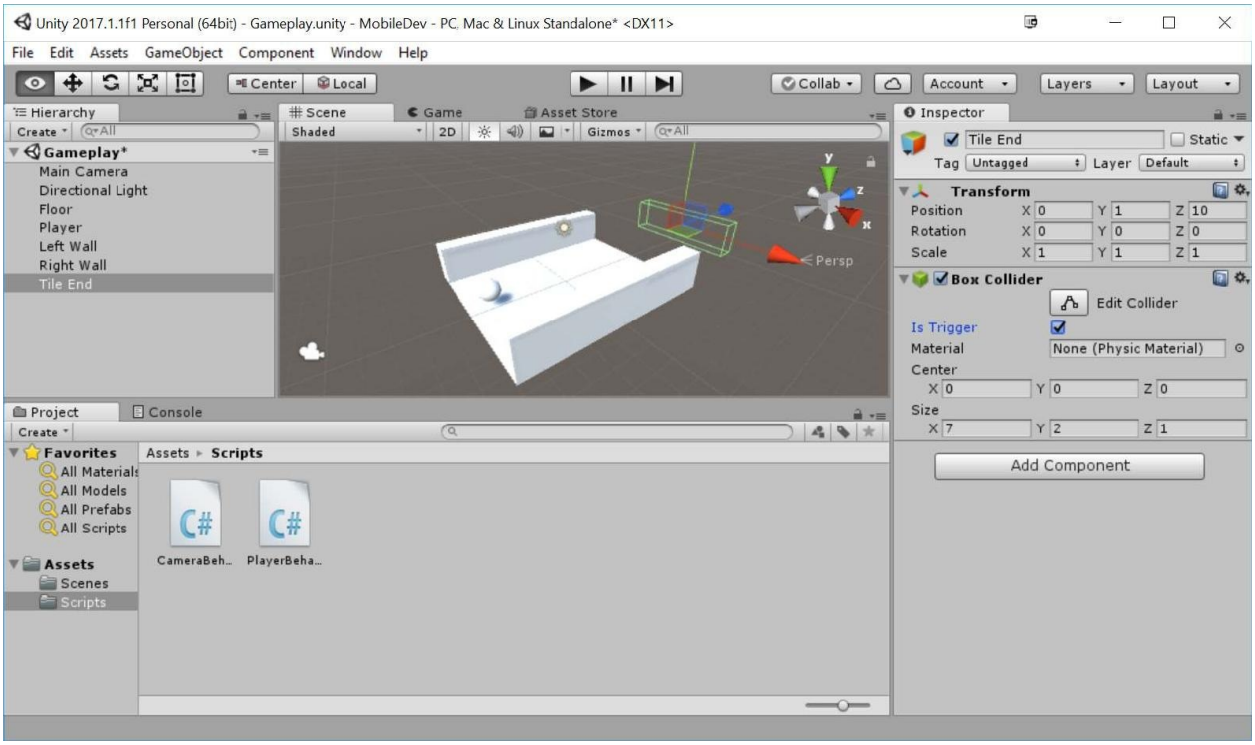
As you can see in the preceding screenshot, we now protect the player from falling off the left and right edges of the play area. Due to how the walls are set up, if we move the floor object, the walls will move as well.

For info on moving Unity's camera or navigating to the scene view, check out: https://docs.unity3d.com/Manual/SceneViewNavigation.html.

The way this game is designed, after the ball rolls past a single tile, we will no longer need it to be there anymore. If we just leave it there, the game will get slower over time due to us having so many things in the game environment using memory, so it's a good idea to remove assets we are no longer using. We also need to have some way to figure out when we should spawn new tiles to continue the path the player can take.

6. Now, we also want to know where this piece ends, so we'll add an object with a Trigger collider in it. Select GameObject | Create Empty and name this object `Tile End`.
7. Then, we will add a Box Collider component to our `Tile End` object. Under

the Box Collider in the Inspector window, set the Size to `(7, 2, 1)` to fit the size of the space the player can walk in. Note that there is a green box around that space showing where collisions can take place. Set the Position property to `(0, 1, 10)` to reach past the end of our tile. Finally, check the Is Trigger property so that the collision engine will turn the collider into a **trigger**, which will be able to run code events when it is hit, but will not prevent the player from moving:

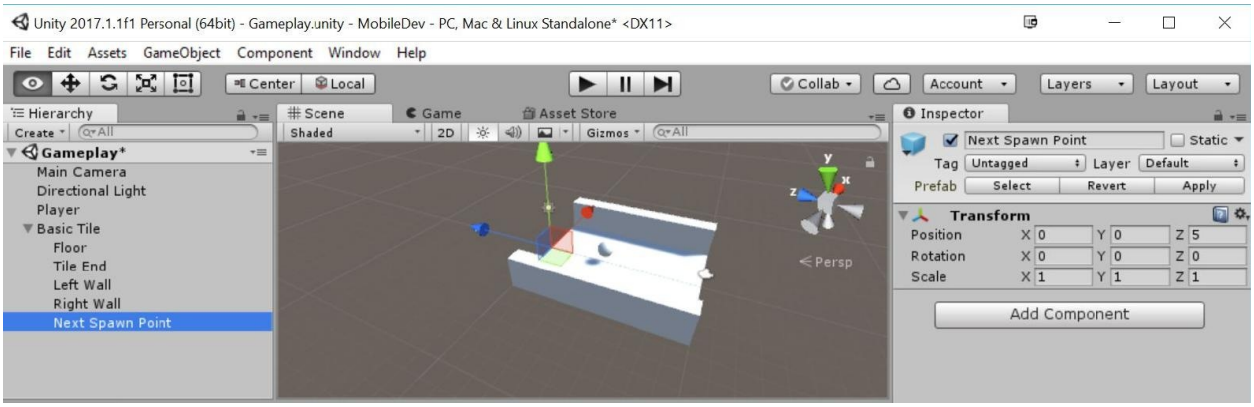


Like I mentioned briefly before, this trigger will be used to tell the game that our player has finished walking over this tile. This is positioned past the tile due to the fact that we want to still see tiles until they pass what the camera can see. We'll tell the engine to remove this tile from the game, but we will dive more into that later on in the chapter.

8. Now that we have all of the objects created, we want to group our objects together as one piece that we can create duplicates of. To do this, let's create an Empty Game Object by going to GameObject | Create Empty and name the newly created object to `Basic Tile`.

9. Then, go to the Hierarchy window and drag and drop the `Floor`, `Tile End`, `Left Wall`, and `Right Wall` objects on top of it to make them children of the `Basic Tile` object.
10. Currently, the camera can see the start of the tiles, so to fix that, let's set the `Basic Tile`'s Position to `(0, 0, -5)` so that the entire tile will shift back.
11. Finally, we will need to know at what position we should spawn the next piece, so create another child of `Basic Tile`, give it the name, `Next Spawn Point`, and set its Position to `(0, 0, 5)`.

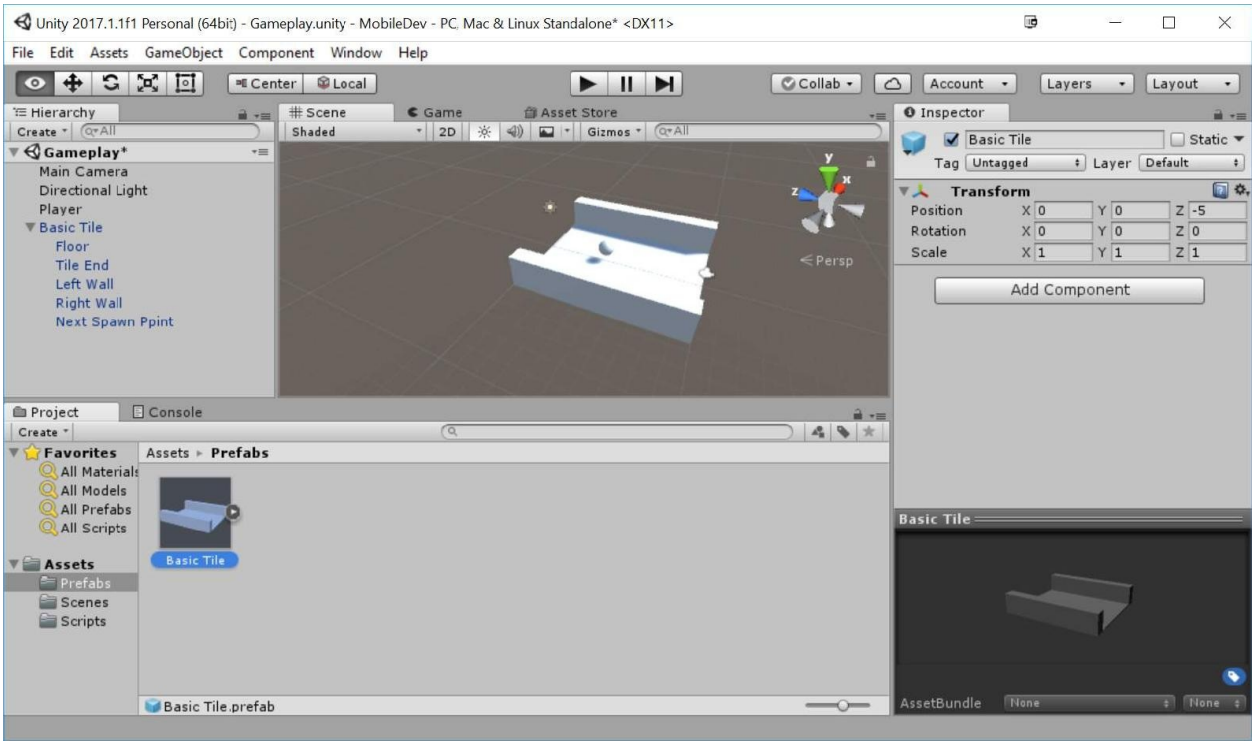
Note that when we modify an object that has a parent, the position is relative to the parent, not its world position.



Notice that the spawn point is on the edge of our current title. Now we have a single tile that is fully completed. Instead of duplicating this a number of times by hand, we will make use of Unity's concept or **prefabs**.

Prefabs, or prefabricated objects, are blueprints of game objects and components that we can turn into files, which can be duplicates. There are other interesting features that prefabs have, but we will discuss them as we make use of them.

12. From the Project window, go to the `Assets` folder and then create a new folder called `Prefabs`. Then, drag and drop the `Basic Tile` object from the Hierarchy window to the Project window inside the `Prefabs` folder. If the text on the `Basic Tile` name in the Hierarchy window becomes blue, we will know that it was made correctly:



With that, we now have a tile prefab that we can create duplicates of the tile through code to extend our environment.

Making it endless

Now that we have a foundation, let's now make it so that we can continue running instead of stopping after a short time:

1. To start off with, we have our prefab, so we can delete the original `Basic Tile` in the Hierarchy window by selecting it and then pressing the *Delete* key.

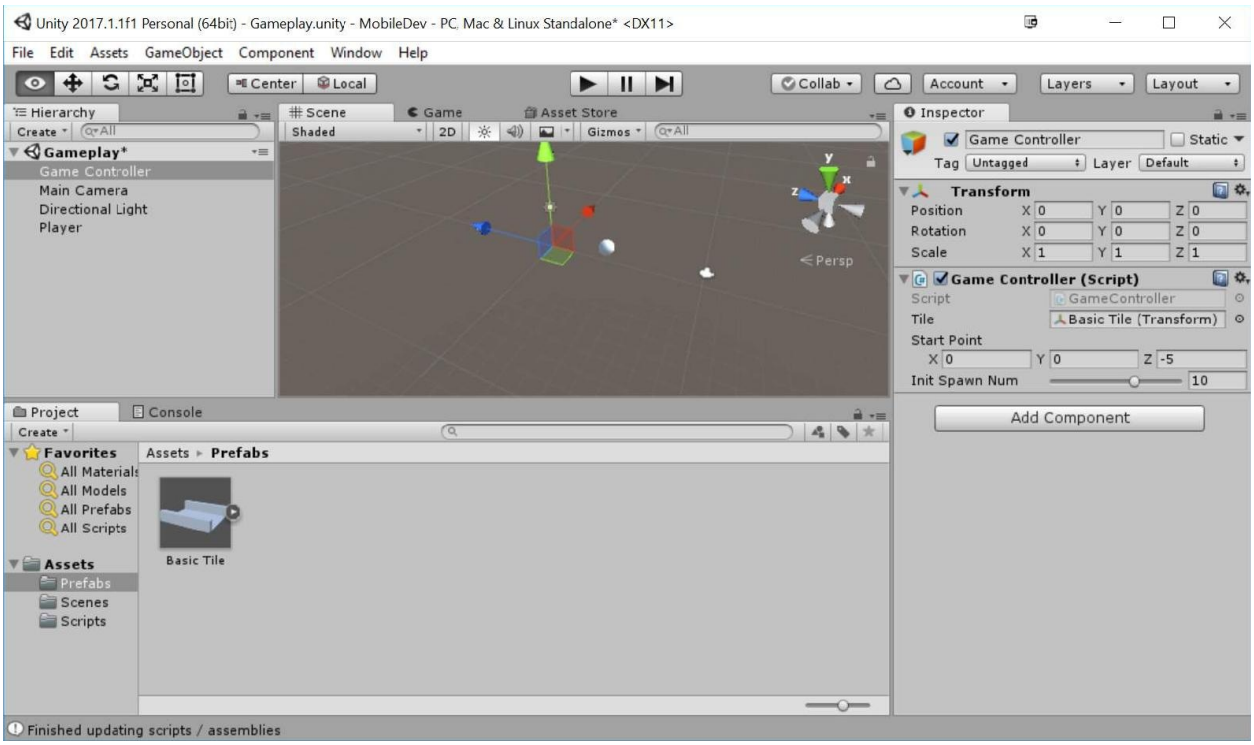
2. We need to have a place to create all of these tiles and potentially manage information for the game, such as the player's score. In Unity, this is typically referred to as a `GameController`. From the Project window, go to the `Scripts` folder and create a new C# script called `GameController`.
3. Open the script in your IDE, and use the following code:

```
using UnityEngine;

/// <summary>
/// Controls the main gameplay
/// </summary>

public class GameController : MonoBehaviour
{
    [Tooltip("A reference to the tile we want to spawn")]
    public Transform tile;

    [Tooltip("Where the first tile should be placed at")]
    public Vector3 startPoint = new Vector3(0, 0, -5);

    [Tooltip("How many tiles should we create in advance")]
    [Range(1, 15)]
    public int initSpawnNum = 10;

    /// <summary>
    /// Where the next tile should be spawned at.
    /// </summary>
    private Vector3 nextTileLocation;

    /// <summary>
    /// How should the next tile be rotated?
    /// </summary>
    private Quaternion nextTileRotation;

    /// <summary>
    /// Used for initialization
    /// </summary>

    void Start()
    {
        // Set our starting point
        nextTileLocation = startPoint;
        nextTileRotation = Quaternion.identity;
```

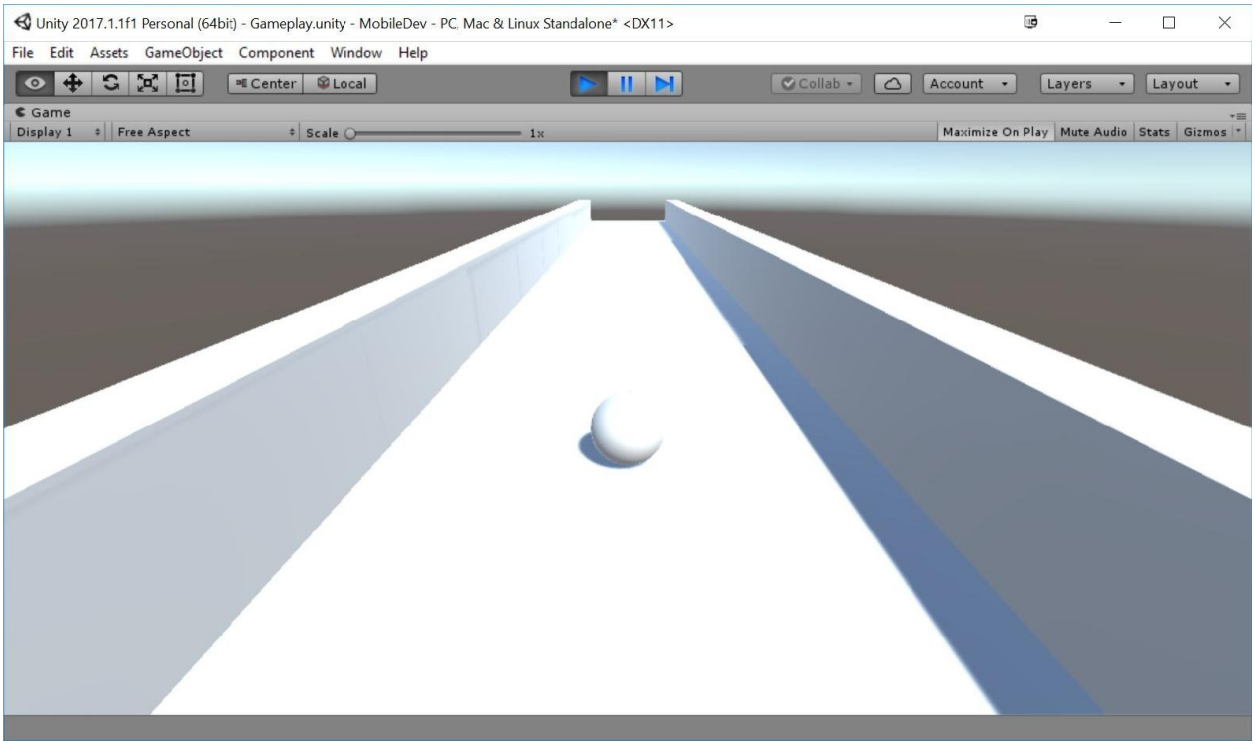
```
        for (int i = 0; i < initSpawnNum; ++i)
        {
            SpawnNextTile();
        }
    }

    /// <summary>
    /// Will spawn a tile at a certain location and setup the next position
    /// </summary>

    public void SpawnNextTile()
    {
        var newTile = Instantiate(tile, nextTileLocation,
                                  nextTileRotation);

        // Figure out where and at what rotation we should spawn
        // the next item
        var nextTile = newTile.Find("Next Spawn Point");
        nextTileLocation = nextTile.position;
        nextTileRotation = nextTile.rotation;
    }
}
```

This script will spawn a number of tiles, one after another, based on the `tile` and `initSpawnNum` properties.

4. Save your script and dive back into Unity. From there, create a new Empty game object and name it `Game Controller`. Drag and drop it to the top of the Hierarchy window. For clarity's sake, go ahead and reset the position if you want to. Then, attach the Game Controller script to the object and then set the Tile property by dragging and dropping the `Basic Tile` prefab from the Project window into the Tile slot:



5. Save your scene and run the project:



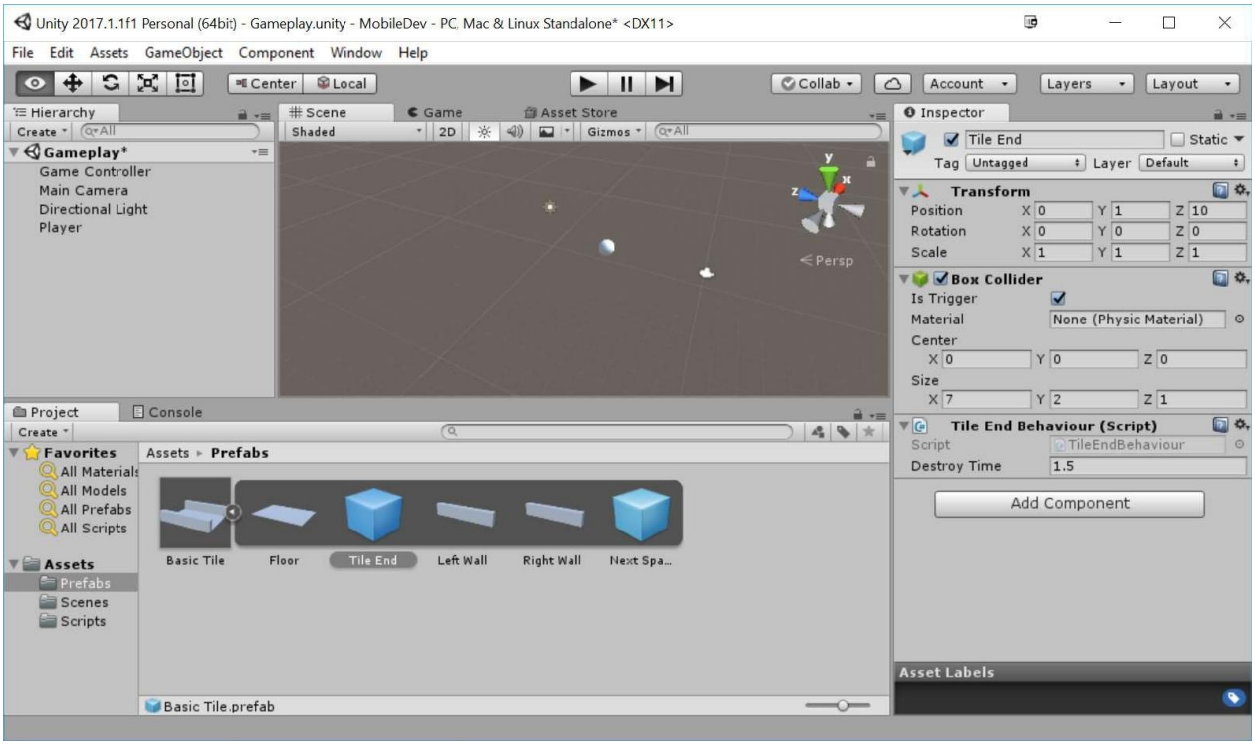
Great, but now we will need to create new objects after these, but we don't want to spawn a crazy number of these at once. It's better that once we reach the end of a tile, we create a new tile and remove it. We'll work on optimizing this more later, but that way we always have around the same number of tiles in the game at one time.

6. Go into the Project window and create a new script called `TileEndBehaviour`, using the following code:

```
using UnityEngine;

/// <summary>
/// Handles spawning a new tile and destroying this one
/// upon the player reaching the end
/// </summary>
public class TileEndBehaviour : MonoBehaviour
{
    [Tooltip("How much time to wait before destroying " +
             "the tile after reaching the end")]
    public float destroyTime = 1.5f;

    void OnTriggerEnter(Collider col)
    {
        // First check if we collided with the player
        if (col.gameObject.GetComponent<PlayerBehaviour>())
        {
            // If we did, spawn a new tile
            GameObject.FindObjectOfType<GameController>().SpawnNextTile();
```

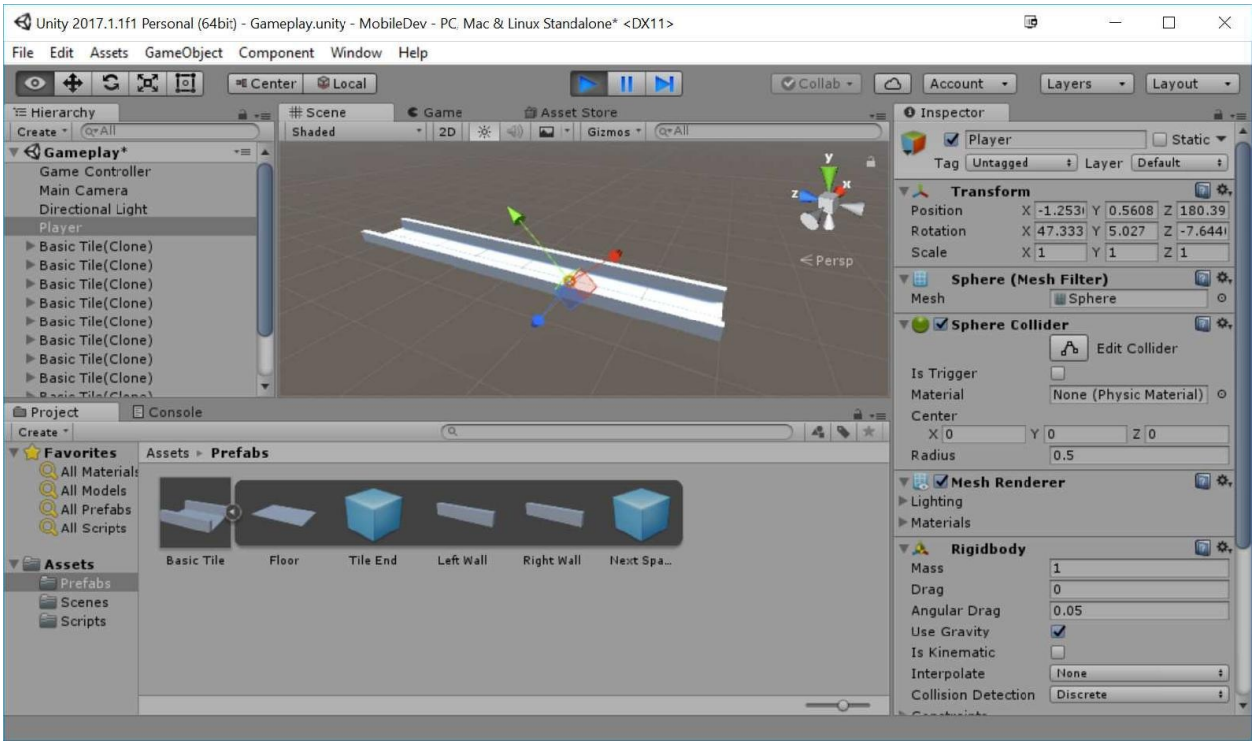
```
            // And destroy this entire tile after a short delay
            Destroy(transform.parent.gameObject, destroyTime);
        }
    }
}
```

7. Now, to assign it to the prefab, we can go to the Project window and then go into the `Prefabs` folder. From there, click on the arrow beside the `Basic Tile` to open up its objects and then add a Tile End Behaviour component to the `Tile End` object:



8. Save your scene and play.

You'll note now that as the player continues to move, new tiles will spawn as you continue; if you switch to the Scene tab while playing, you'll see that as the ball passes the tiles they will destroy themselves:

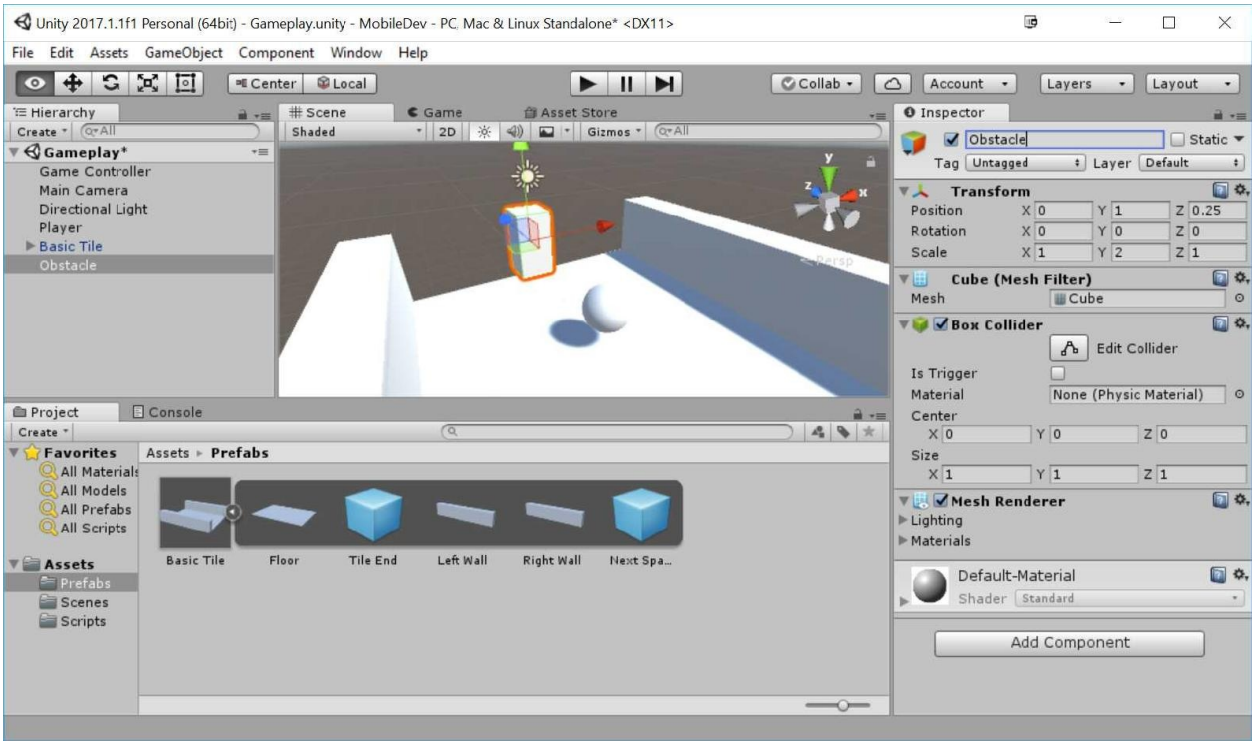

Unity 2017.1.1f1 Personal (64bit) - Gameplay.unity - MobileDev - PC, Mac & Linux Standalone* <DX11>
File Edit Assets GameObject Component Window Help
Center Local
Collab
Account
Layers
Layout
Hierarchy
Create
Gameplay*
Game Controller
Main Camera
Directional Light
Player
Basic Tile(Clone)
Scene
Game
Asset Store
Shaded
2D
Gizmos
Persp
Inspector
Player
Static
Tag Untagged
Layer Default
Transform
Position
Rotation
Scale
Sphere (Mesh Filter)
Mesh Sphere
Sphere Collider
Edit Collider
Is Trigger
Material None (Physic Material)
Center
Radius 0.5
Mesh Renderer
Lighting
Materials
Rigidbody
Mass 1
Drag 0
Angular Drag 0.05
Use Gravity
Is Kinematic
Interpolate None
Collision Detection Discrete
Project
Console
Favorites
All Materials
All Models
All Prefabs
All Scripts
Assets
Prefabs
Scenes
Scripts
Assets ▸ Prefabs
Basic Tile
Floor
Tile End
Left Wall
Right Wall
Next Spa...

Creating obstacles

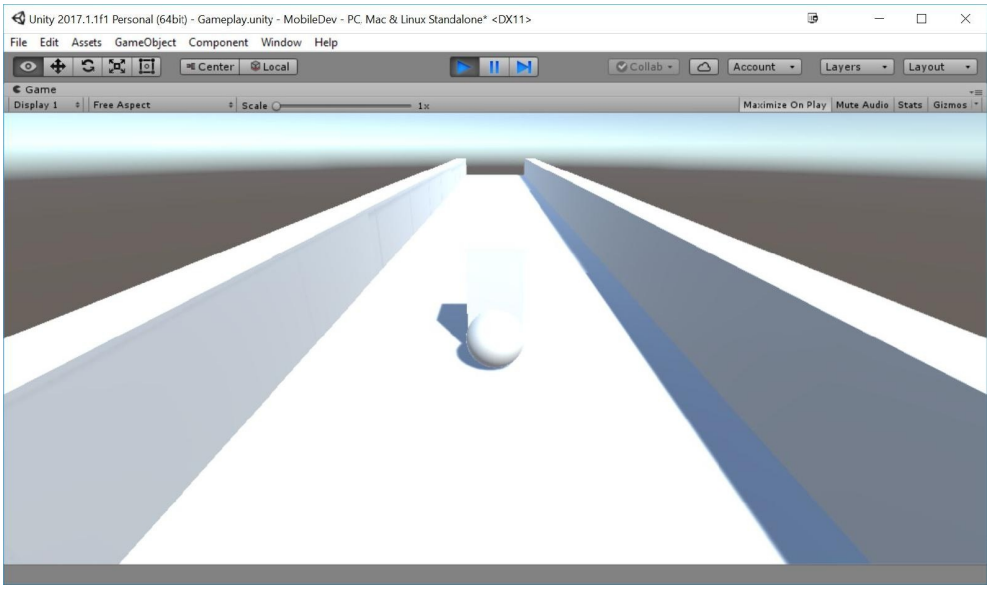
It's great that we have some basic tiles, but it's a good idea to give the player something to do or, in our case, something to avoid. In this section, you'll learn how to customize your tiles to add obstacles for your player to avoid:

1. So, just like we created a prefab for our basic tile, we will create a single obstacle through code. I want to make it easy to see what the obstacle will look like in the world and make sure that it's not too large, so I'll drag and drop a `Basic Tile` prefab back into the world.

2. Next, we will create a cube by going to GameObject | 3D Object | Cube. We will then name this object `Obstacle`. Change the Y Scale to `2` and position it above the platform at `(0, 1, .025)`:



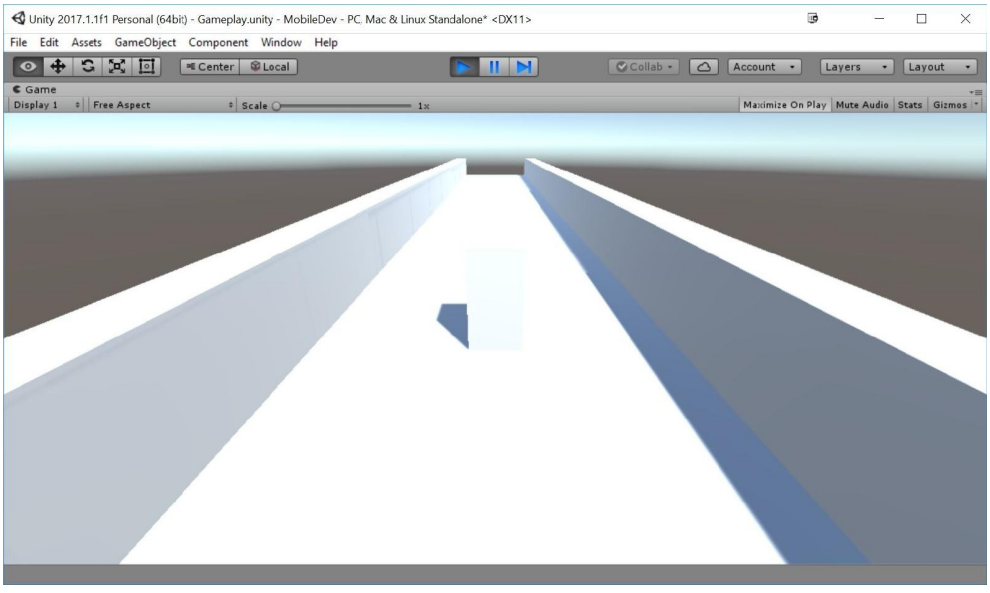
3. We can then play the game to see how this'll work:



4. As you can see in the preceding screenshot, the player gets stopped, but nothing really happens. In this instance, we want the player to lose when he hits this and then restart the game; so, to do that, we'll need to write a script. From the Project window, go to the `Scripts` folder and create a new script called `ObstacleBehaviour`. We'll use the following code:

```
using UnityEngine;
using UnityEngine.SceneManagement; // LoadScene

public class ObstacleBehaviour : MonoBehaviour {

    [Tooltip("How long to wait before restarting the game")]
    public float waitTime = 2.0f;

    void OnCollisionEnter(Collision collision)
    {
        // First check if we collided with the player
        if (collision.gameObject.GetComponent<PlayerBehaviour>())
        {
            // Destroy the player
            Destroy(collision.gameObject);

            // Call the function ResetGame after waitTime has passed
            Invoke("ResetGame", waitTime);
        }
    }

    /// <summary>
    /// Will restart the currently loaded level
    /// </summary>
    void ResetGame()
    {
        // Restarts the current level
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
}
```

5. Save the script and return to the editor, attaching the script to the Obstacle property we just created.
6. Save your scene and try the game:



As you can see in the preceding screenshot, once we hit the obstacle,

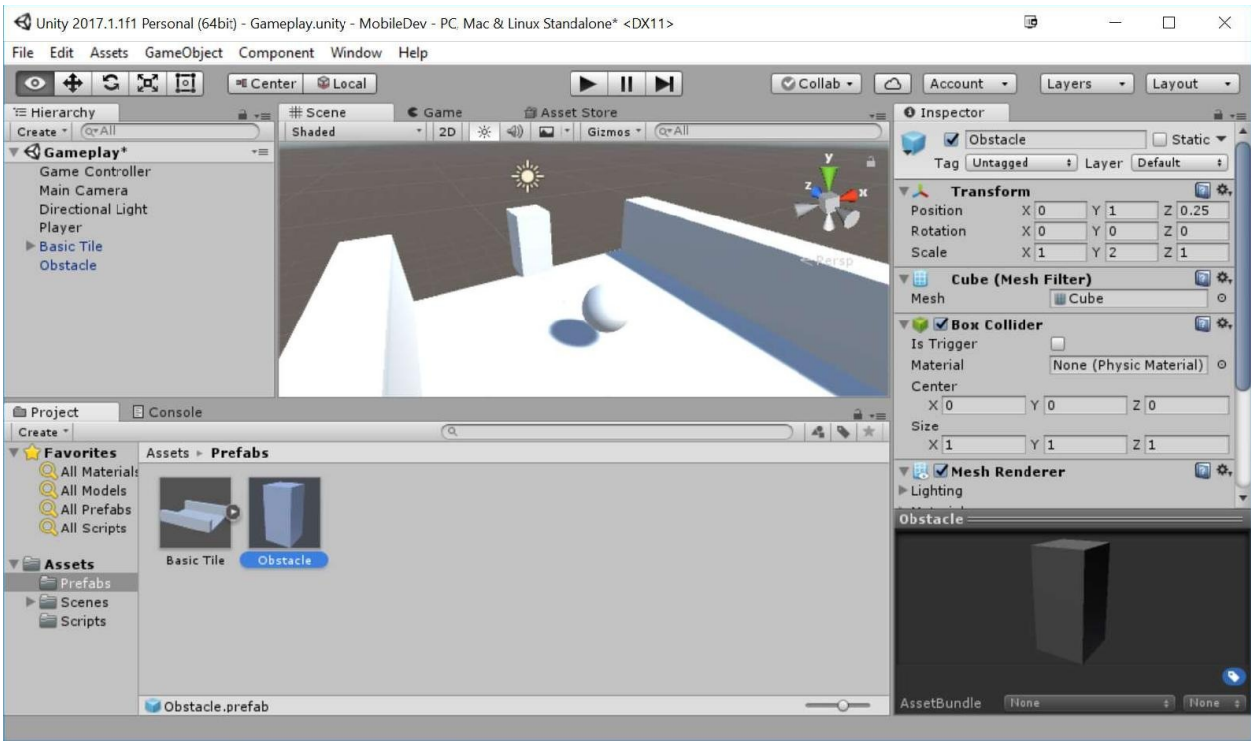
the player gets destroyed, and then after a few seconds, the game starts up again. You'll learn to use particle systems and other things to polish this up, but at this point, it's functional, which is what we want.

You may note that when the level is reloaded, there are some lighting issues in the editor. The game will still work correctly when exported, but this may be a minor annoyance.

7. (Optional) To fix this, go to Window | Lighting | Settings. Uncheck the Auto Generate option from there, and click on Generate Lighting--once it is finished, our issue should be solved.

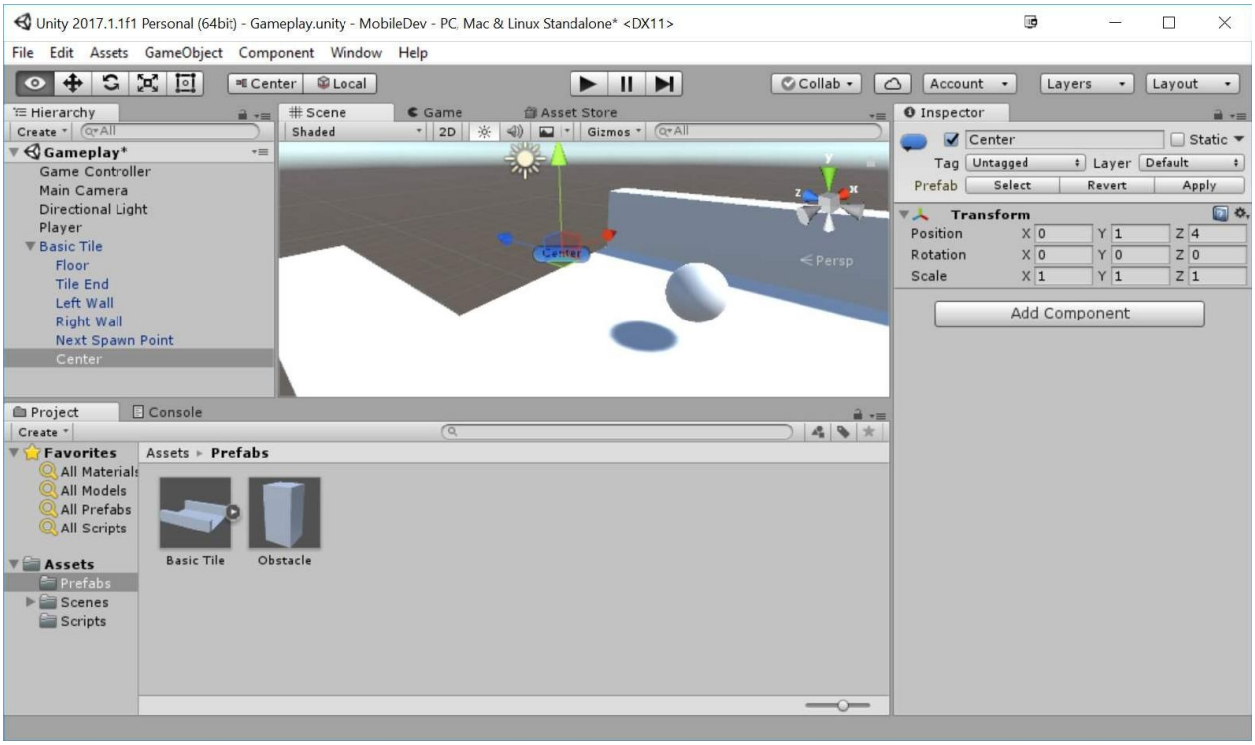
It isn't an issue with our game, but if you employ the fix above in your own titles, you must remember to go here every time you alter a game level and rebuild the lightmap for it to be updated correctly.

8. Now that we know it works correctly, we can make it a prefab. Just as we did with the original tile, go ahead and drag and drop it from the Hierarchy into the Project tab and into the `Prefabs` folder:



9. Next, we will remove the `Obstacle`, as we'll spawn it upon creating the tile.

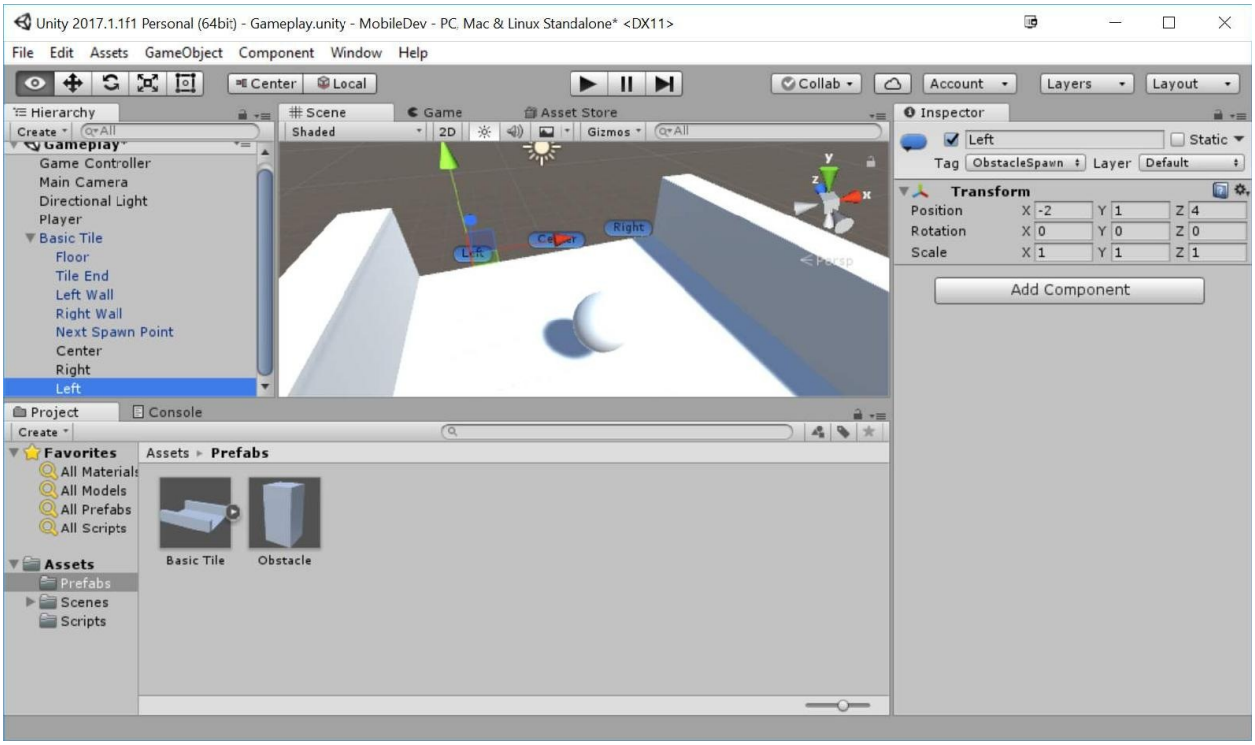
10. We will make markers to indicate where we would possibly like to spawn our obstacles. Duplicate the `Next Spawn Object` object and move the new one to `(0, 1, 4)`. We will then rename the object as `Center`. Afterwards, click on the icon on the top left of the blue box and then select the blue color. Upon doing this, you'll see that we can see the text inside the editor, if we are close to the object (but it won't show up in the Game tab by default):



11. We want a way to get all of the potential spawn points we will want in case we decide to extend the project in the future, so we will assign a **tag** as a reference to make those objects easier to find. To do that at the top of the Inspector window, click on the tag dropdown and select Add Tag. From the menu that pops up, press the + button and then name it `ObstacleSpawn`.

For more information on tags and why we'd want to use them, check out https://docs.unity3d.com/Manual/Tags.html.

12. Go ahead and duplicate this twice and name the others `Left` and `Right`, respectively, moving them 2 units to the left and right of the center to become other possible obstacle points:



13. Note that these changes don't affect the original prefab, by default; that's why the objects are currently black text. To make this happen, select `Basic Tile`, and then in the Inspector window under the Prefab section, click on Apply.
14. Now that the prefab is set up correctly, we can go ahead and remove it by selecting it and pressing *Delete*.
15. We then need to go into the `GameController` script and modify it to have the following code:

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

/// <summary>
/// Controls the main gameplay
/// </summary>

public class GameController : MonoBehaviour
{
    [Tooltip("A reference to the tile we want to spawn")]
    public Transform tile;

    [Tooltip("A reference to the obstacle we want to spawn")]
    public Transform obstacle;

    [Tooltip("Where the first tile should be placed at")]
    public Vector3 startPoint = new Vector3(0, 0, -5);

    [Tooltip("How many tiles should we create in advance")]
    [Range(1, 15)]
```

```
    public int initSpawnNum = 10;

    [Tooltip("How many tiles to spawn initially with no obstacles")]
    public int initNoObstacles = 4;

    /// <summary>
    /// Where the next tile should be spawned at.
    /// </summary>
    private Vector3 nextTileLocation;

    /// <summary>
    /// How should the next tile be rotated?
    /// </summary>
    private Quaternion nextTileRotation;

    /// <summary>
    /// Used for initialization
    /// </summary>
    void Start()
    {
        // Set our starting point
        nextTileLocation = startPoint;
        nextTileRotation = Quaternion.identity;

        for (int i = 0; i < initSpawnNum; ++i)
        {
            SpawnNextTile(i >= initNoObstacles);
        }
    }

    /// <summary>
    /// Will spawn a tile at a certain location and setup the next
    position
    /// </summary>
    public void SpawnNextTile(bool spawnObstacles = true)
    {
        var newTile = Instantiate(tile, nextTileLocation,
                                  nextTileRotation);

        // Figure out where and at what rotation we should spawn
        // the next item
        var nextTile = newTile.Find("Next Spawn Point");
        nextTileLocation = nextTile.position;
        nextTileRotation = nextTile.rotation;

        if (!spawnObstacles)
            return;

        // Now we need to get all of the possible places to spawn the
        // obstacle
        var obstacleSpawnPoints = new List<GameObject>();

        // Go through each of the child game objects in our tile
        foreach (Transform child in newTile)
        {
            // If it has the ObstacleSpawn tag
            if (child.CompareTag("ObstacleSpawn"))
            {
                // We add it as a possibilty
                obstacleSpawnPoints.Add(child.gameObject);
            }
        }

        // Make sure there is at least one
```

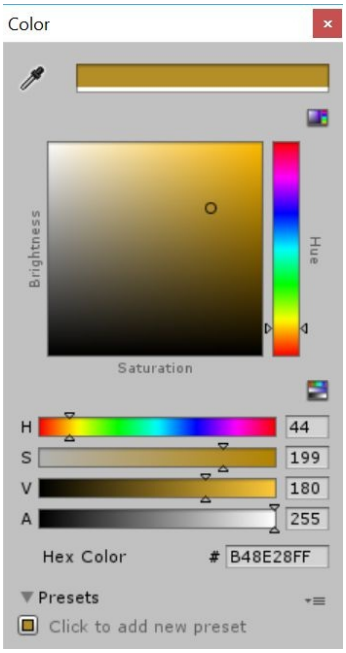
```
        if (obstacleSpawnPoints.Count > 0)
        {
            // Get a random object from the ones we have
            var spawnPoint = obstacleSpawnPoints[Random.Range(0,
                                        obstacleSpawnPoints.Count)];

            // Store its position for us to use
            var spawnPos = spawnPoint.transform.position;

            // Create our obstacle
            var newObstacle = Instantiate(obstacle, spawnPos,
                                        Quaternion.identity);

            // Have it parented to the tile
            newObstacle.SetParent(spawnPoint.transform);
        }
    }
}
```

Note that we modified the `SpawnNextTile` function to now have a default parameter set to `true`, which will tell us if we want to spawn obstacles or not. At the beginning of the game, we may not want the player to have to start dodging immediately, but we can tweak the value to increase or decrease the number we are using.

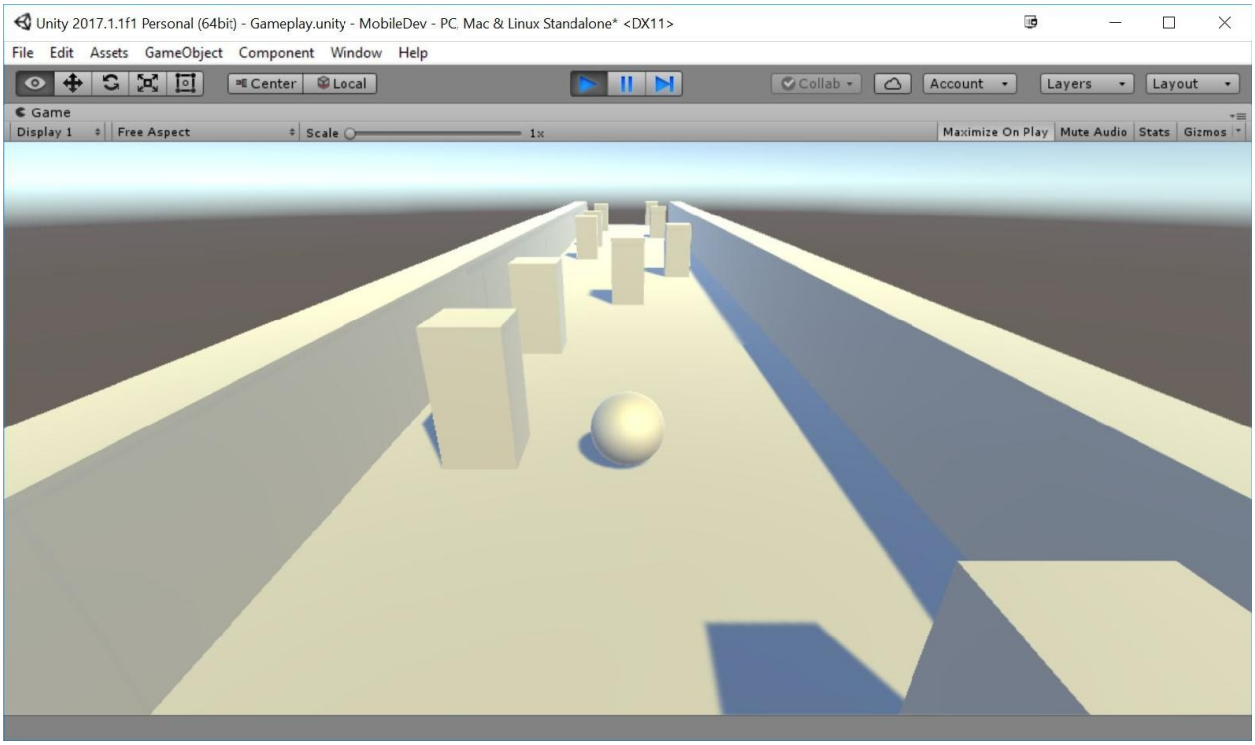
16. Save the script and go back to the Unity editor. Then, assign the Obstacle variable in the Inspector with the obstacle prefab we created previously.
17. It's a bit hard to see things currently due to the default light settings, so let's go to the Hierarchy window and select the `Directional Light` object.

    A directional light acts similarly to how the sun works on earth, shining everywhere from a certain rotation.

18. With the default settings the light is too bright, making it difficult to see, so we can just change the `Color` to be darker. I used the following:



19. Save your scene and play the game:



For more information on directional lights and the other lighting types that Unity has, check out: https://unity3d.com/learn/tutorials/topics/graphics/light-types?playlist=17102.

As you can see in the preceding screenshot, we now have a number of obstacles for our player to avoid, and due to how the player works, he will gradually get faster and faster, causing the game to increase in difficulty over time.

Summary

There we have it! A solid foundation, but just that, a foundation. However, that being said, we covered a lot of content in this chapter. We discussed how to create a new project in Unity, we built a player that will move continuously, as well as take inputs to move horizontally. We then discussed how we can use Unity's attributes and XML comments to improve our code quality and help us when working with teams. We also covered how to have a moving camera. We created a tile-based level design system, where we created new tiles as the game continued, randomly spawning obstacles for the player to avoid.

Throughout this book, we will explore more that we can do to improve this project and polish it, while adapting it to being the best experience possible on mobile platforms. However, before we get to that, we'll actually need to figure out how to deploy our projects.

Setup for Android and iOS Development

We now have a project to start off with, but currently it's built with a PC in mind. Since this book is about mobile development, it's very important to have the game working on the device itself before we get too far ahead.

Chapter overview

In this chapter, we will go through all of the setup that we'll need in order to deploy the project in its current state onto our mobile devices. At the time of writing this book, mobile development is typically done either for Android or iOS, so we will cover that.

Our objectives

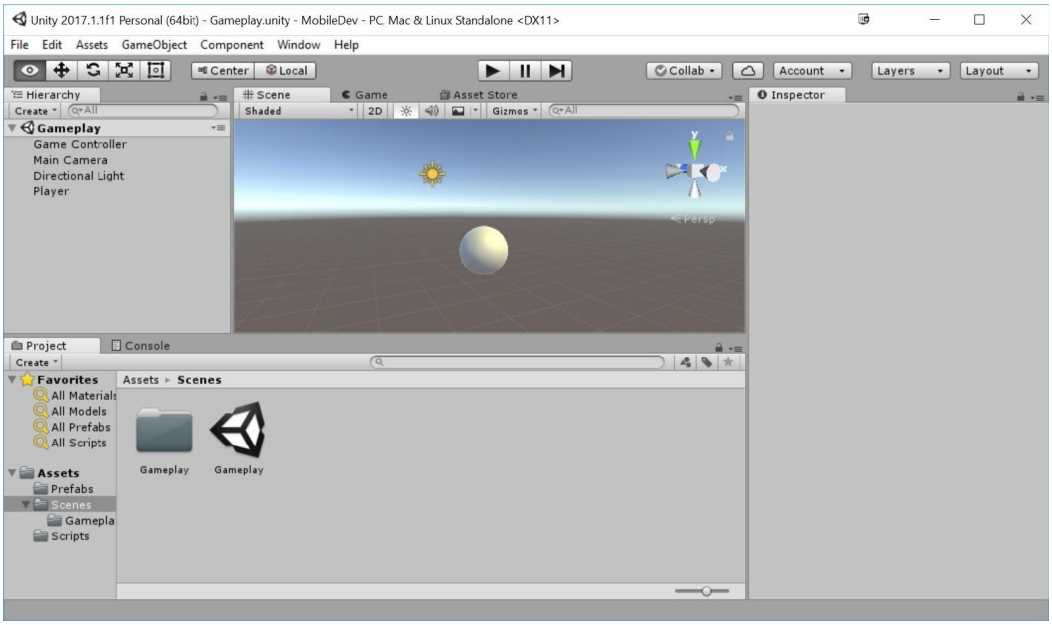
This chapter will be split into a number of topics. It will contain a simple step-by-step process from beginning to end. The following is the outline of our tasks:

- Introduction to build settings
- Building a project for a PC
- Installing the Java Development Kit and Android SDK
- Exporting your project for Android
- Putting the project on your Android device
- Unity iOS installation and Xcode setup
- Building a project for iOS

Introduction to build settings

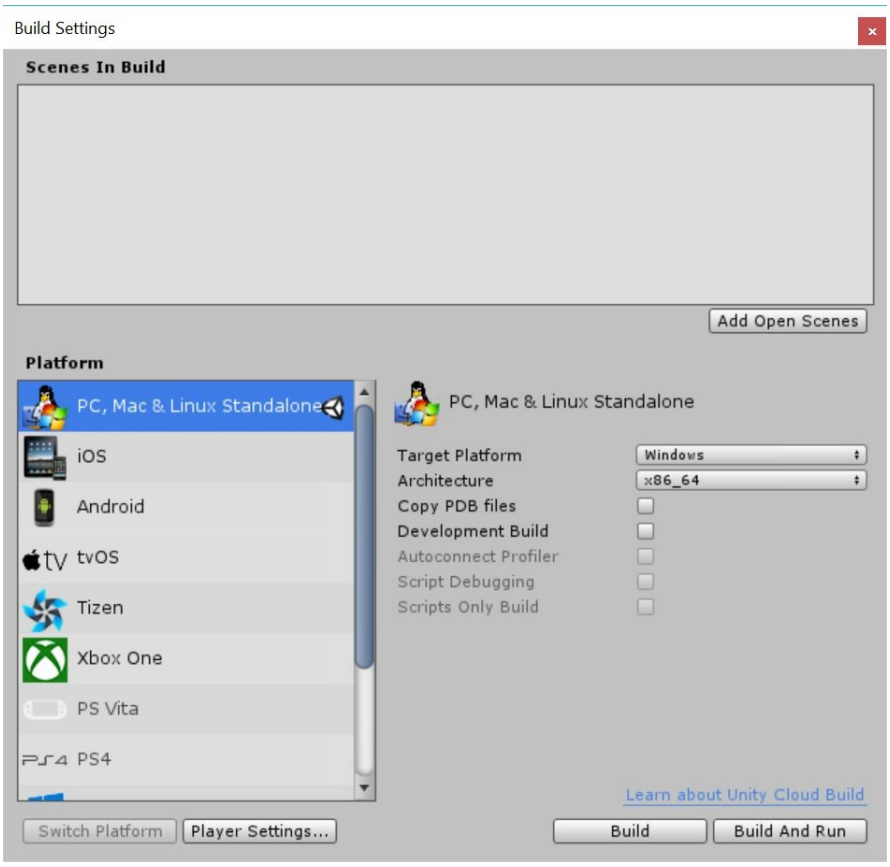
There are times during development that you may want to see what your game looks like if you build it outside of the editor. It can give you a sense of accomplishment; I know, I felt that way the first time I pushed a build to a console devkit. Whether it's for PC, Mac, Linux, web player, mobile, or console, we have to go through the same menu, the Build Settings menu:

1. Start off by opening up the project that we created in the preceding chapter. In addition, open the scene we created (`Gameplay.unity`, which is inside the `Scenes` folder):



2. From here, we will open the Build Settings menu by selecting File | Build Settings.

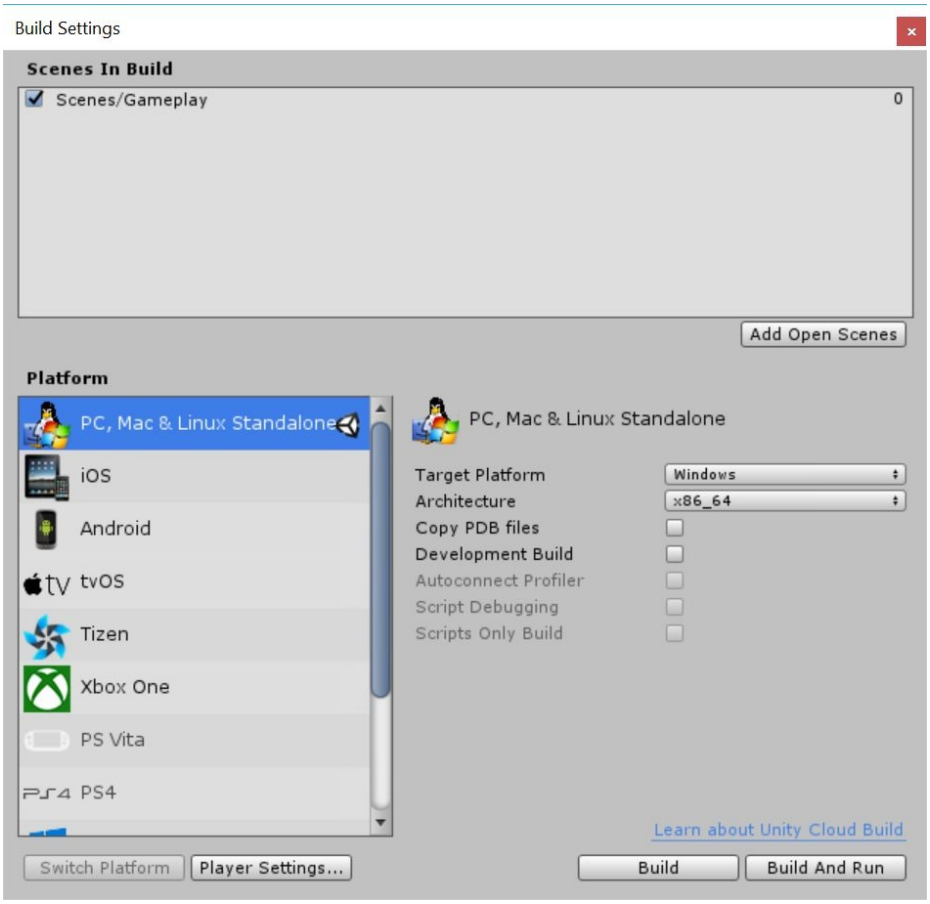
You may alternatively press Ctrl + Shift + B or command + Shift + B to bring the menu up as well.



In the preceding image, you will notice the Build Settings menu came menu contains three sections:

- Scenes in Build: This window contains the scenes in our project that we want to include when we build our project. This ensures that things such as test levels won't be included unless you specify.
- Platform: This is a list of all of the platforms that you can export your game to. The Unity logo shows up on the current platform you're compiling for. In order to change your platform, you'll need to select it from this list and then click on the Switch Platform button below the list.
- Options: To the right of the Platform section, you'll see some settings, which can be tweaked based on how you want the build to work with certain options that change based on the platform you will work with.

3. By default, we have no scenes in our build, so let's go ahead and change that. Go ahead and click on the Add Open Scenes button; you should see the `Gameplay` level appear in the list at index `0`, which means that when your game is played, this level will be the first one to load:

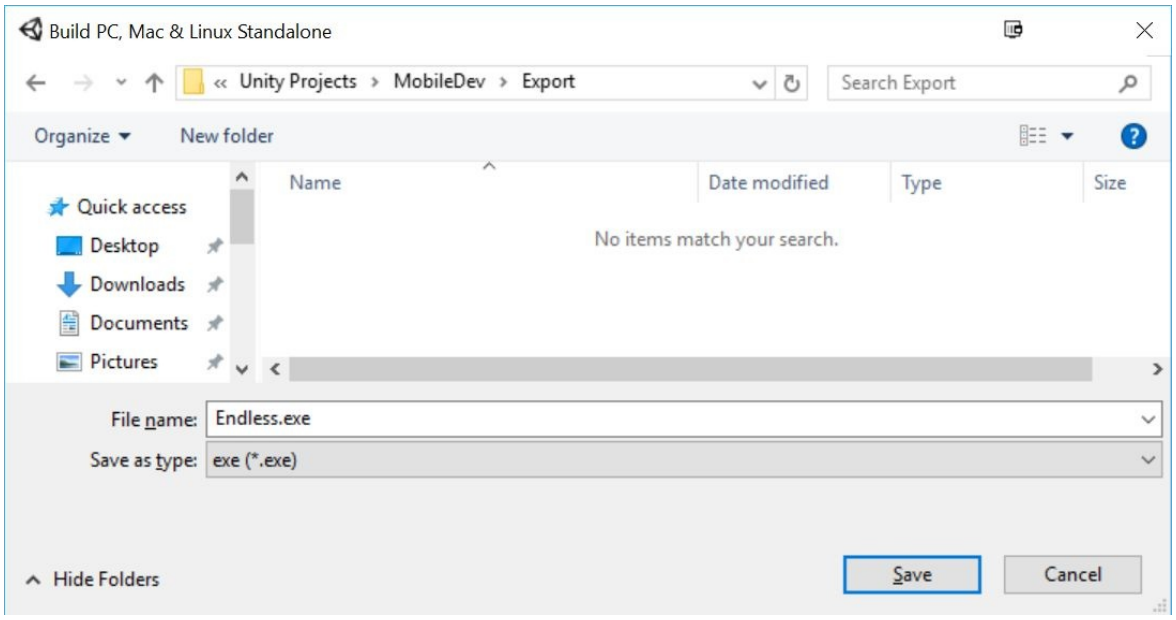


You may also add scenes to the Scenes in Build section by dragging an dropping them from the Project window. You may also drag the scenes them however you wish.

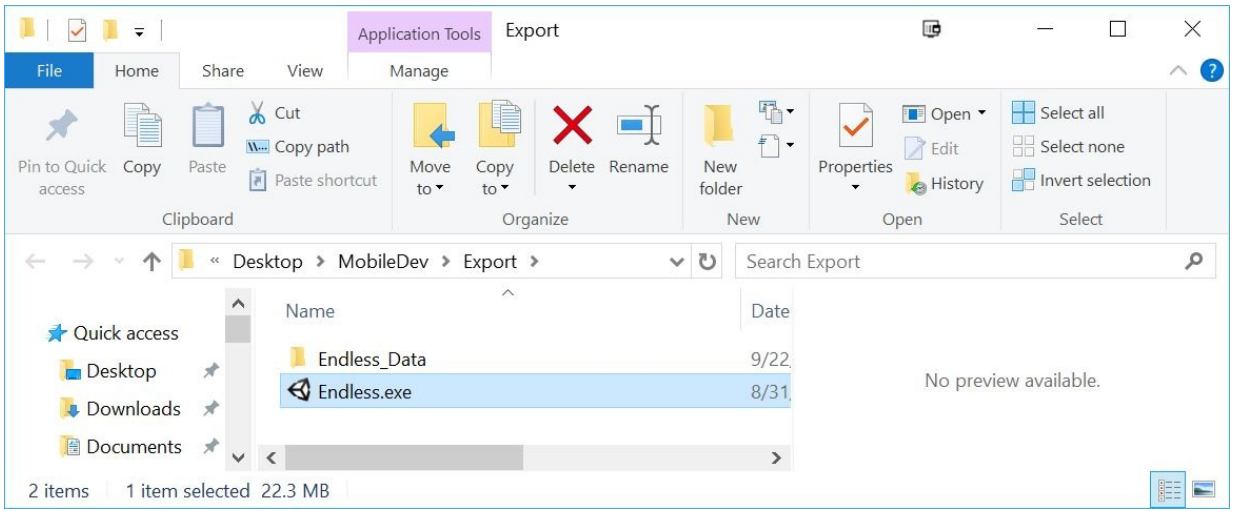
Building a project for PC

By default, our platform is set to PC, Mac & Linux Standalone. Just to verify that everything is working correctly, let's go ahead and get the game working on our own platform before moving to mobile:

1. To get started, we will select the Build option. In my instance, I'll be exporting our project to Windows, but the process is similar for Mac and Linux.
2. Once this is done, a window will pop up asking for a name and a location to put the game in. I'm going to name it `Endless` and put it in a new `Export` folder located in the same folder that contains `Assets` and `Library`, so it won't show up in the Project window, but it will be in the same folder as my project:



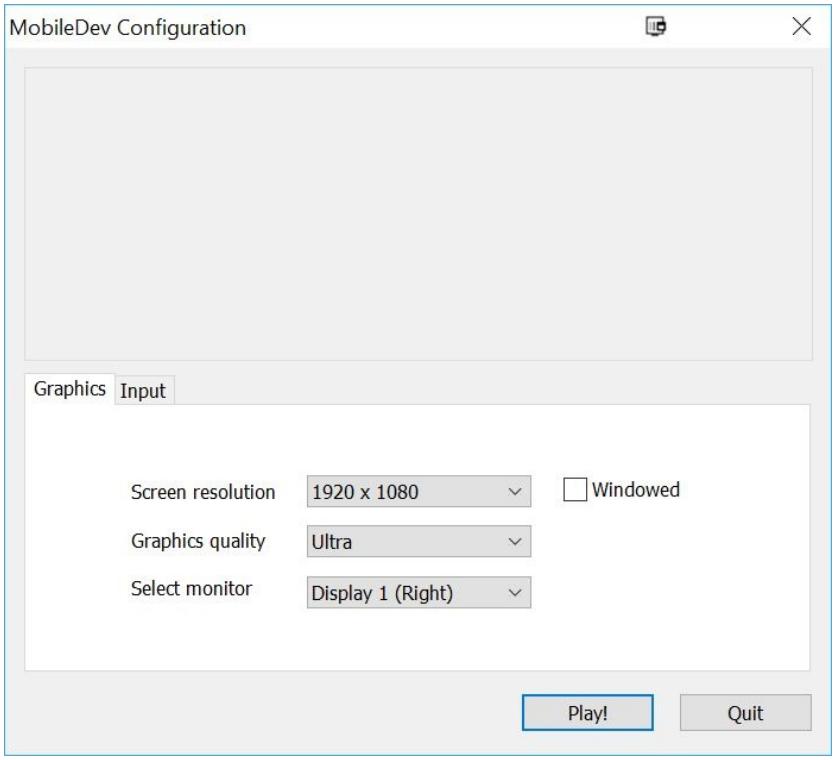
3. Click on Save and wait for it to finish. Once it's done, you should have a window appear, as follows:



We have the executable, but we also have a data folder that contains all the assets for our application (right now, called `Endless_Data`). You must include the data folder with your game, or it will not run.

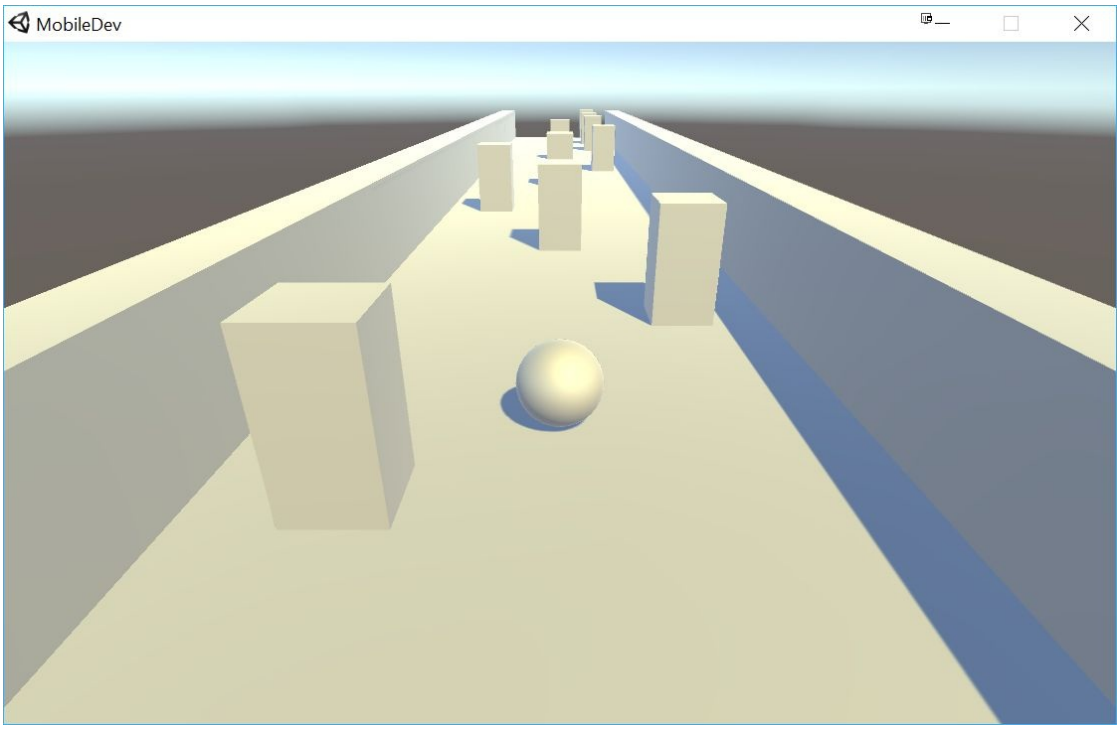
If you build for Mac, it will bundle the app and data all together, so once you export it, all you need to provide is the application.

4. If you double-click on the `.exe` file to run the game, you'll be taken to the following startup menu, as shown in the following screenshot:



This will allow players to customize their Screen resolution values as well as other options, such as what buttons to use for input. This menu will not appear when exporting for mobile.

Anyway, once we click on the Play! button, we'll be taken to the proper game screen, as shown in the following screenshot:



With that, we should be able to control and play the game as we usually would do. This is great!

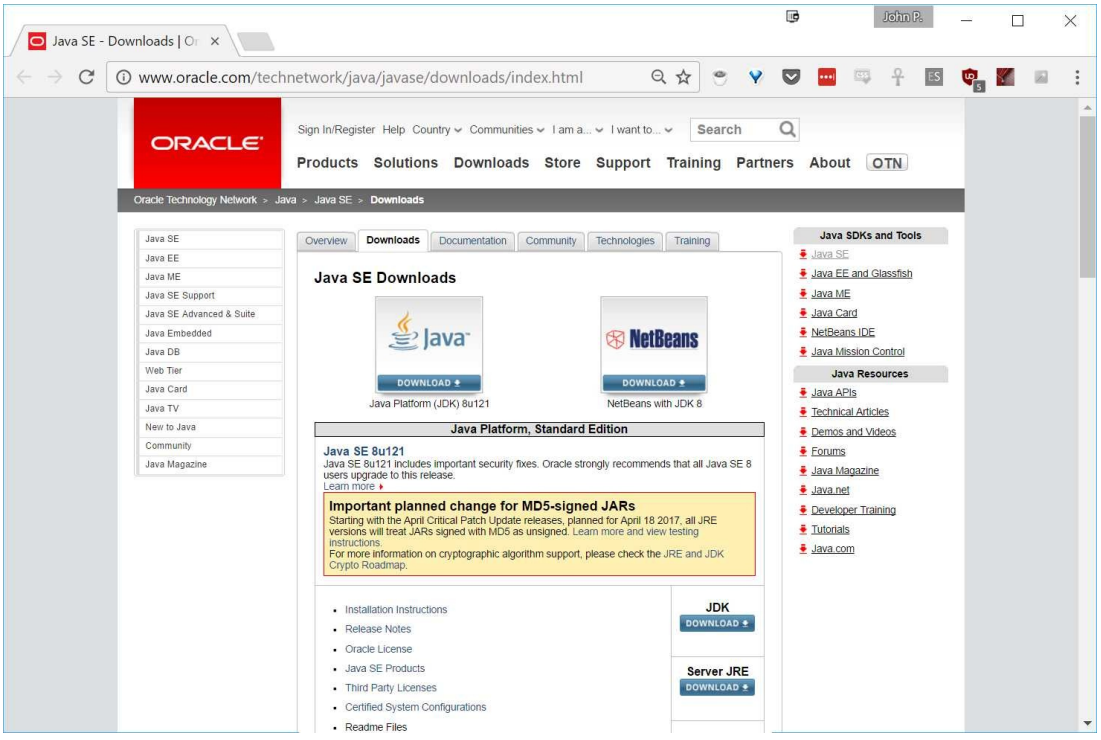
Note that if you made the game full screen, you'll have to use Alt + F4 (command + Q on Mac) to quit the game.

Now that we have talked about the universal ways of building a project, let's dive into specifics for different platforms. In this section, we will discuss getting our project onto an Android device.

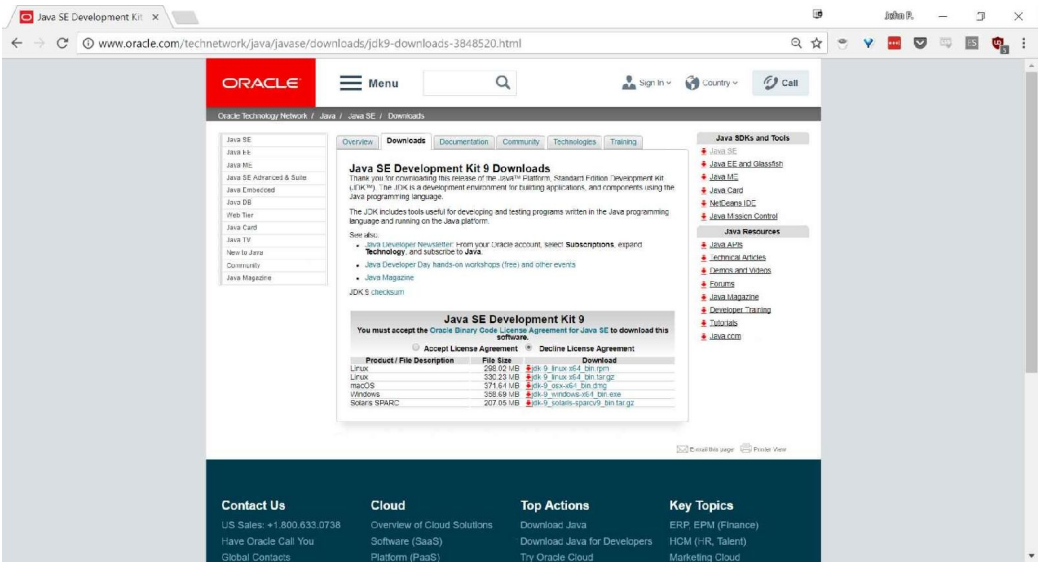
Installing the Java Development Kit (JDK)

The first thing we'll need to do is install the Java Development Kit, which Android uses as their programming language:

1. First, we will need to open up our web browser and visit Java's website at: http://www.oracle.com/technetwork/java/javase/downloads/index.html:

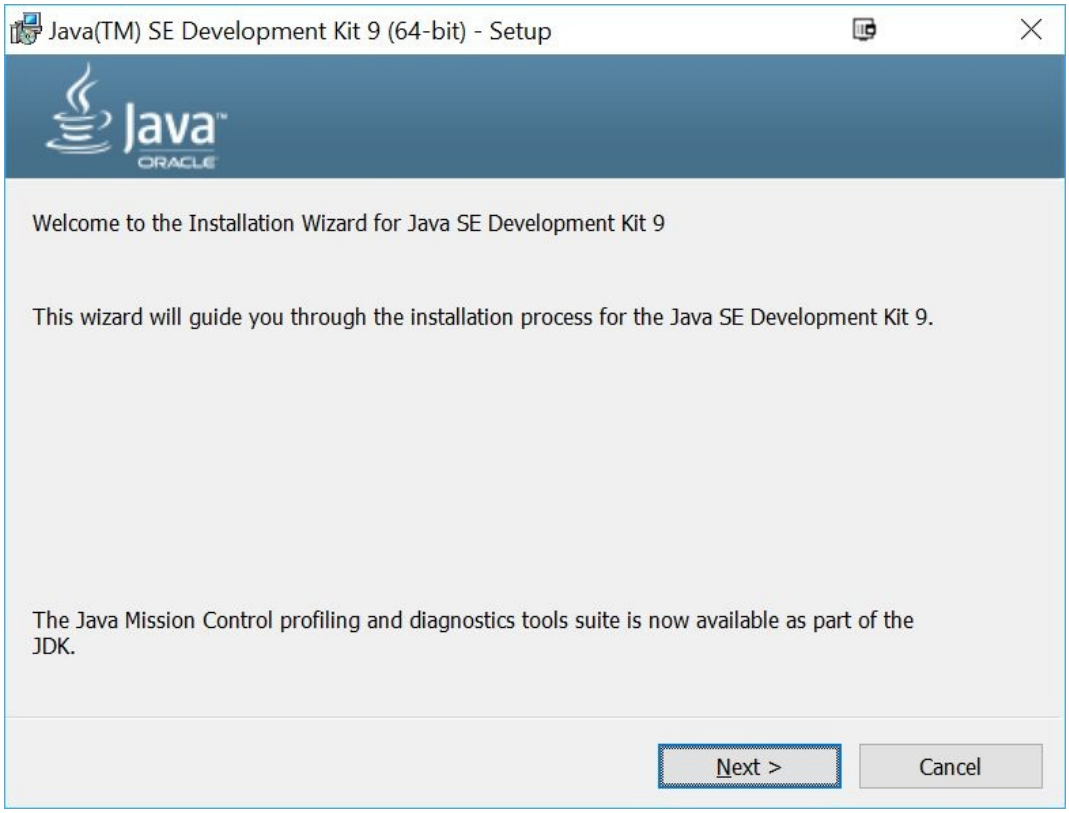


2. From the preceding page, we will then locate the section that says JDK and click on the blue Download button:



3. From there, we will need to first select the Accept License Agreement option and then pick the download platform that we're currently on and click on it to download.

4. Once it's finished, open it up and install the project. If your computer asks whether you want to run the software, go ahead and say Yes:

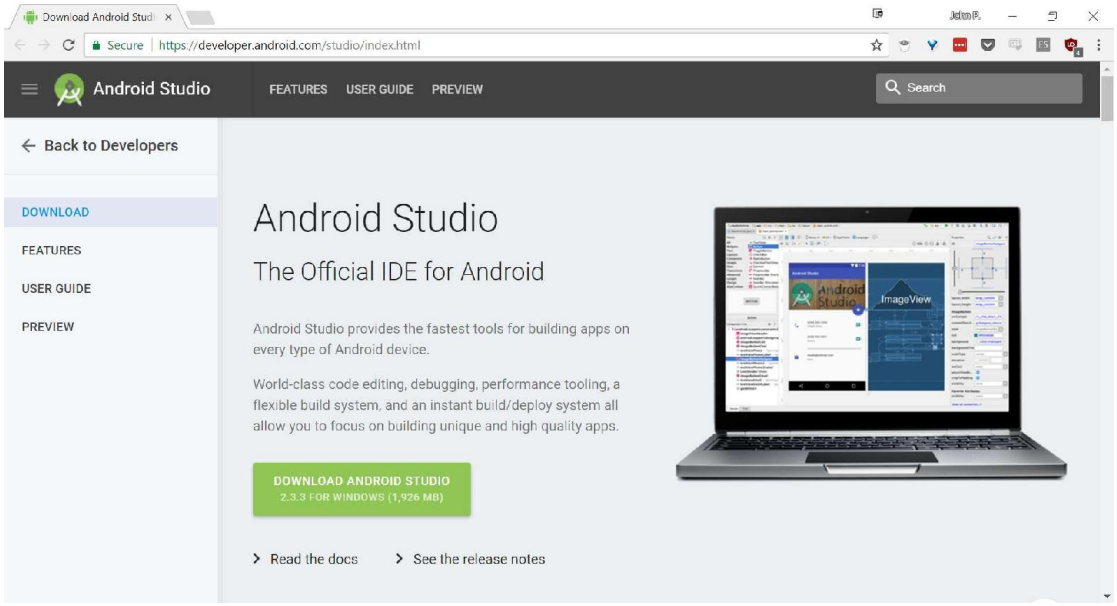


5. Continue through the installation--leave the default values checked, install Java if you need to, and finally, click on Close when it's finished.

Installing the Android SDK

Now that we have the JDK installed, we will also need to install the Android Software Development kit; let's do that now:

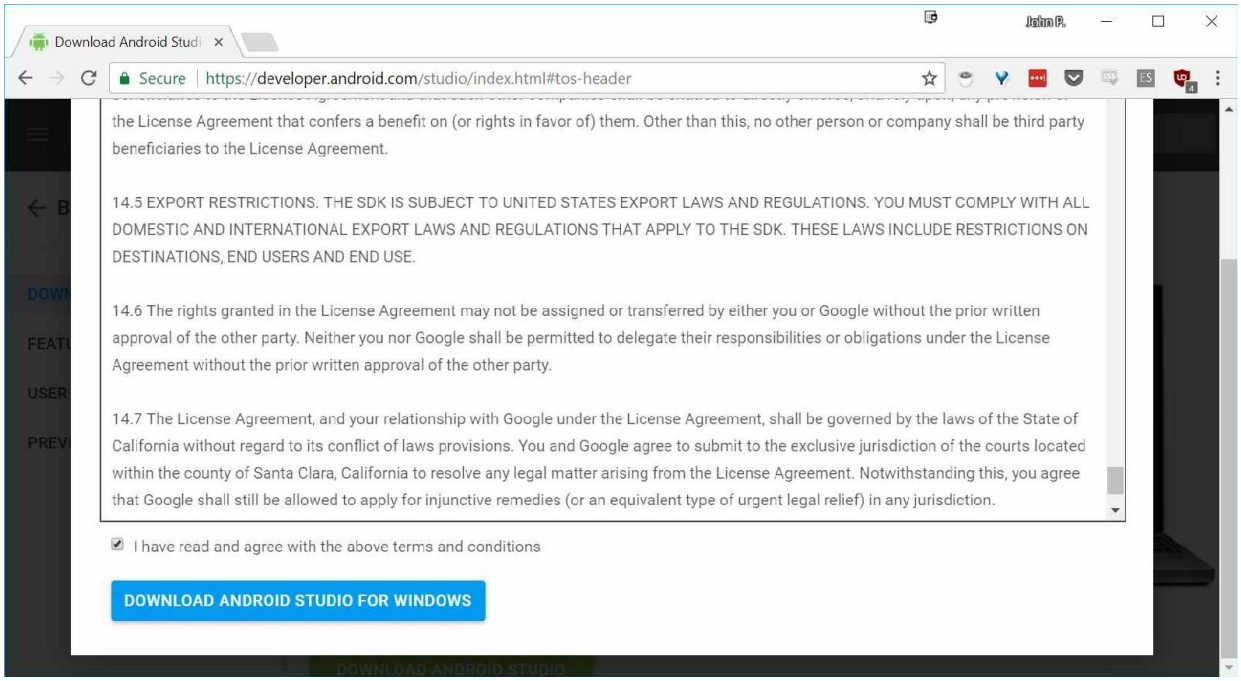
1. In order to install the Android SDK Tools, we'll need to go to the Android Studio page at: https://developer.android.com/studio/index.html:



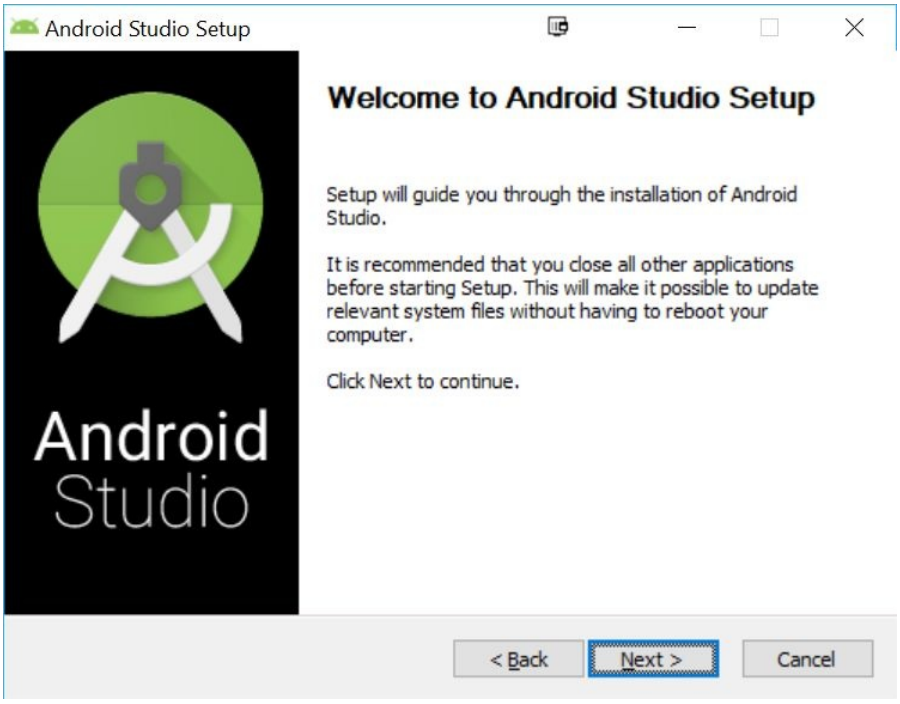
   Android Studio is used to build apps for Android devices when you're building it from scratch, and it also includes the Android SDK with a nice GUI for being able to select what parts we'd like to feature, so we will be installing it.

   However, we will not actually install Android Studio as it's not needed for Unity development:

2. Click on the DOWNLOAD ANDROID STUDIO FOR WINDOWS button. From there, you'll note that some terms and conditions appear. Go ahead and click on the checkbox saying you agree, and from there, click on the blue download button below it:

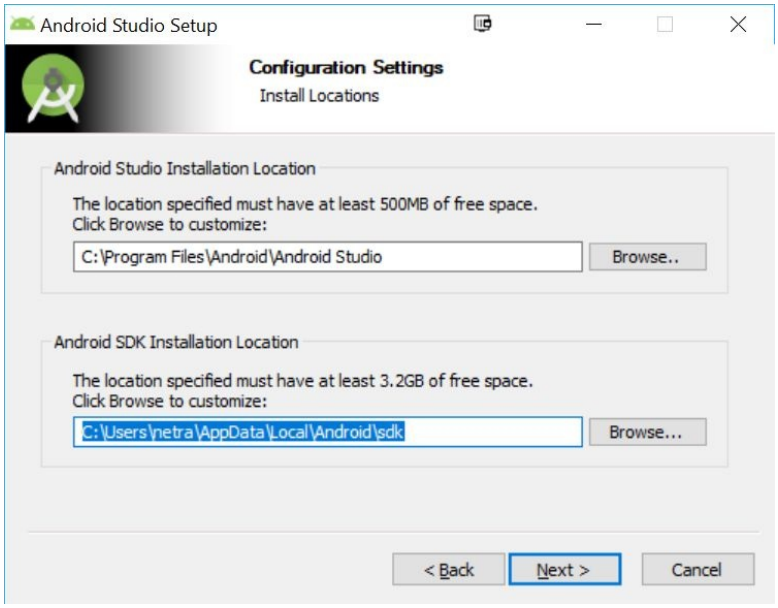


3. Once it is downloaded, open it and allow it to run if you get a security warning and also allow it to make changes to your computer:

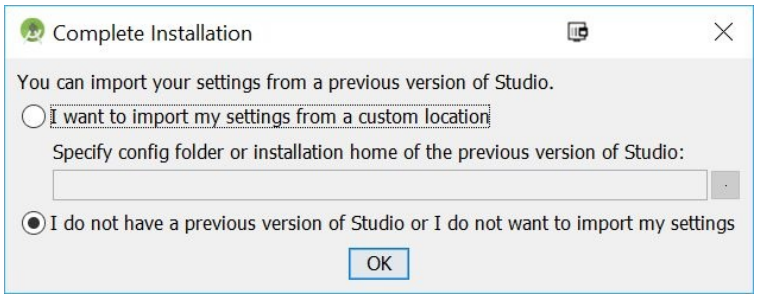


4. Go ahead and click on Next and go through the installation. Note that on one of the screens, there is a setting for the Android SDK Installation Location. Save this, as you'll need it later to bring into Unity. (In the

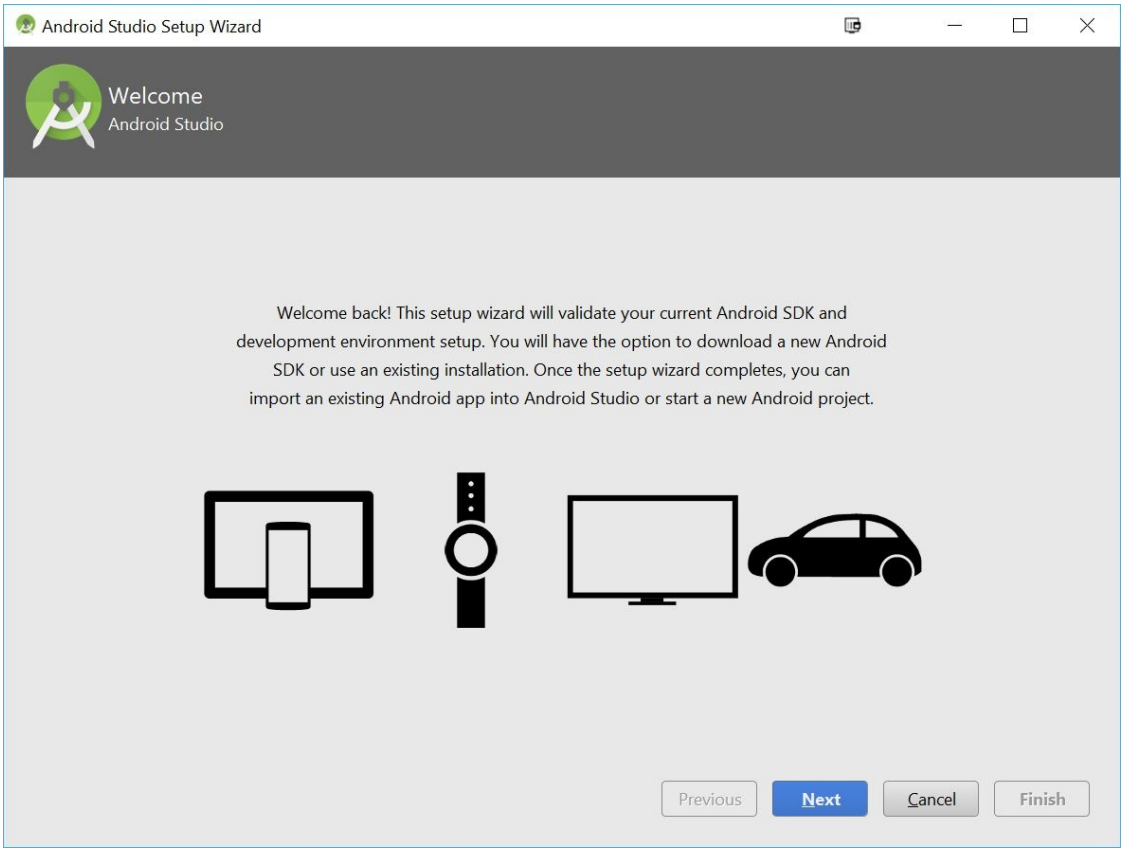
following screenshot, it is at `C:\Users\netra\AppData\Local\Android\sdk`):



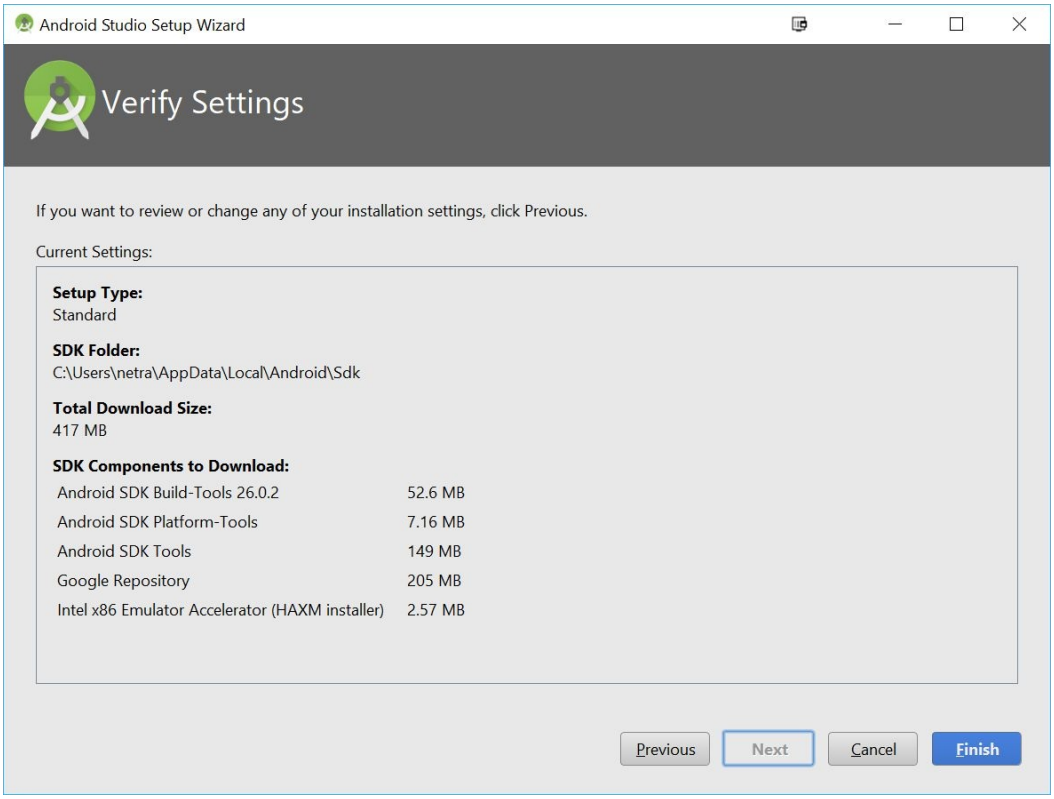
5. Once the installation finishes, you'll note that the Start Android Studio option is checked. Go ahead and click on the Finish button and wait for it to start. You'll see a Complete Installation button; go ahead and click on OK:



6. You'll then be brought to a setup wizard. Go ahead and click on the Next button, as follows:

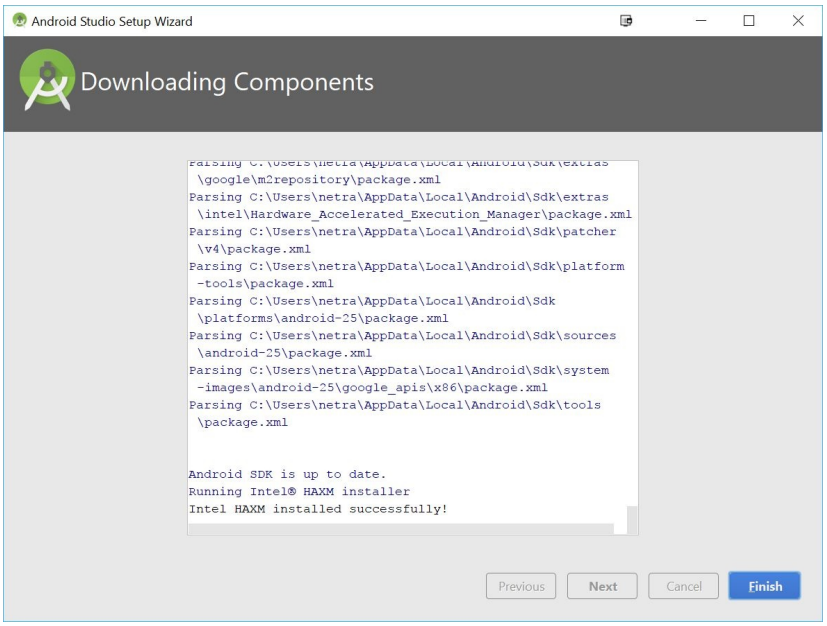


7. Select the standard setup, as it will include the items that we will need to export it to Android in Unity. After clicking on Next, you'll be asked to verify that's what you want. Go ahead and click on the Finish button and wait for everything to be downloaded:



8. During installation, you may be asked whether the Windows Command Processor can make changes to your computer. Go ahead and click on Yes.

9. At the end, you'll get a message that should say that everything was installed successfully. Go ahead and click on the Finish button, and then on the Welcome Android Studio screen, go ahead and close the program by clicking on the X in the top-right corner of the screen:



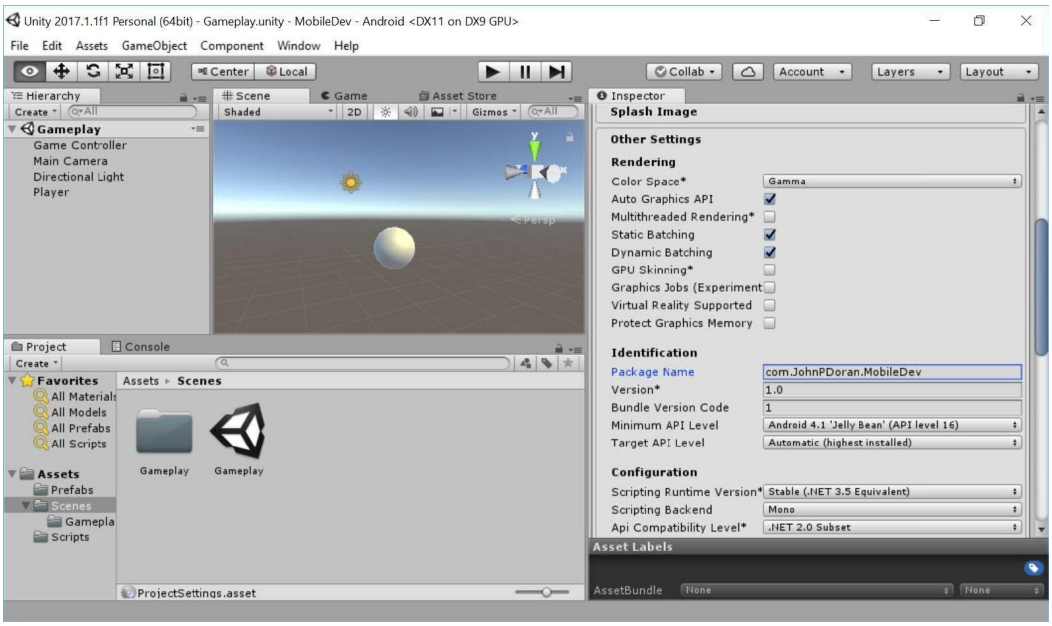
With that, we now have everything we need installed to deploy our game to Android.

Exporting a project for Android

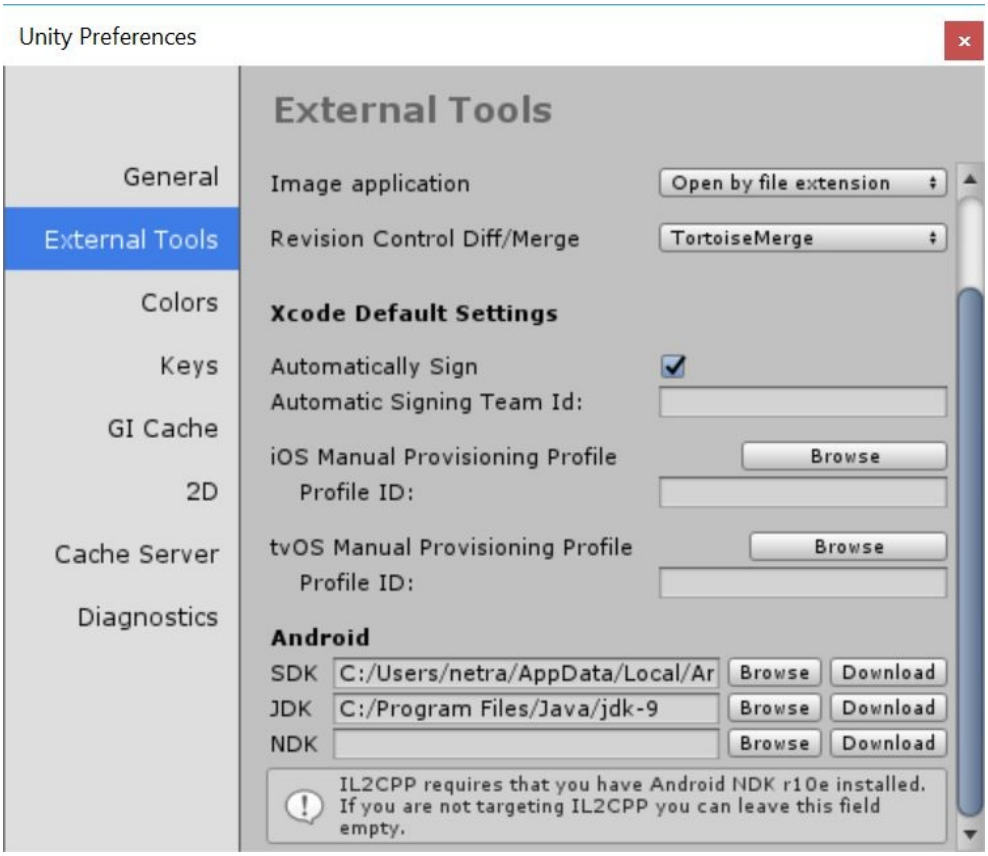
Now that we have all of the setup done, we can open up Unity with our project and export it for Android devices:

1. First of all, if you haven't done so already, you'll need to have selected to add Android Build Support as an option when you are installing Unity. If you did not install it when doing the initial installation, you may reinstall Unity again with those selections checked for the platforms that you're trying to build to.
2. At this point, we will dive into Unity and then move into our Build Settings menu once again by going to File | Build Settings.

3. Click on the Android option from the Platform list and then click on the Switch Platform button to make the change. Note that this will make Unity reimport all of the assets in our game, so this may be time-consuming as you build larger projects.
4. Now, in order to be able to build our project, we must set the bundle identifier for our game, which is a string that identifies the app. It's written like a website in reverse, for example, `com.yourCompanyName.yourGameName`. To modify this, we'll need to open up the Player Settings menu, which we can get to by clicking on the Player Settings button from the bottom of the Build Settings menu or by going into Edit | Project Settings | Player. You'll note that the menu shows up in the Inspector tab.
5. Now that we're in Android mode (note the text on the title bar of the Unity Editor), we can change these properties. We'll discuss more of these in a later chapter, but for right now, scroll down until you get to the Other Settings option, and from there, you'll see the Package Name property. We will change this to something else, for example, I used `com.JohnPDoran.MobileDev`. There's also a Minimum API Level option; make sure that your option is set to the same version as your phone or earlier, depending on what you want to support. Note that the earlier you go, the less things you'll have access to:

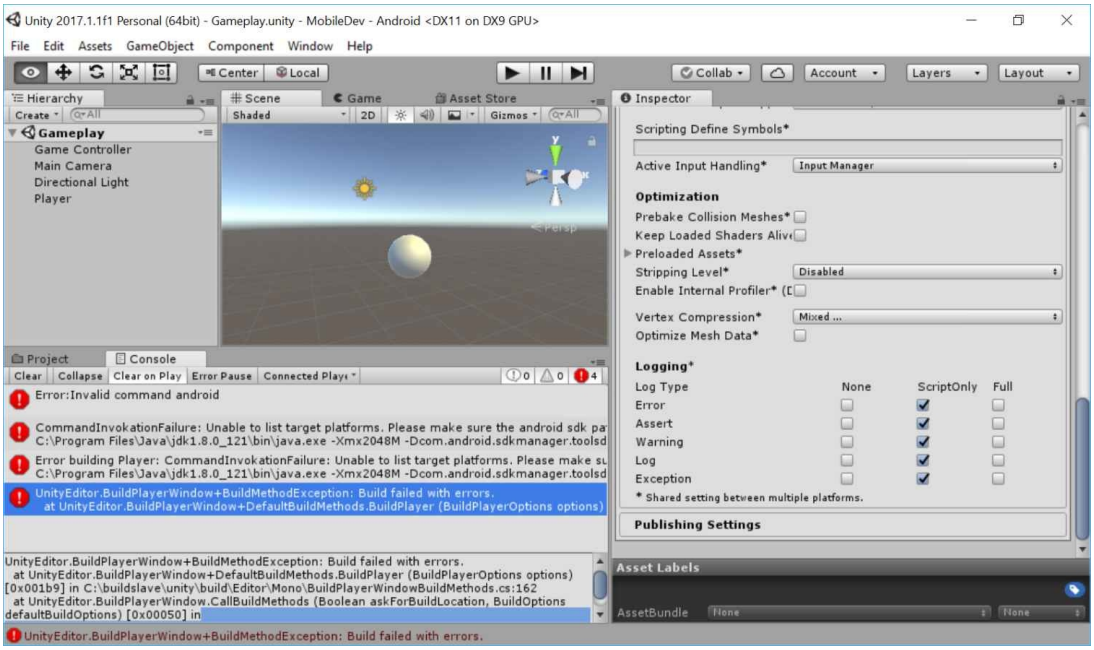


6. We also need to set the location for Android SDK. We can do that by going to Edit | Preferences. From there, go to the External Tools section. Then, set the location of the Android SDK to the folder we specified earlier, and if it isn't set, go ahead and set the JDK folder as well:



7. Open up the Build Settings menu again by going to File | Build Settings. Now, we can try to build the project by clicking on the Build button, saving it in the same `Export` folder we created earlier. We can even name it `Endless` like we did previously, because instead of an `.exe` file, it will be creating a `.apk` file.

    If you are using a future version of Unity, this should work perfectly fine, and you should be taken to the folder in which your game is being exported:

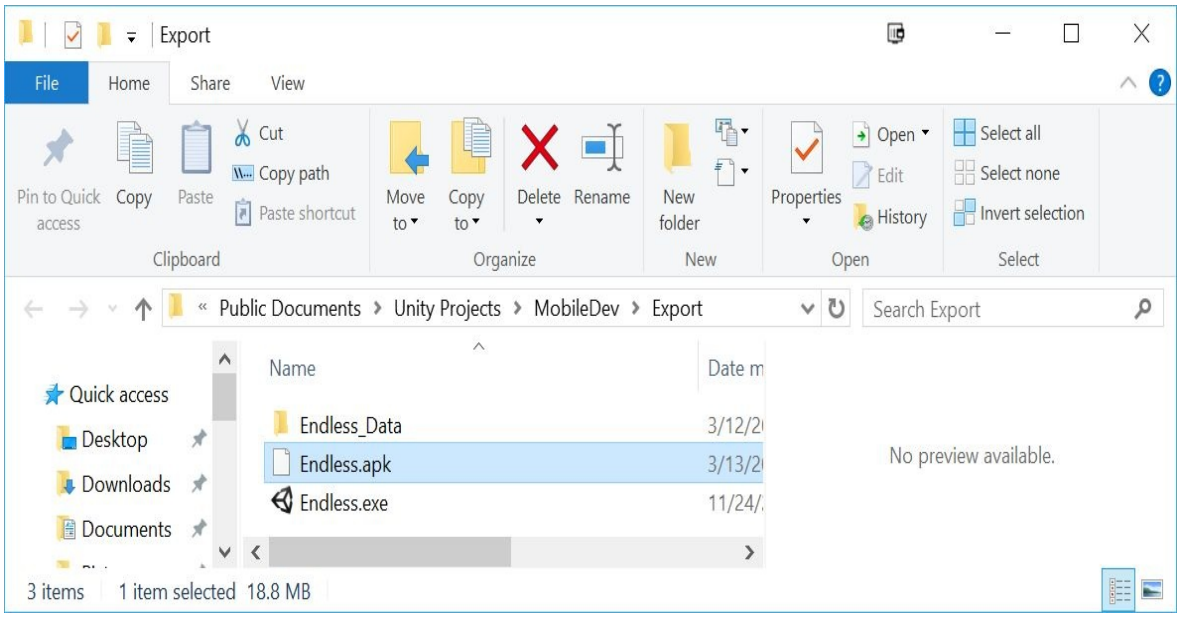


    Now, unfortunately, with the current version of Unity, there is an issue causing Unity to not be able to build a project with the latest version of the Android SDK. If you are reading this book when later versions of Unity have come out, this issue should be resolved and you can skip the next steps, but for those using the version of Unity this book was written for, read on.

For more information on this problem, feel free to visit: https://issuetracker.unity3d.com/issues/android-build-fails-when-the-latest-android-sdk-tools-25-dot-3-1-version-is-used.

8. Download an earlier version of the Android tools from: http://dl-ssl.google.com/android/repository/tools_r25.2.5-windows.zip.

9. Go to your Android SDK folder and rename the `tools` folder as `old tools`. Then, unzip the tools folder from the link into that location to replace the SDK with a working version.
10. Now, build the game again and wait for it to complete the operation:



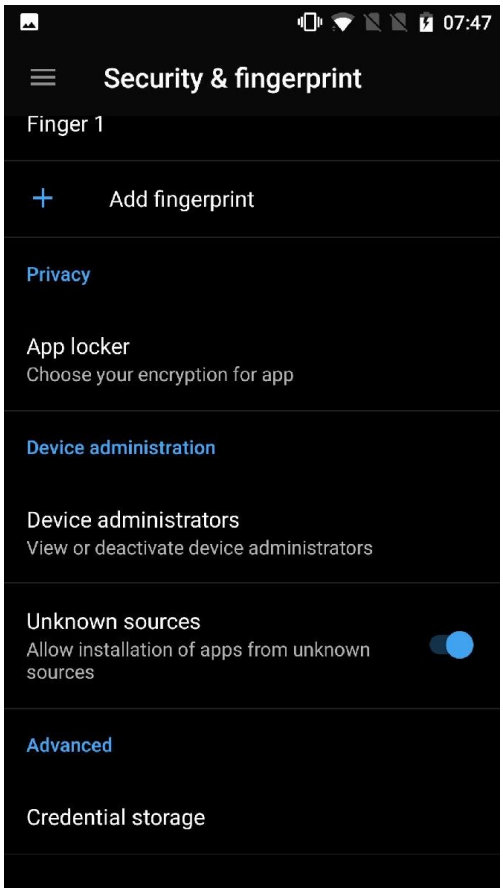
Wait a bit and once it's finished, you should have a new `.apk` file located in the folder.

Putting the project on your Android device

Of course, just having the APK doesn't do much if we can't put it on our actual phone; so, in this section, we will enable our phone to test the game on our device:

1. On your Android device, you'll need to go to your Settings app.
2. From there, scroll down till you get to the Security | Security & fingerprint section or similar, and then tap on it to go into the menu.

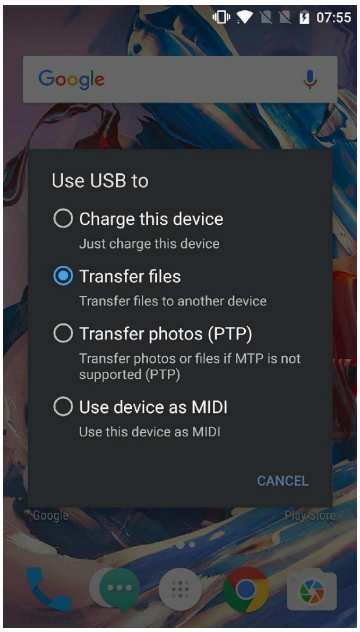
3. Inside there, you'll see a section called Unknown sources, which you'll want to enable:



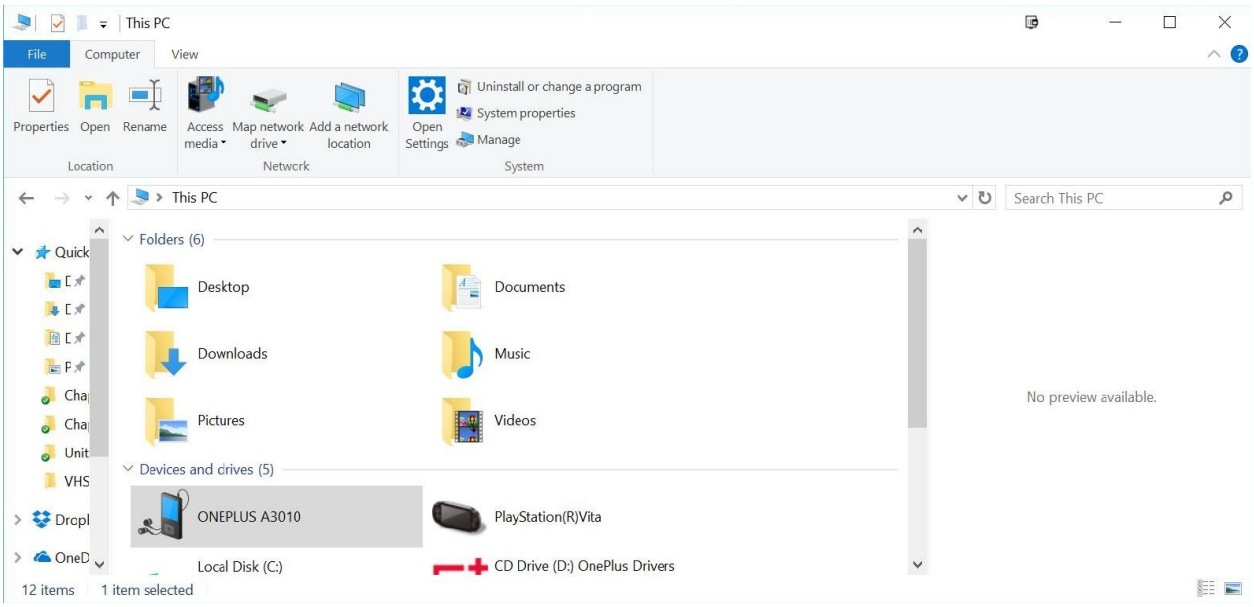
With this enabled, your device can now install the `.apk` file, but now yc need to move your game over onto the device so that you can install it easiest way is to transfer it to your device via USB; we'll do that now.

For those of you who'd rather not use USB, I will sometimes use a cloud storage app, such as Dropbox, to upload my `.apk` file and then download it from the app and then install that way. There's also another tool called ADB, which can send files to your phone via USB or Wi-Fi. For more info on that and the rest of the Android build process, check out: https://docs.unity3d.com/Manual/android-BuildProcess.html.

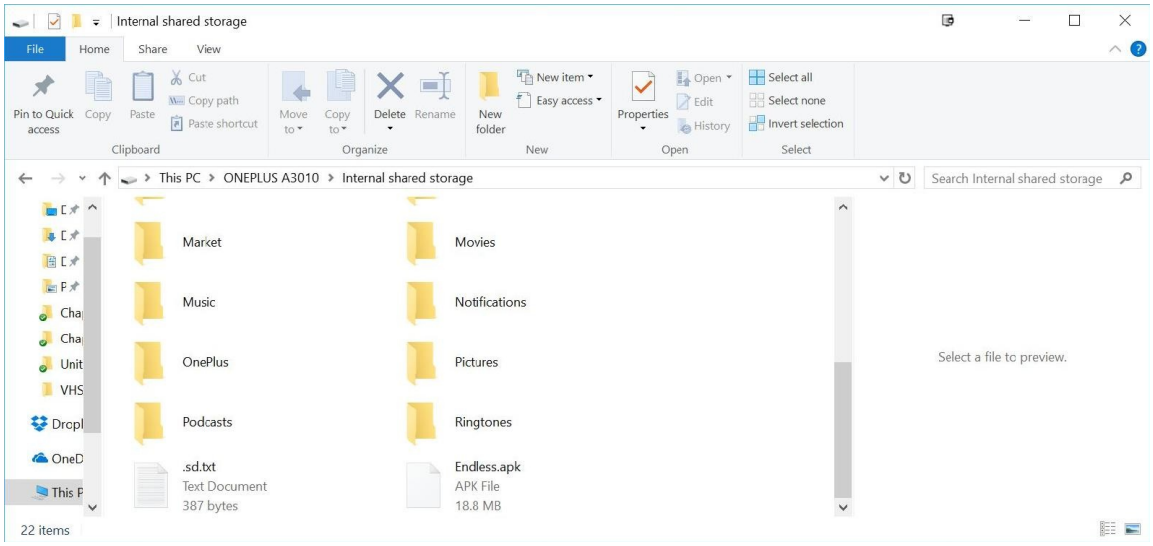
4. Connect your phone to your computer via USB. Upon being connected, your phone will show a notification saying that it's connected via USB for charging. Click on that notification and change the option to Transfer files:



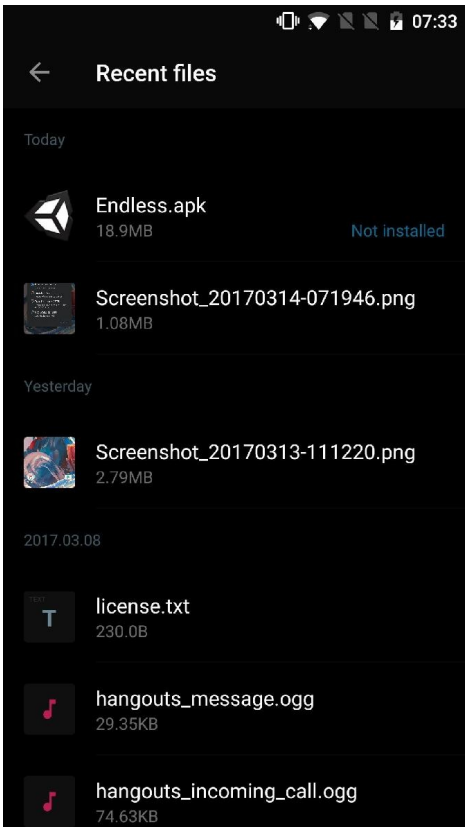
5. After that, go back to your computer and go into Windows Explorer | Finder and then to the Devices and Drives section; you should see your device appear there:



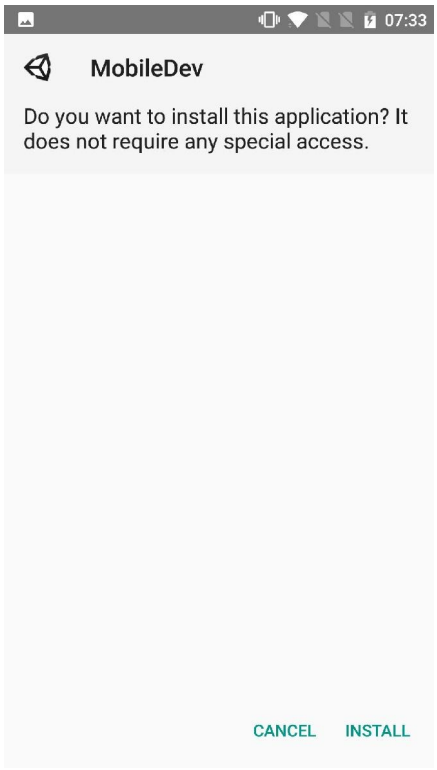
6. Double-click on your device and access the internal shared storage section from there. Then, drag the `.apk` file we made before into this folder:



7. Now, back in your phone, open the File Explorer app and go to the Recent files section. From there, click on the app icon:

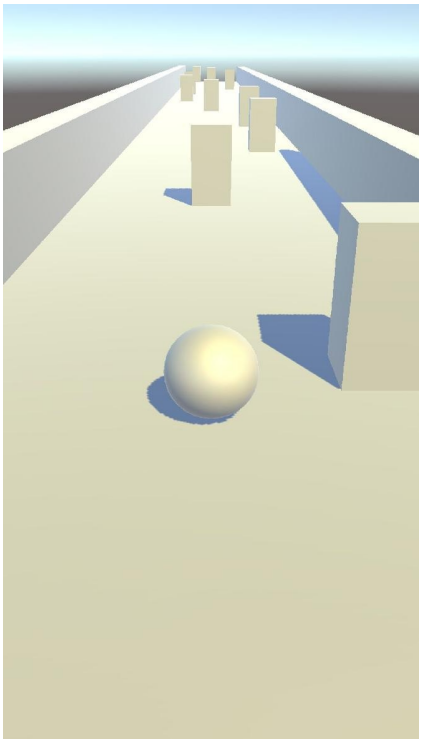


8. This will open up the installer go ahead and click on the Install button and wait for it to finish:



Of course, I can't note the steps to get your phone on all devices as some have different drivers that are required or additional steps that need to be performed in order to open files on your device. If these steps do not work and you do not know how to get files onto your device and access them and add new ones to them, go ahead and search for `phone name file transfer`*, replacing the phone name with your phone's name.*

9. Once it's finished, go ahead and click on the Open button to open our game:



As you can see, the game is on there and it's working. Granted, you can't control it yet, and there's a lot of new things that you can do, but this lets you know that you've set up our Android device properly.

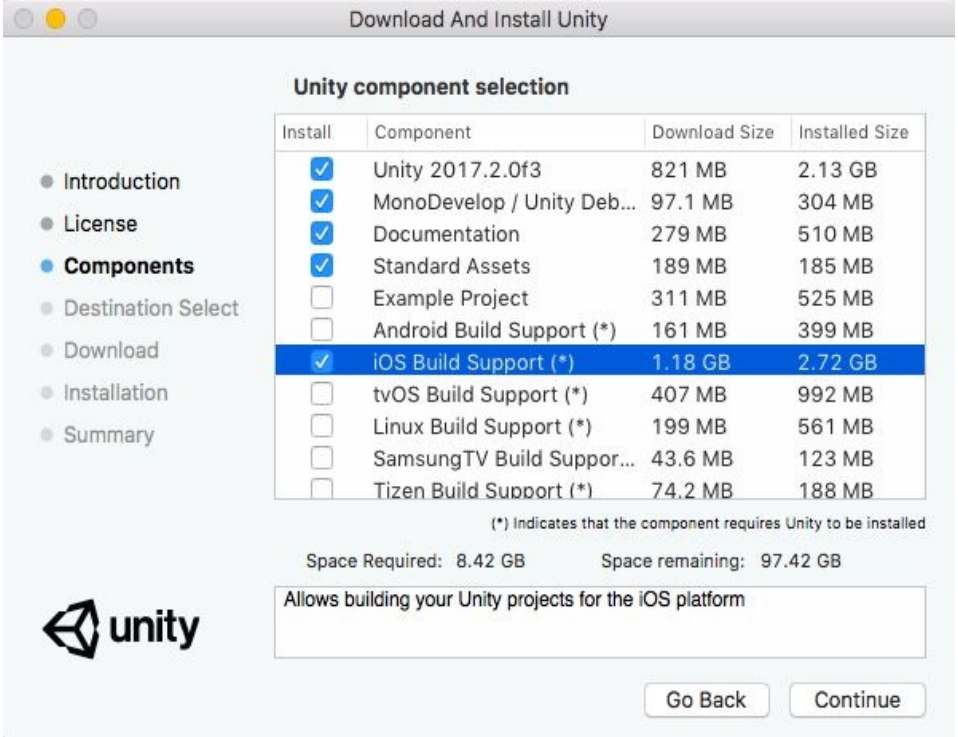
Unity for iOS setup and Xcode installation

Now that you have your game on an Android device, you now need to get it working on iOS. With Android, there's a lot of setup, but building and getting it on your device is more work, whereas, with iOS, there's less work on the setup end and more involvement with getting it actually onto the device.

Previously, you had to have a paid Apple Developer license in order to get your game onto an iOS device. Although that's still required to get the game on the App Store, you are no longer required to get it for testing. Note that the free option doesn't have everything available to you, most notably IAPs (In-App Purchases) and the Game Center; however, for making sure that it works on your device, it'll work just nicely. We will go over how to adjust your project to reflect being in the Apple Developer portal in the final chapter when we go over putting our project on the App Store.

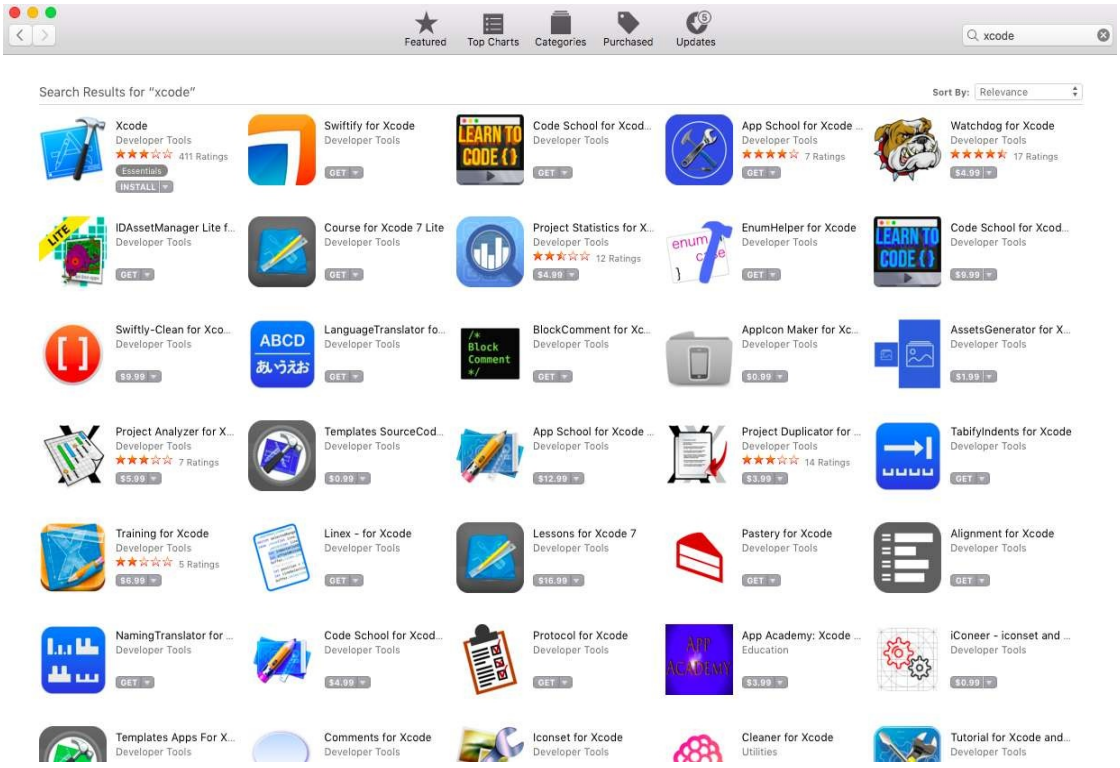
To develop for an iOS device, in addition to the device itself, you'll also need to go on a Mac computer that runs OS X 10.11 or later version. I'll be using 10.12.3 macOS Sierra. Just like working with Android, we'll also need to do some setup before we can actually do the exporting. Let's get started on that now:

1. First of all, if you haven't done so already, you'll need to add iOS Build Support (*) as an option when you are installing Unity. If you did not install it when doing the initial installation, you may reinstall Unity again with those selections checked for the platforms that you're trying to build:



This makes it so you can export your projects for iOS. Since I'll be using my Windows machine mainly, I'm only adding in iOS support, but you can do both from your Mac computer:

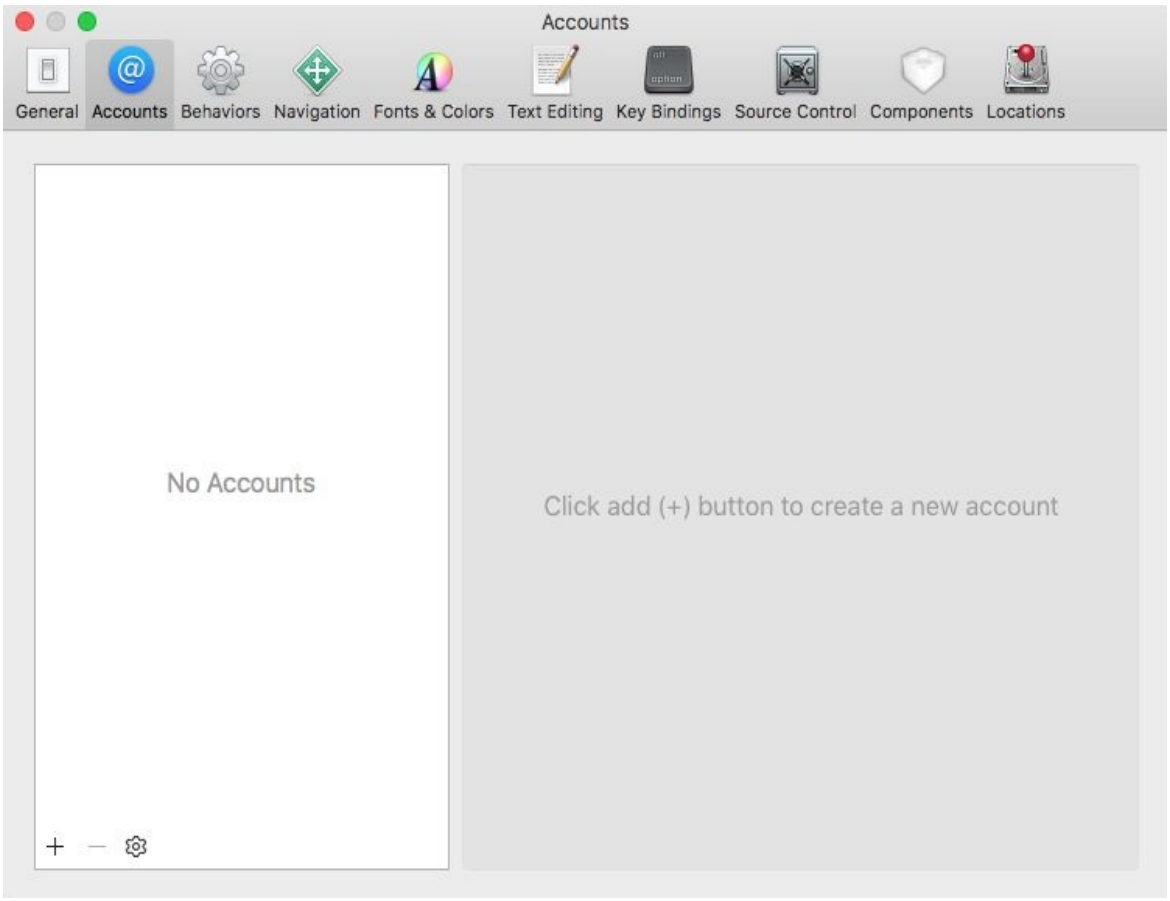
2. You'll also need to have **Xcode**, which is the program used to build iOS apps. To download it, you'll need to open up the App Store application on your computer. From the search bar in the top-right corner, type in Xcode and press *Enter*:



3. From there, you'll see the Xcode program at the top left of the page--click on it and then on the Install button. You may need to enter your Apple ID information; go ahead and do so and then wait for it to finish.

If you do not have an Apple ID, you may get one from: *http://appleid.apple.com/*.

4. Once Xcode is installed, open it up. There will be a license agreement for Xcode and the iOS SDK; go ahead and click on Agree. It'll then begin installing components that are needed for it to work.
5. You'll then be brought to a welcome screen, but we want to do some setup first. From the top menu bar, go ahead and select Xcode | Preferences (or press *command* + ,). From there, click on the Accounts button. This will display all of the Apple IDs that you want to be able to use in Xcode:



6. Click on the plus icon on the bottom left of the screen and then select Add Apple ID. From the menu that pops up, go ahead and add in your Apple ID info and you should see it appear on the screen.

If you select the name, you'll see additional info on the right side, such as what teams you are on. If you are not enrolled in the Apple Developer Program it'll just be a personal team, but if you are paying for it, you should see if there as well.

Building a project for iOS

At this point, we will be building our project for the iOS device. While there are some similarities to working with Android, there are some differences that are very important to note, so keep that in mind while reading this section.

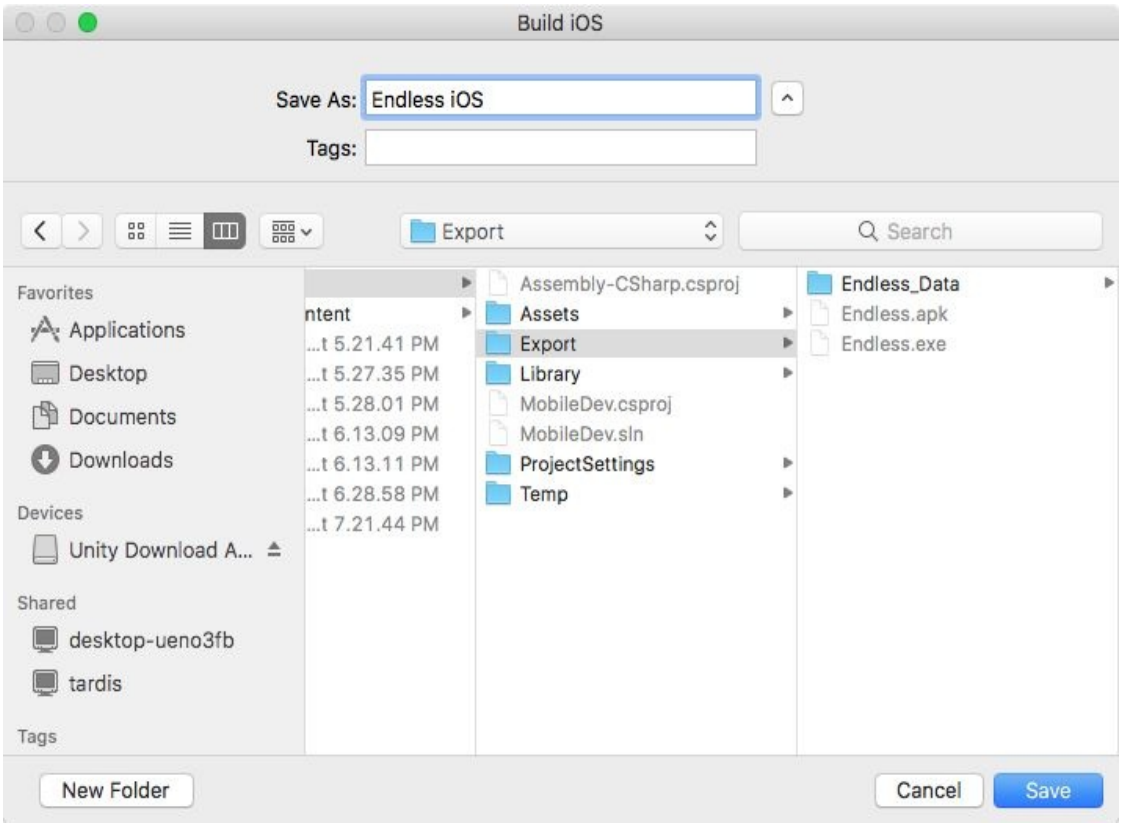
1. At this point, we will dive into Unity and then move into our Build Settings menu once again by going to File | Build Settings.
2. Click on the iOS option from the Platform list and then click on the Switch Platform button to make the change.

   > Note that this will make Unity reimport all of the assets in our game, so this may be time-consuming as you build larger and larger projects. This now also means that when we build our project, it will create an Xcode project instead of just an app, which we will need to open and work with once it's built.

3. If we didn't do so earlier when building for iOS, we must set the bundle identifier for our game at this point, which is a string that identifies the app. It's written like a website in reverse, for example, `com.yourCompanyName.yourGameName`. To modify this, we'll need to open up the Player Settings menu, which we can get to by clicking on the Player Settings... button in the Build Settings menu or by going to Edit | Project Settings | Player.
4. Open up the Other Settings section, and then put in a value that you'd like under Package Name.

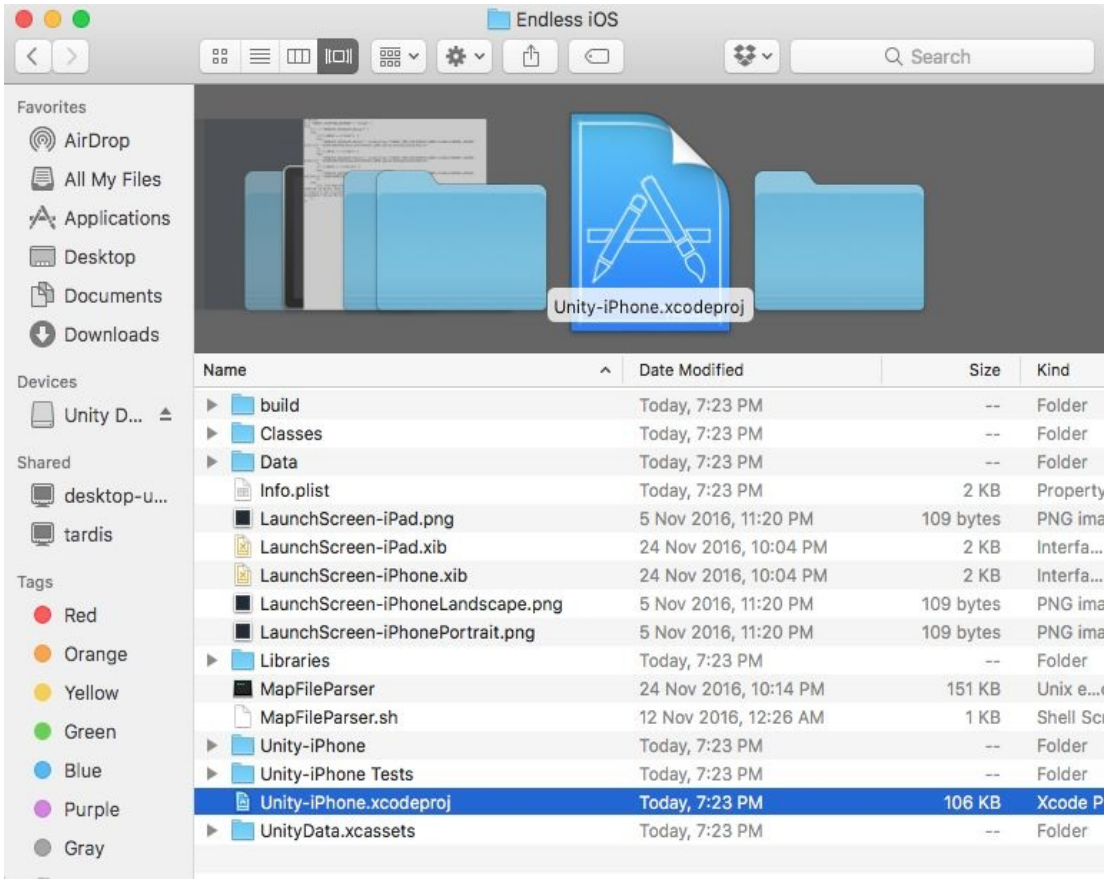
Note that if you have already changed this property when building for Android, it will already be done; there's no need to do this again.

5. Now, we can try to build the project, saving it in the same `Export` folder we created earlier--in this case, I named it `Endless iOS`--and then click on Save:

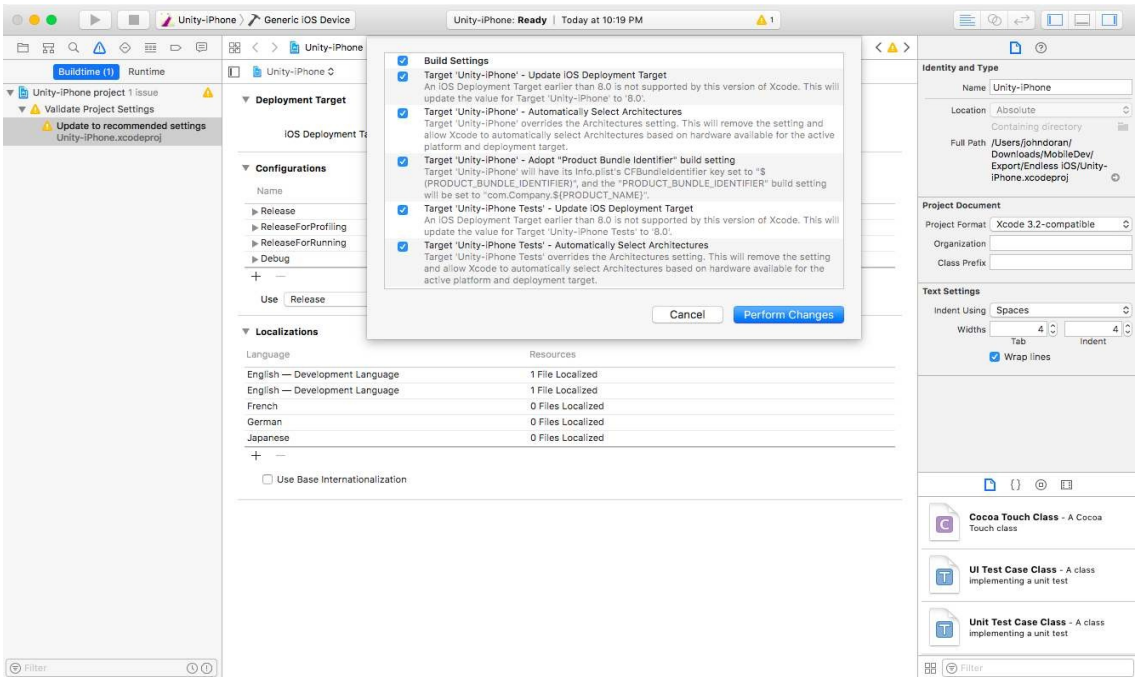


You can press the down arrow button to search for folders in the Finder window that pops up.

6. Once the project has been built, we will be taken to a Finder window at the location we created the project. From there, we can double-click on the `.xcodeproj` file to open the project inside Xcode:

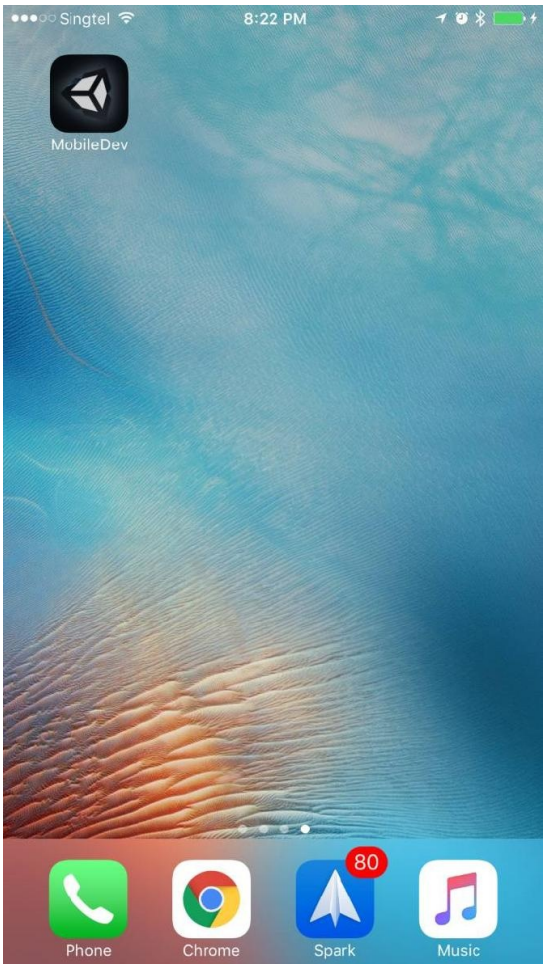


7. In Xcode, after waiting for everything to load in, you'll notice a yellow triangle with an ! in the center of it on the top center console. If you click on it, you'll see some info appear on the left-hand side. You'll see a yellow box open up with a request to make requested changes. Double-click on it and then allow it to make the automatic changes:



8. Click on the Unity-iPhone portion on the left hand side and then go down to the Signing portion of the Build Settings options, and then under Team, set it to your profile.
9. Once all the preceding steps are done, plug in your phone via USB. After loading all of the symbols it needs (wait until the top middle section says Ready); on the top right, instead of Generic iOS Device, change it to the device you've connected.
10. Once you click on the Play button, the computer will ask whether you want to enable Developer Mode; to do so, go ahead and select Enable and enter your password when it asks for it.

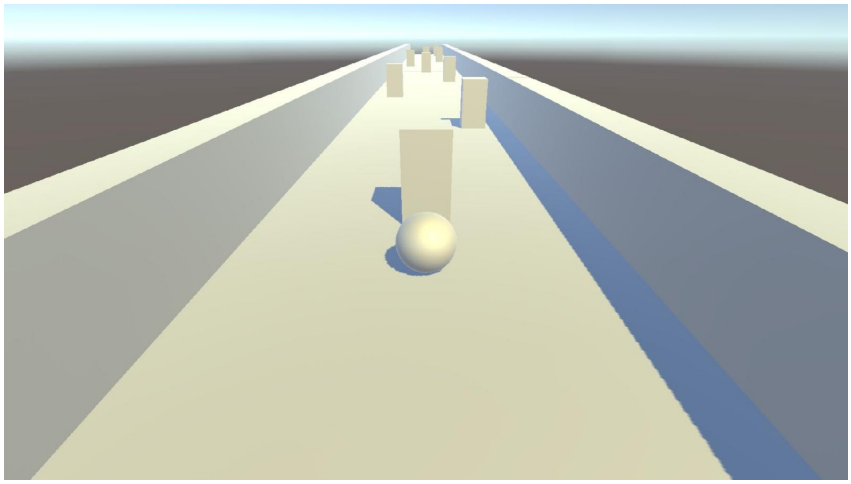
11. Your phone may be busy, so you may need to wait a bit before you're able to build to the device. You may get a window asking you to access the key access in your keychain. Go ahead and Allow it. You'll also need to unlock your phone at some point as well, so it may make the install.
12. The app will now be on your iOS device, as you can see by the following screenshot:



Right now, it has a generic Unity icon just like in Android, and we will customize it later on in this book, but for right now, we have the following issue: the game won't run.

In order to run the app, you must verify that you want the device to be able to run the app to prevent security issues. To let you know that this is an issue, Xcode will give you a warning that it couldn't launch it, so we'll need to say we want to be able to run it:

13. From your iOS Device, open up the Settings app. From there, go to General | Device Management. Then, from the menu that pops up, you'll need to say Trust "Developer Name" with Developer Name being your Apple ID account. You'll need to verify again that you wish to trust apps created by this account, so go ahead and agree, and all of the steps are done.
14. With that, exit out of settings by clicking on the home button and then go over to the location where the app was installed and tap on it to run:



With that, we have the game running on the iOS side as well.

Note that when building in the following manner without the paid license apps will only work for a limited time, possibly up to a week. If your game crashes immediately and it worked correctly beforehand this is most likely the culprit. Redeploy to the device again to check if that is the issue before modifying your actual project.

Summary

We now have our game running on both Android and iOS devices, and we have learned the steps that we'll need to take each time we want to deploy our games on these devices.

While I will not be writing about exporting to both devices again until we get to Chapter 10, *Game Build and Submission*, it's a good idea on your end to do so to see how the changes that we make will work with both devices and keep testing on each platform to make sure that your project works correctly and at a frame rate that you are okay with.

This is especially important to note as running the project on your PC via the editor or an emulator will not always work the same both ways. You may find that certain things will slow down your device and make your game choppy. You may also find something that runs fine on your mobile device will slow down your computer. The thing is, you won't know unless you are always checking it out, so I highly advise that you do so.

Now that we have our game working on a mobile device right now, it currently will not react to anything we do due to how we wrote our input code. In the next chapter, we will explore how we can add input to our project as well as the design considerations to make in regards to how the different forms of input will change our game.

Mobile Input/Touch Controls

How players interact with your project is probably one of the most important things that can be established as a part of your project. While inputting is done for projects no matter what platform you are using, this is also one area that can make or break your mobile title.

If the controls that are implemented don't fit the game that you're making, or if the controls feel clunky, players will not play your game for long stretches of time. While many people consider Rockstar's *Grand Theft Auto* series of games to be a great on console and PC, playing the game on a mobile device provides a larger barrier of entry, due to all of the buttons on the screen and replacing joysticks with virtual versions that don't have haptic feedback in the same manner as on other platforms.

Mobile and tablet games that tend to do well typically have controls that are simple, finding as many ways to streamline the gameplay as possible. Many popular games require a single input, such as Dong Nguyen's *Flappy Bird* and Ketchapp's *Ballz*.

There are many different ways for games to interact with a mobile device, which is different than for traditional games, and we will explore a number of those in this chapter.

Chapter overview

In this chapter, we will cover the different ways that inputs will work on mobile devices. We will start off with the input that is already built in to our project using mice, and then move on to touch events, gestures, using the accelerometer, and accessing information via the `Touch` class.

Our objectives

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

- Using mouse input for mobile input
- Movement via touch
- Implementing a gesture
- Using the accelerometer
- Reacting to touch

Using mouse input

Now, before we dive into mobile-only solutions, I do want to point out that it is possible to write inputs that work on both mobile and PC, namely using mouse controls. Mobile devices support using mouse clicks as taps on the screen, and we can use the mouse position as being where the screen was touched. This doesn't give you all of the features that the mobile-only options do; we will be discussing that later on in this chapter, but I think it's important to note, since I use this often for ease of testing on both the PC and on my device, making it so I don't have to deploy to a mobile device to test every single change made in the project:

1. Inside Unity, open up your `PlayerBehaviour` script and add the following highlighted code to the `Update` function:

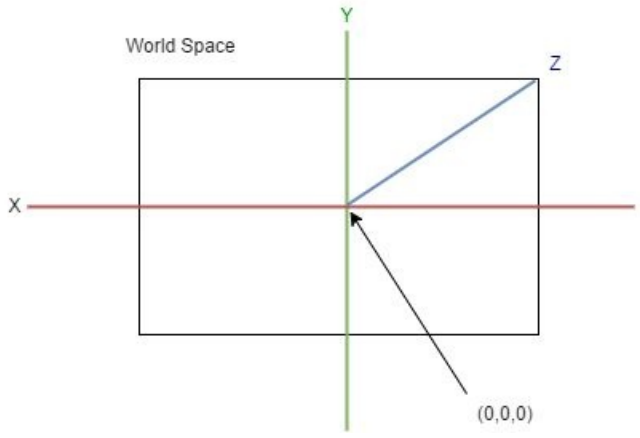
```
void Update()
    {
        // Check if we're moving to the side
        var horizontalSpeed = Input.GetAxis("Horizontal") *
                              dodgeSpeed;

        // If the mouse is held down (or the screen is tapped
        // on Mobile)
        if (Input.GetMouseButton(0))
        {
            // Converts to a 0 to 1 scale
            var worldPos =
            Camera.main.ScreenToViewportPoint(Input.mousePosition);
            float xMove = 0;

            // If we press the right side of the screen
            if (worldPos.x < 0.5f)
            {
                xMove = -1;
            }
            else
            {
                // Otherwise we're on the left
                xMove = 1;
            }

            // replace horizontalSpeed with our own value
            horizontalSpeed = xMove * dodgeSpeed;
        }

        // Apply our auto-moving and movement forces
        rb.AddForce(horizontalSpeed, 0, rollSpeed);
    }
```

We have added a number of things to the preceding script. Firstly, we check whether the mouse button had been held down or not through the use of the `Input.GetMouseButton` function. The function will return `true` if the mouse is held down, and `false` if it is not. The function takes in a parameter, which is for what mouse button we'd like to check, providing `0` for the left button, `1` for the right, and `2` for the middle button. For mobile, however, only `0` will be picked up as a click.

For more info on the `Input.GetMouseButton` function, check out https://docs.unity3d.com/ScriptReference/Input.GetMouseButton.html.

To start off, we can get the position that the mouse is at using the `Input.mousePosition` property. However, this value is given to us in screen space. What is screen space? Well, let's first talk about how we traditionally deal with positions in Unity making use of world space.

When dealing with positions in Unity through the Inspector window, we have the point **(0,0,0)** in the middle of our world, which we call the origin, and then we refer to everything else based on an offset from there. We typically refer to this method of positioning as **World Space**. Assuming that we have our camera pointing toward the origin, **World Space** looks like this:

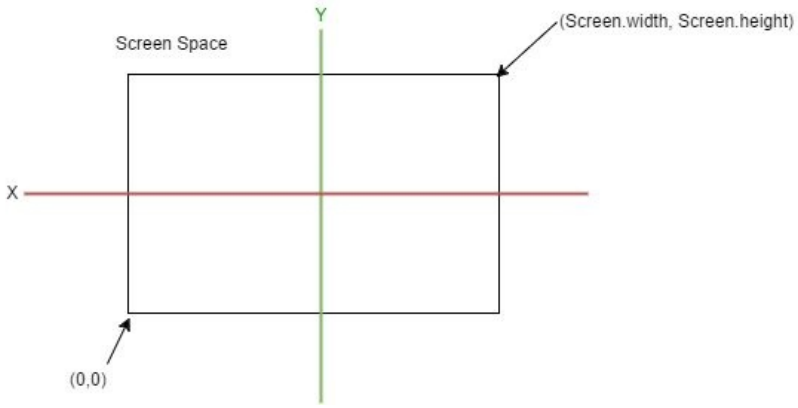


The lines are the *x*, *y*, and *z* axes of our world. If I were to move an object to the right or left, it would move along the X-axis positively

or negatively, respectively. When in school, you may have learned about using graphs and points, and world space works very much like that.

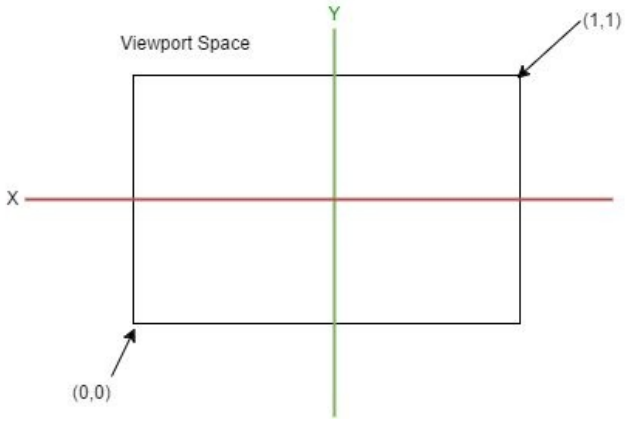
It's not important for the current conversation, but I should note that children of parented objects use a different system in the Inspector, which is that they are given positions relative to their parent instead. This system is called ***local space****.*

When using mouse input, Unity gives us this information in another space, **Screen Space**. In this space, the position is based on where the camera is and isn't involved with the actual game world. This space is also just in 2-D, so there's only an *x* and *y* position with *z* always being stuck at `0`:



In this case, the bottom left of the screen would be **(0,0)** and the top right would be **(Screen.width, Screen.height)**. `Screen.width` and `Screen.height` are values in Unity that will give us the screen size of the screen window in pixels.

We could use these values as provided and then compare what side of the screen the player pressed, but, in my case, I think it'd be better to convert the position into an easier space to work with. One such space is the **Viewport space**, which goes from (0,0) to (1,1):

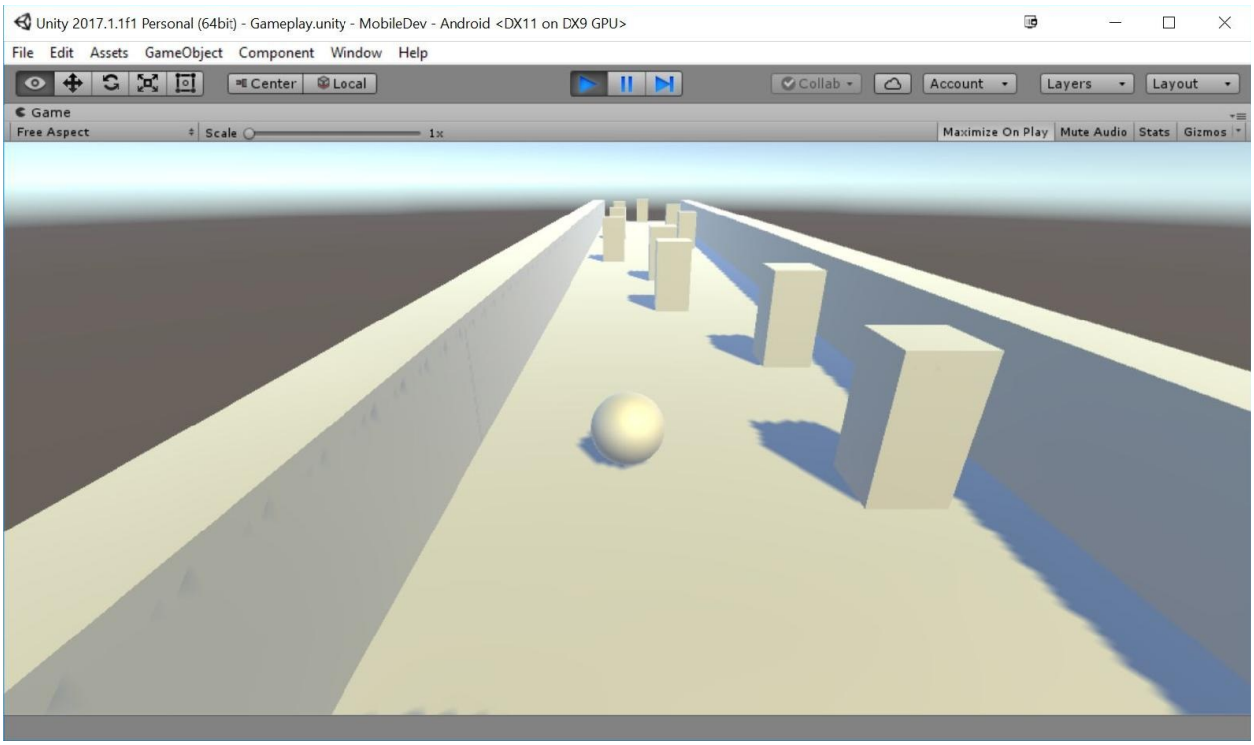


Note that some of Unity's functions will use a `Vector3` *instead of a* `Vector2` *in order to work with 3D spaces as well.*

Instead of searching whether our *x* position is less than half of the screen width, I can instead just check whether it's less than `0.5`, which is what we are doing here. If the value is less than `0.5`, it's on the left side of the screen so we return `-1`; otherwise, it's on the right side, so we give `1`.

Once we know that, we can then set the horizontal speed variable to move to the left or right based on our movement.

2. Save the script and dive back into Unity; you will see the following:



As you can see in the preceding screenshot, we can now use either the mouse (`Input.mousePosition`) or our keyboard (`Input.GetAxis`) to move our player.

Moving via touch

This works well enough for what we're doing right now, but I'm assuming that you'll want to know how to use the mobile device's own way of moving, so we will go ahead and learn how to replicate the same functionality using touch instead.

Unity's Input engine has a property called `Input.touches`, which is an array of `Touch` objects. The `Touch` struct contains information on the touch that occurred, having information such as the amount of pressure on the touch and how many times you tapped the screen. It also contains the position property--like `Input.mousePosition`--that will tell you what position the tap occurred at, in pixels.

For more info on the Touch struct, check out https://docs.unity3d.com/ScriptReference/Touch.html.

To adjust our preceding code to support touch instead of mouse inputs, our code could look something like the following:

```
//Check if Input has registered more than zero touches
if (Input.touchCount > 0)
{
   //Store the first touch detected.
   Touch myTouch = Input.touches[0];

   // Converts to a 0 to 1 scale
   var worldPos = Camera.main.ScreenToViewportPoint(myTouch.position);

   float xMove = 0;

   // If we press the right side of the screen
   if (worldPos.x < 0.5f)
   {
      xMove = -1;
   }
   else
   {
      // Otherwise we're on the left
      xMove = 1;
   }

   // replace horizontalSpeed with our own value
   horiztonalSpeed = xMove * dodgeSpeed;

}
```

Now, you may note that this code looks very similar to what we've written in the preceding section. With that in mind, instead of copying and pasting the appropriate code twice and making changes like a number of starting programmers would do, we can instead take the similarities and make a function. For the differences, we can use parameters to change the value instead; consider the following parameters:

1. Keeping that in mind, let's add the following function to the `PlayerBehaviour` class:

```
/// <summary>
/// Will figure out where to move the player horizontally
/// </summary>
/// <param name="pixelPos">The position the player has
/// touched/clicked on</param>
/// <returns>The direction to move in the x axis</returns>
float CalculateMovement(Vector3 pixelPos)
{
    // Converts to a 0 to 1 scale
    var worldPos = Camera.main.ScreenToViewportPoint(pixelPos);

    float xMove = 0;

    // If we press the right side of the screen
    if (worldPos.x < 0.5f)
    {
        xMove = -1;
    }
    else
    {
        // Otherwise we're on the left
        xMove = 1;
    }

    // replace horizontalSpeed with our own value
    return xMove * dodgeSpeed;
}
```

   Now, instead of using `Input.mousePosition` or the touch position, we instead use a parameter for the function. Also, unlike previous functions we've written, this one will actually use a return value; in this case, it will give us a floating point value. We will use this value in the `Update` to set `horiztonalSpeed` to a new value when this function is called. Now, we can call it appropriately.

2. Now, update the `Update` function, as follows:

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update ()
```

```
{
    // Movement in the x axis
    float horizontalSpeed = 0;

    //Check if we are running either in the Unity editor or in a
    //standalone build.
    #if UNITY_STANDALONE || UNITY_WEBPLAYER

    // Check if we're moving to the side
    horizontalSpeed = Input.GetAxis("Horizontal") *
                            dodgeSpeed;

    // If the mouse is held down (or the screen is tapped
    // on Mobile)
    if (Input.GetMouseButton(0))
    {
        horizontalSpeed = CalculateMovement(Input.mousePosition);
    }

    //Check if we are running on a mobile device
    #elif UNITY_IOS || UNITY_ANDROID

    //Check if Input has registered more than zero touches
    if (Input.touchCount > 0)
    {
        //Store the first touch detected.
        Touch myTouch = Input.touches[0];
        horizontalSpeed = CalculateMovement(myTouch.position);

    }

    #endif

    // Apply our auto-moving and movement forces
    rb.AddForce(horizontalSpeed, 0, rollSpeed);
}
```

In the preceding example, I am using a `#if` directive based on the platform selected. Unity will automatically create a `#define` depending on what has been selected as the platform we are deploying for. What this `#if` does, along with `#elif` and `#endif`, is allow us to include or exclude code from our project based on these symbols.

In Visual Studio, you may note that if you're building for iOS or Android, the code within the `UNITY_STANDALONE` section is grayed out, meaning that it won't be called currently. However, if we change our platform, the appropriate code will become used, depending on the platform that we would like to create for.

To take a look at all of the other platform-dependent `#defines`, check out https://docs.unity3d.com/Manual/PlatformDependentCompilation.html.

This will allow us to specify code for different versions of our project, which is

vital when dealing with multiplatform development.

In addition to Unity's built-in `#defines`*, you can create your own by going to Edit | Project Settings | Player, scrolling down to Other Settings in the Inspector window, and changing the Scripting Define Symbols. This can be great for targeting specific devices or for showing certain pieces of debug information, in addition to a number of other things.*

3. Save the script and dive back into Unity. Upon exporting your game to your Android device, you should note that the controls now work correctly using our newly created touch code.

Implementing a gesture

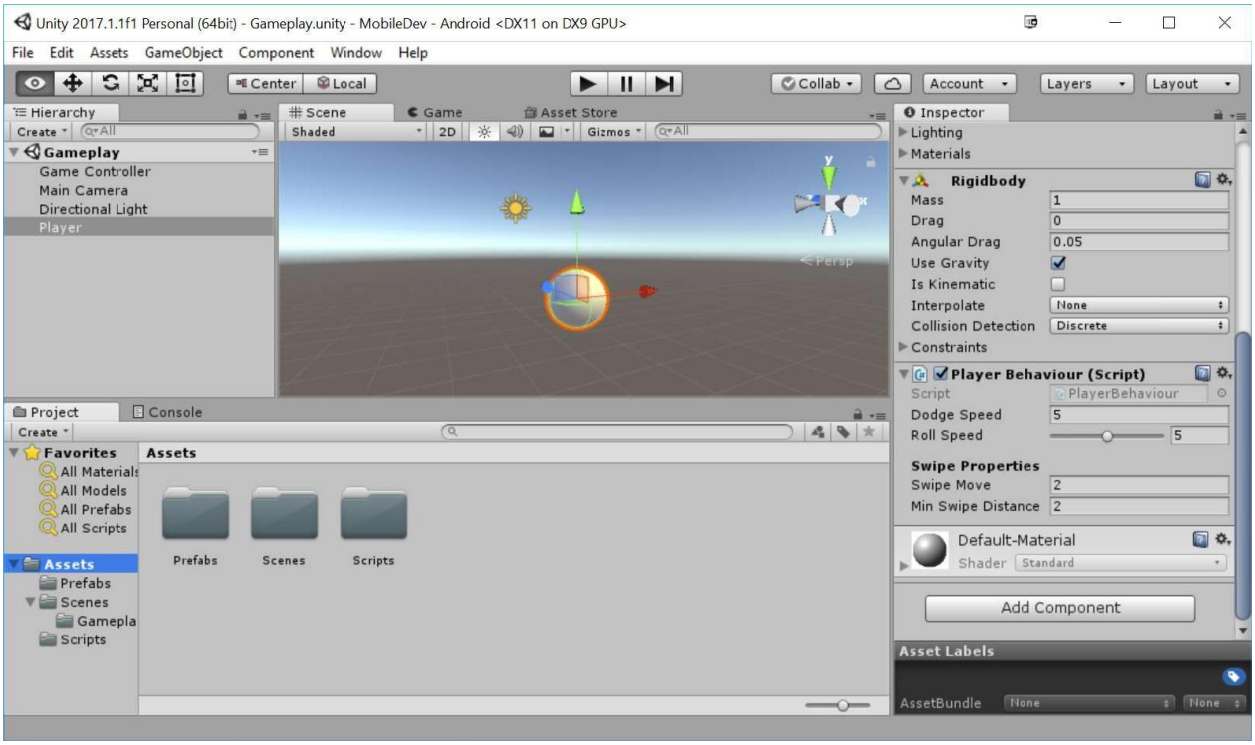
Another type of input that you'll find in mobile games is that of a swipe. This will allow us to use the general movement of the touch to dictate a direction for us to move in. This is usually used to have our players *jump* from one position to another or move quickly in a certain direction, so we'll go ahead and put that in, instead of our previous movement system:

1. First, in the `PlayerBehaviour` script, go ahead and add some new variables for us to work with:

```
[Header("Swipe Properties")]
    [Tooltip("How far will the player move upon swiping")]
    public float swipeMove = 2f;


    [Tooltip("How far must the player swipe before we will execute the action (in
pixel space)")]
    public float minSwipeDistance = 2f;

    /// <summary>
    /// Stores the starting position of mobile touch events
    /// </summary>
    private Vector2 touchStart;
```

In order to determine whether we are swiping, we will need to first check the start and the end of our movement. We will store the starting position in the `touchStart` property. We will also have the `swipeMove` property to set how far we will "jump" when the swipe happens. Lastly, we have the `minSwipeDistance` variable which will make sure that the player has moved on the x-axis a little before actually making the jump.

You'll also note that the `Header` attribute has been added to the top of the first variable. This will add a header to the Inspector tab, making it easier to break apart different sections of your script. If you were to save the script and dive into Unity, you should note that it has been added when you select the player:



2. Now, back in the `PlayerBehaviour` script, update the `Update` function to add the following highlighted code:

```
/// <summary>
/// Update is called once per frame
/// </summary>

void Update ()
{
    // Movement in the x axis
    float horizontalSpeed = 0;

    // Check if we are running either in the Unity editor
    // or in a standalone build.
    #if UNITY_STANDALONE || UNITY_WEBPLAYER || UNITY_EDITOR

    // Check if we're moving to the side
    horizontalSpeed = Input.GetAxis("Horizontal") *
                          dodgeSpeed;

   // If the mouse is held down (or the screen is tapped
   // on Mobile)
    if (Input.GetMouseButton(0))
    {
        horizontalSpeed = CalculateMovement(Input.mousePosition);
    }

    // Check if we are running on a mobile device
    #elif UNITY_IOS || UNITY_ANDROID

    // Check if Input has registered more than zero touches
    if (Input.touchCount > 0)
    {
```

```
        // Store the first touch detected.
        Touch touch = Input.touches[0];

        // Uncomment to use left and right movement
        //horizontalSpeed = CalculateMovement(touch.position);

        SwipeTeleport(touch);
    }

    #endif

    // Apply our auto-moving and movement forces
    rb.AddForce(horizontalSpeed, 0, rollSpeed);
}
```

In the preceding code, we commented out the `CalculateMovement` function and instead added a new behavior called `SwipeTeleport`. It hasn't been created yet, but this will take in the `Touch` event and use its properties to move the player if a swipe happens.

3. We will then create a function to handle this new swiping behavior, as follows:

```
/// <summary>
    /// Will teleport the player if swiped to the left or right
    /// </summary>
    /// <param name="touch">Current touch event</param>

    private void SwipeTeleport(Touch touch)
    {
        // Check if the touch just started
        if (touch.phase == TouchPhase.Began)
        {
            // If so, set touchStart
            touchStart = touch.position;
        }

        // If the touch has ended
        else if (touch.phase == TouchPhase.Ended)
        {
            // Get the position the touch ended at
            Vector2 touchEnd = touch.position;

            // Calculate the difference between the beginning and
            // end of the touch on the x axis.
            float x = touchEnd.x - touchStart.x;

            // If we are not moving far enough, don't do the teleport
            if (Mathf.Abs(x) < minSwipeDistance)
            {
                return;
            }

            Vector3 moveDirection;

            // If moved negatively in the x axis, move left
            if (x < 0)
            {
```

```
                moveDirection = Vector3.left;
            }
            else
            {
                // Otherwise we're on the right
                moveDirection = Vector3.right;
            }

            RaycastHit hit;

            // Only move if we wouldn't hit something
            if (!rb.SweepTest(moveDirection, out hit, swipeMove))
            {
                // Move the player
                rb.MovePosition(rb.position + (moveDirection *
                                swipeMove));
            }
        }
    }
```

In this function, instead of just using the current touch position, we instead store the starting position when the touch begins. When the player lifts their finger, we get the position as well. We then get the direction of that movement and then apply that movement to the ball, checking whether we'll collide with something before actually causing the movement.

4. Save your script and dive back into Unity, exporting your project onto your mobile device.

Now, whenever we swipe to the left or right, the player will move accordingly.

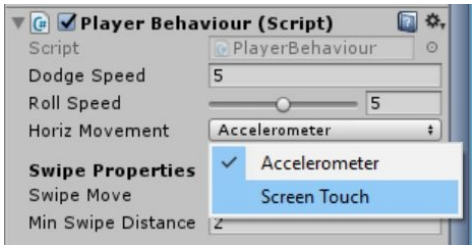
Using the accelerometer

Another type of input that mobile has, that PC doesn't, is the accelerometer. This allows you to move as the phone is tilted. The most popular example of this is likely the movement of the player in games such as Lima Sky's *Doodle Jump*. To do something similar, we can retrieve the acceleration of our device using the `Input.acceleration` property:

1. First, we may want to allow our designers to set whether they want to use this mode, or the `ScreenTouch` we used previously. With that in mind, let's create a new `enum` with the possible values to place in the `PlayerBehaviour` script above the Swipe Properties header:

```
public enum MobileHorizMovement
{
    Accelerometer,
    ScreenTouch
}

public MobileHorizMovement horizMovement = MobileHorizMovement.Accelerometer;
```

This gives us a property called `horizMovement` and the two possibilities that it can have given to us by the `enum` definition that we've made. Now, if you were to save the `PlayerBehaviour` script and dive back into the Inspector tab, you can see we can select one of these two options (Accelerometer or Screen Touch). This makes it so the game designer of the project can easily select which of the two options we'd like to use, and then we can expand to even more if we'd like to in the future:



2. Now, let's update the `Update` function with the following highlighted code:

```
//Check if we are running on a mobile device
```

```
#elif UNITY_IOS || UNITY_ANDROID

        if(horizMovement == MobileHorizMovement.Accelerometer)
        {
            // Move player based on direction of the accelerometer
            horizontalSpeed = Input.acceleration.x * dodgeSpeed;
        }

        // Check if Input has registered more than zero touches
        if (Input.touchCount > 0)
        {
            // Store the first touch detected.
            Touch touch = Input.touches[0];

            if(horizMovement == MobileHorizMovement.ScreenTouch)
            {
                horizontalSpeed = CalculateMovement(touch.position);
            }

            SwipeTeleport(touch);
        }

#endif
```

This will make use of the acceleration of our device, instead of the position or touch screen.

3. Save your script and export the project.

With that, you'll note that we can now tilt our screen to the right or left and the player will move in the appropriate direction.

In Unity, acceleration is measured in g-force values with 1 being 1g of force. If you hold the device upright (with the home button at the bottom) in front of you, the *x*-axis is positive along the right, the *y*-axis is positive upward, and the *z*-axis is positive when pointing toward you.

For more information on the accelerometer, check out: https://docs.unity3d.com/Manual/MobileInput.html.

Detecting touch on game objects

It's great to know that our regular input is working, but you may want to check whether a game object in our scene has been touched so that we can react to it. In our case, to add something else for our player to do, we'll make it so that if the player taps an obstacle, it will be destroyed.

1. In the `PlayerBehaviour` script, add the following new function:

```
/// <summary>
/// Will determine if we are touching a game object and if so
/// call events for it
/// </summary>
/// <param name="touch">Our touch event</param>
private static void TouchObjects(Touch touch)
{
    // Convert the position into a ray
    Ray touchRay = Camera.main.ScreenPointToRay(touch.position);

    RaycastHit hit;

    // Are we touching an object with a collider?
    if (Physics.Raycast(touchRay, out hit))
    {
        // Call the PlayerTouch function if it exists on a
        // component attached to this object
        hit.transform.SendMessage("PlayerTouch",
                          SendMessageOptions.DontRequireReceiver);
    }
}
```

Here, we use a different version to determine collision: a raycast. The **raycast** is basically an invisible vector leading in a given direction, and we will use it to check whether it collides with any objects inside of our scenes. This is often used in games, such as first-person shooters, to determine whether the player has hit an enemy or not without spawning a projectile and moving it there.

For more information on raycasting, check out https://docs.unity3d.com/ScriptReference/Physics.Raycast.html.

If we do hit something, we call a function named `SendMessage` on the object that we collided with. This function will attempt to call a

function with the same name as the first parameter if it exists on any component on the game object. The second parameter lets us know whether we should display an error if it doesn't exist.

For more info on the `SendMessage` *function, check out https://docs.unity3d.com/ScriptReference/GameObject.SendMessage.html.*

2. Now, in the `Update` function, let's actually call the above-mentioned function:

```
/// <summary>
    /// Update is called once per frame
    /// </summary>
    void Update()
    {
        // Movement in the x axis
        float horizontalSpeed = 0;

//Check if we are running either in the Unity editor or in a standalone //build.
#if UNITY_STANDALONE || UNITY_WEBPLAYER  || UNITY_EDITOR

        // Check if we're moving to the side
        horizontalSpeed = Input.GetAxis("Horizontal") *
                          dodgeSpeed;

    // If the mouse is held down (or the screen is tapped
    // on Mobile)
    if (Input.GetMouseButton(0))
    {
        horizontalSpeed = CalculateMovement(Input.mousePosition);
    }

//Check if we are running on a mobile device
#elif UNITY_IOS || UNITY_ANDROID

        if(horizMovement == MobileHorizMovement.Accelerometer)
        {
            // Move player based on direction of the accelerometer
            horizontalSpeed = Input.acceleration.x * dodgeSpeed;
        }

        // Check if Input has registered more than zero touches
        if (Input.touchCount > 0)
        {
            // Store the first touch detected.
            Touch touch = Input.touches[0];

            if(horizMovement == MobileHorizMovement.ScreenTouch)
            {
                horizontalSpeed = CalculateMovement(touch.position);
            }

            SwipeTeleport(touch);

            TouchObjects(touch);
        }

        #endif
```

```
        // Apply our auto-moving and movement forces
        rb.AddForce(horizontalSpeed, 0, rollSpeed);
    }
```

3. Finally, we call a `PlayerTouch` function if it exists. So, let's open up the `ObstacleBehaviour` script and add the following code:

```
public GameObject explosion;

/// <summary>
/// If the object is tapped, we spawn an explosion and
/// destroy this object
/// </summary>
void PlayerTouch()
{
    if (explosion != null)
    {
        var particles = Instantiate(explosion, transform.position,
                                    Quaternion.identity);
        Destroy(particles, 1.0f);
    }

    Destroy(this.gameObject);
}
```

This function will basically destroy the game object it is attached to and create an explosion that will also destroy itself after `1` second.

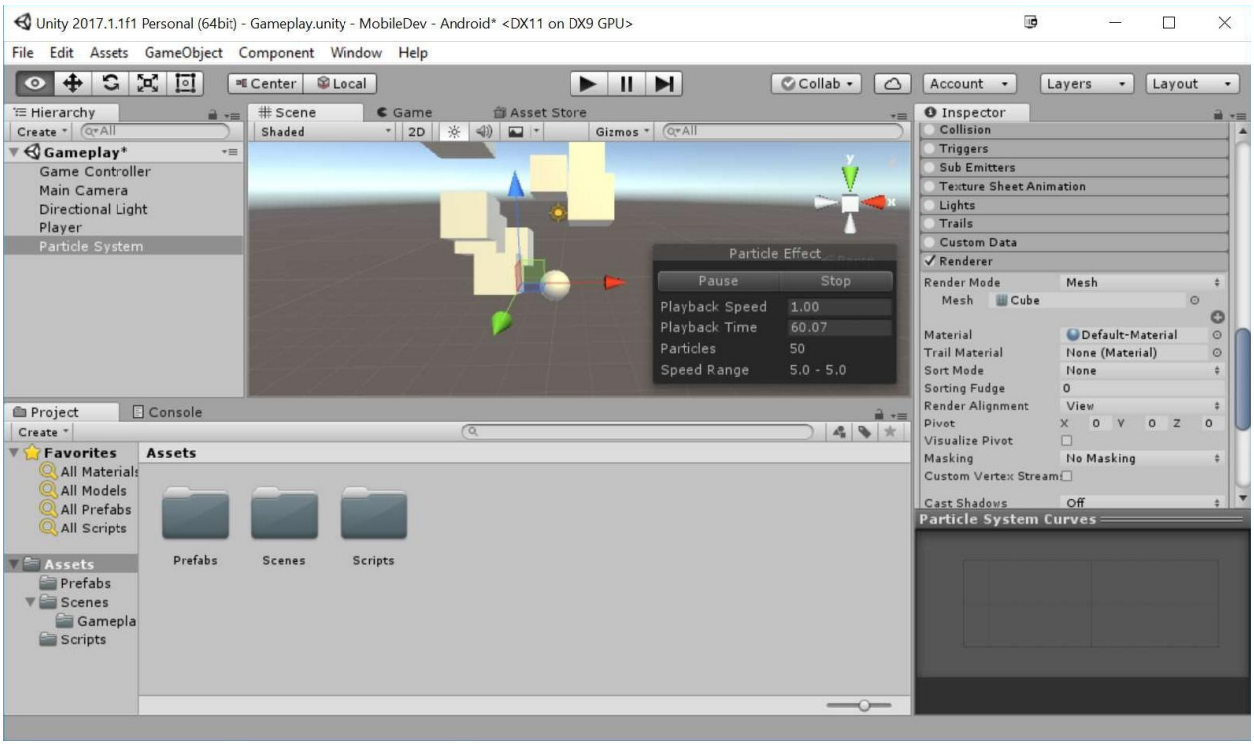
It is possible to get similar results to what we are writing by making use of Unity's `OnMouseDown` function. As we have already discussed, it is possible to use mouse events when developing for mobile. Keep in mind, though, that the use of that function is more computationally expensive than the method I'm suggesting here. This is because when you tap the screen, every object that has an `OnMouseDown` method will do a raycast to check whether it was touched. When you have many objects on the screen, you'll note massive performance differences between one raycast and one hundred, and it's important to keep performance in mind when dealing with mobile development. For more information on this, check out: http://answers.unity3d.com/questions/1064394/onmousedown-and-mobile.html.

4. Save the scripts and dive back into Unity.

We haven't created an explosion particle effect yet, so let's go ahead and do that. To create this effect, we will make use of a **particle**

system. We'll be diving into particle systems at a much deeper level in the Chapter 9, *Making Our Game Juicy*, but, for now, we can consider a particle system as a game object that is made as simply as possible so that we can spawn many of them on the screen at once without causing the game to slow down too much. This is mostly used for things such as smoke or fire, but, in this case, we will have our obstacle explode.

5. Create a particle system by going to GameObject | Effects | Particle System.
6. Select the game object in the Hierarchy window and then open the Particle System component in the Inspector tab. In there, click on the Renderer section and change RenderMode to Mesh and Material to Default-Material by clicking on the circle next to the name and selecting it from the menu that pops up:



This will make the particles look like the obstacles that we've already created as a box with the default material.

7. Next, under the top Particle System section, change the Gravity Modifier property to `1`.

   This makes it so the objects will fall over time, much like normal objects with rigid bodies do, but with less computation.

8. Then, under Start Speed, move to the right side and click on the downward-facing arrow, and from that menu, select Random Between Two Constants. From there, set the two values to `0` and `8`.

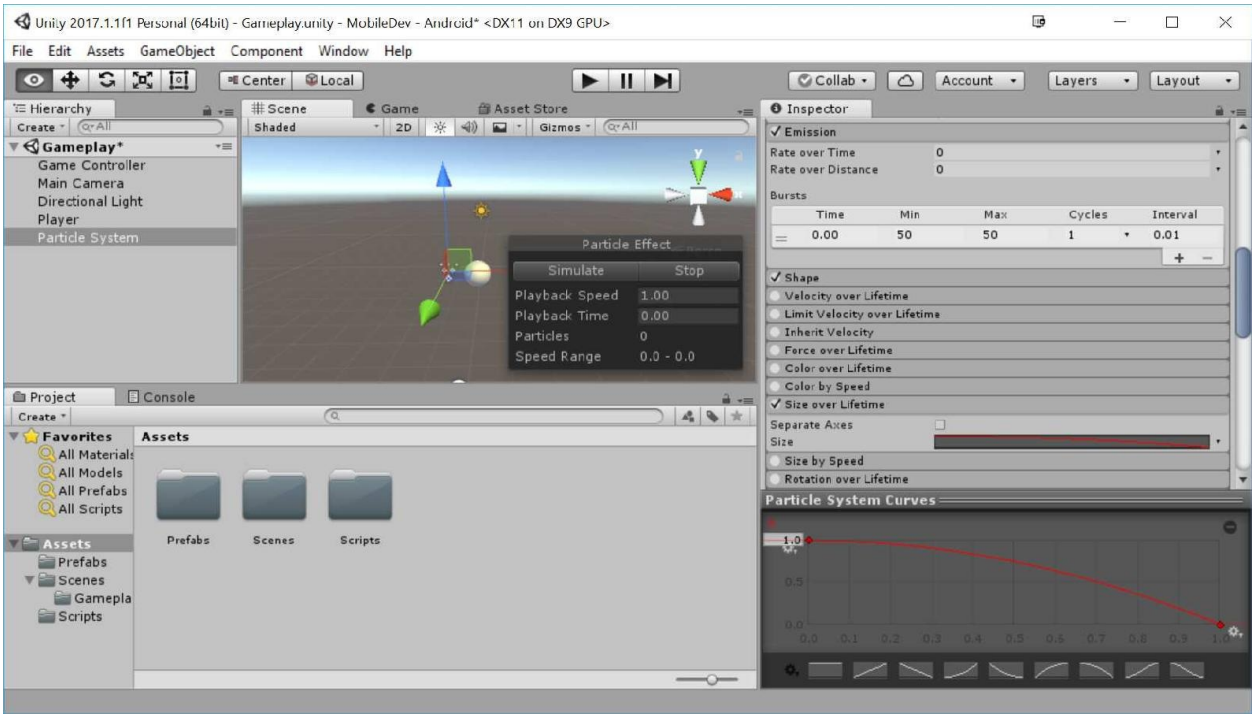
   This makes the objects spawned have starting speeds between `0` and `8`.

9. Then, change the Start Size to something between `0` and `0.25`.
10. Afterward, change Duration to `1` and uncheck the Looping option. This makes it so that the particle system will last only for 1 second, and unchecking looping means that the particle system will happen only once.

11. Finally, change the Start Lifetime property to `1` so that we can ensure that all

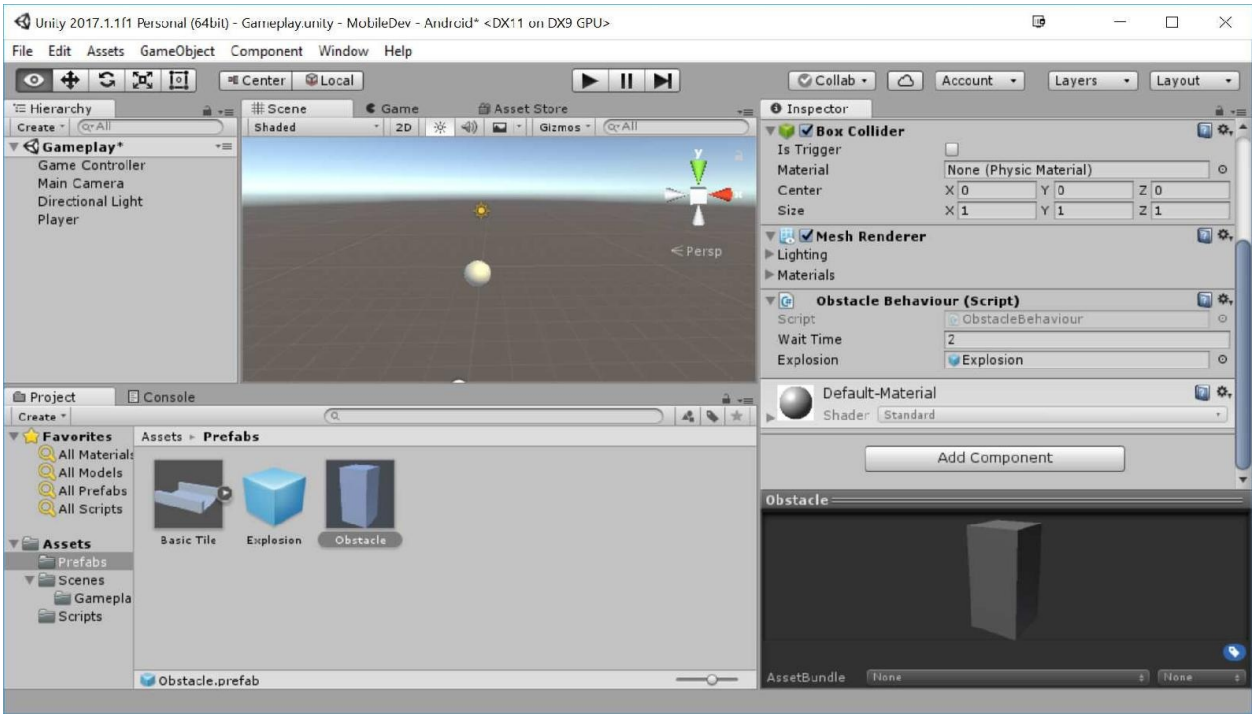
of the particles will be dead before the game object is destroyed.

12. Under the Emission section, change Rate over Time to `0`. Then, under Bursts, click on the + button and then set Min and Max to `50`.
13. Then, check Size over Lifetime and click on the text next to the check-mark to show more details. From there, change the Size property by selecting a curve that decreases over time so that at the end they'll all be `0`:

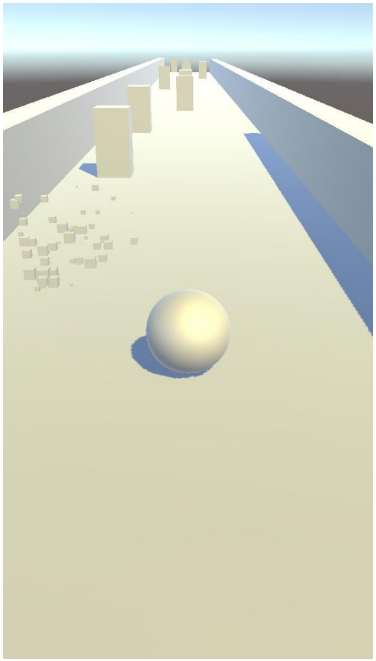


This will make the particles smaller over time, and they will destroy themselves only after they become invisible.

14. Finally, check the Collision property and open it, setting the Type property to World. This will cause the particles to hit the ground.
15. Change the name of the particle system to Explosion. Then, make your object a prefab by dragging and dropping it from the Hierarchy tab into the Project tab in the `Assets/Prefabs` folder. Once the prefab is created, remove the original from the scene by selecting it and pressing the *Delete* key.
16. Afterward, assign it to the Explosion property of the Obstacle Behaviour (Script) in the `Obstacle` prefab:



17. Save your project and export it to your mobile device:



From now on, whenever we click on the obstacles, they will be destroyed.

Summary

With that, we've covered the main ways in which games are controlled when working on mobile devices. In this chapter, we have touched on how we can use mouse inputs, touch events, gestures, and the accelerometer to allow players to interact with our game.

In the next chapter, we will explore the other main way that players interact with a game by diving into the world of user interfaces and creating menus that can be enjoyed, no matter what device the person is playing a game on.

Resolution Independent UI

When working on mobile devices, one of the things that you'll need to spend more time on is the user interface, or UI for short. Unlike when developing projects for PC, where you need to only care about a single resolution or aspect ratio, you should keep in mind that there are many different devices out there when building for mobile. For instance, we have phones, which fit in our pocket, but also tablets, which are huge. Not only that, but games can also be played horizontally or vertically.

A **Graphical User Interface** (**GUI**) is the way that players interact with your games. You've actually been using a GUI in all of the previous chapters (the Unity Editor) and also when interacting with your operating system. Without a GUI of some sort, the only way you'd be able to interact with a computer is a command prompt, such as DOS or 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 main job is programming and/or designing user interfaces, and there are college degrees on the subject as well. So, while we won't talk about everything that we have to work with using GUIs, I do want to touch on the aspects that should be quite helpful when working on your own projects in the future.

When building for mobile, it's very important that you design your UI to be resolution independent, or rather that the UI will scale and/or adjust itself to fit any screen size that is given to it. This will not only help us now, but also in the future.

The chapter overview

In this chapter, we will build the user interface for our game, starting with a title screen, and then build the other menus that we will want to use for our future chapters.

Our objectives

This chapter will be split into a number of topics. It will contain a simple step-by-step process from beginning to end. The following is the outline of our tasks:

- Creating a title screen
- Adding UI elements to the screen
- Implementing a pause menu
- Restarting the game

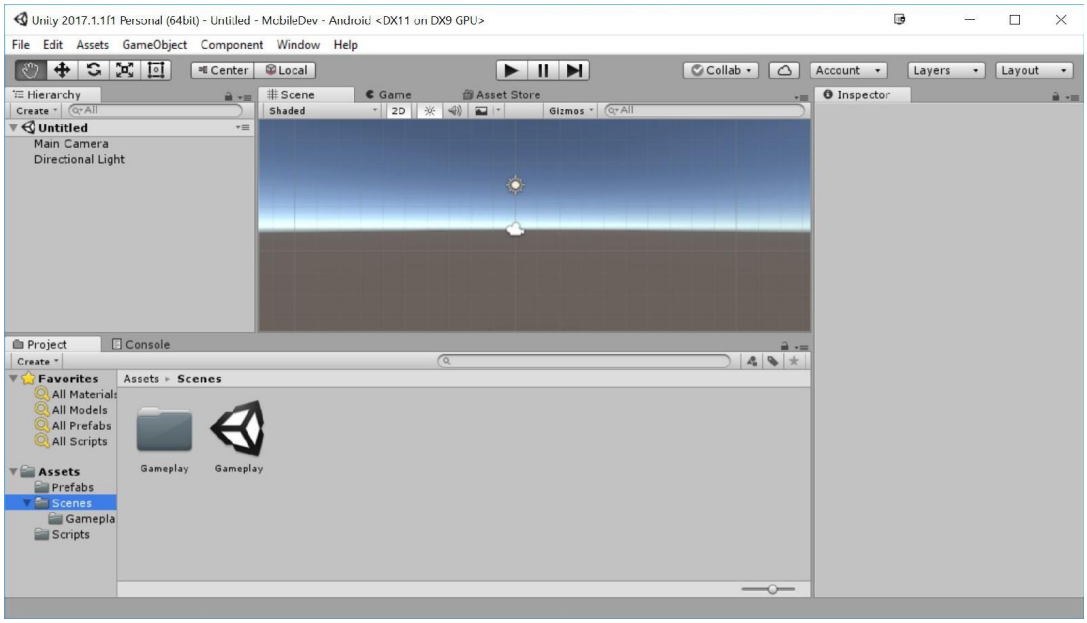
Creating a title screen

Now, before we start adding UI elements to our game, let's first set up some ground work and foundational knowledge by creating something that we will need anyways, a title screen:

1. To start, let's go ahead and create a new scene for us to work with by going to File | New Scene.

   When dealing with a UI, we often will want to see a visual representation of what will be drawn on the screen, so we will want to make use of 2D mode to have a better representation of what our UI will look like in the final version of the game.

2. To do that, go to the Scene view tab—you'll see the control bar menu with a 2D button on it underneath that. Click on it, and you should see the camera automatically move into something that looks similar to the following screenshot:

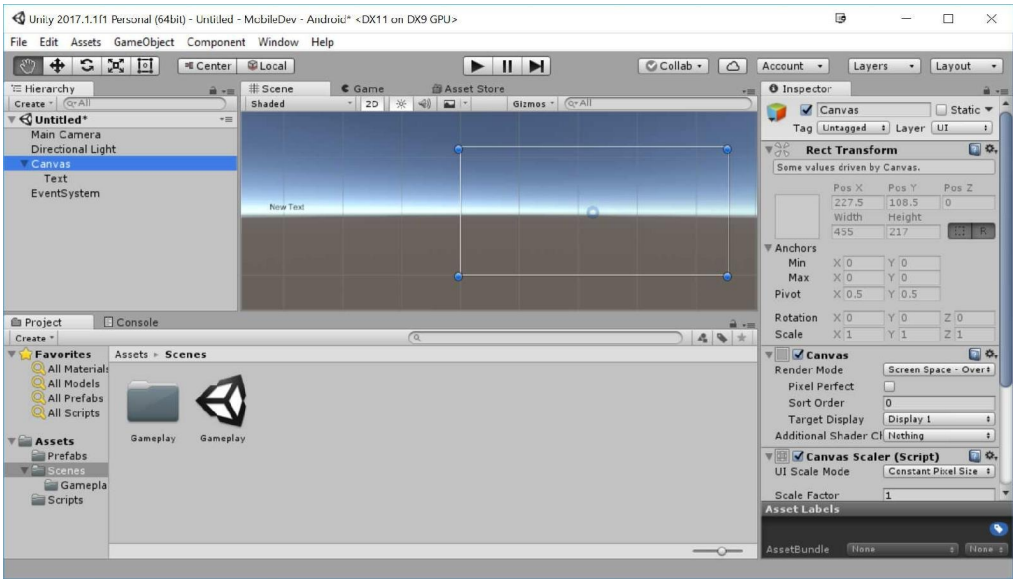


   The 2D button switches the camera between a 2D and 3D view. In 2D mode, you'll note that the Scene Gizmos is gone due to the fact that

the only option is to look perpendicularly at the XY plane (the *x* axis pointing to the right and the *y* size pointing upward) and that our camera has changed to an orthographic view.

3. We will first create a text object with the name of our game. So, with that in mind, let's go to the menu and select GameObject | UI | Text.

4. This will create three new objects, as you can see in the Hierarchy view:
    - Canvas: This is the area where all of the UI elements will reside, and if you try to create a UI element without one existing, Unity will create one for you like it just did here. From the Scene view tab, it will draw a white rectangle around itself to show you how large it is and will resize itself depending on how large the Game view is:



The game object contains a Canvas component, which allows you to dictate how the image will be rendered (and a Canvas Scaler to make your art scale, depending on the resolution of the device the game is running on) and the Graphic Raycaster, which determines whether any objects on the Canvas has been hit. We will dive into these later on, but for now, we'll leave it be.

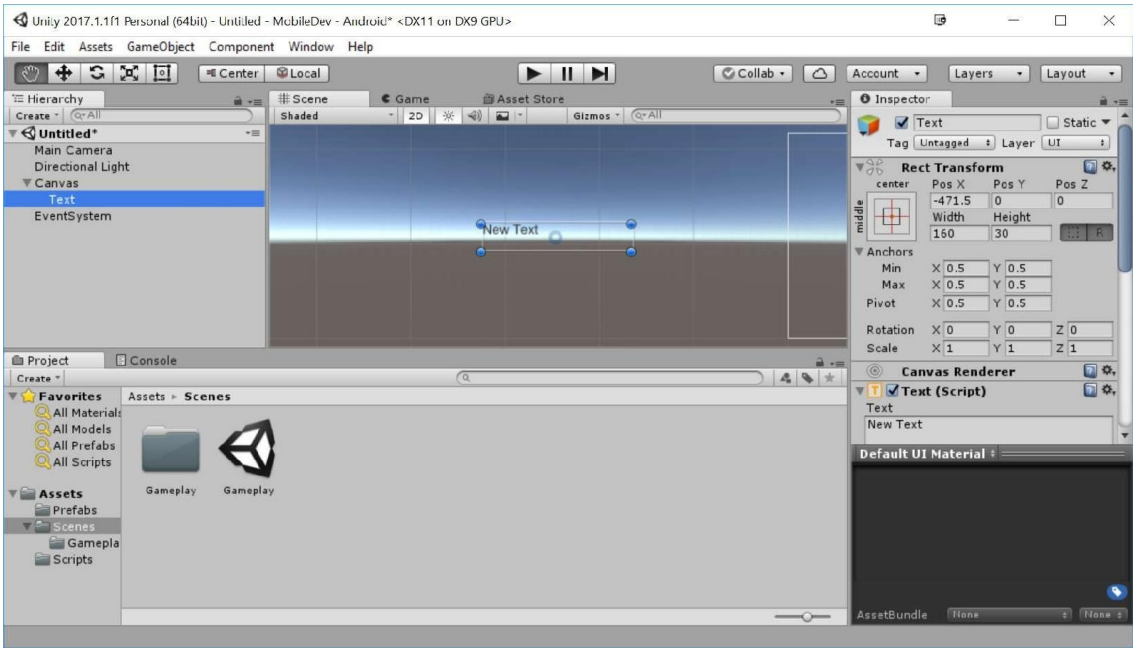
For more information on the Canvas object, check out: http://docs.unity3d.com/Manual/UICanvas.html.

- Text: This object is our actual text object, which has all of the properties to allow us to position the object anywhere on the `Canvas` object and change the text, color, size, and so on that will be displayed.

- EventSystem: This object allows users to send events to objects in our game based on various input types from keyboard to touch events to gamepads. There are properties in this object that allow you to specify how you'd like your users to interface with your UI, and if you try to create a UI element without one existing, Unity will create one for you like it did here.

For more information on the `EventSystem` object, check out: http://docs.unity3d.com/Manual/EventSystem.html.

5. By default, you may not see where our textbox was created, so we can go to the Hierarchy window and then double-click on the Text object. If all went well, we should have something like this:



6. The next thing we will do is make it easier to tell what this object is. So, with that in mind, scroll all the way up on the Inspector tab with the `Text` object selected and change its name to `Title Text`.

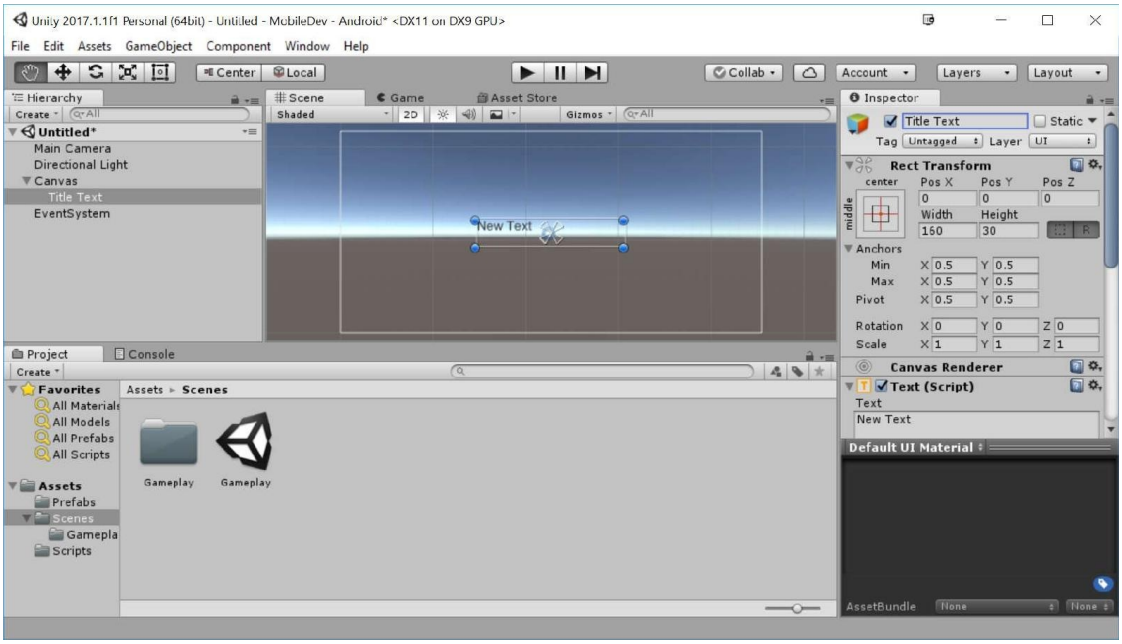
Currently, this object is off the screen due to it not being within the white box created for the Canvas. Instead, we would rather have it positioned inside the screen. After this, we will want to move this object to the proper place in the Scene; however, just like in the preceding chapter, you'll note that instead of the default Transform component, we've been seeing our text object with a Rect Transform component in the same place.

The Rect Transform component

The **Rect Transform** component is probably the most different thing about working in the UI system, so it's a good idea to learn as much as we can about it. The Rect Transform is different than the regular Transform in that while while the Transform component represents a single point, or the center of an object, the Rect Transform represents a rectangle, in which the UI element will reside in. If an object with a Rect Transform has a parent, which also has a Rect Transform, the child will specify how the object should be positioned relative to its parent.

For more information on positioning objects and information on the Rect Transform, check out: *http://docs.unity3d.com/Manual/UIBasicLayout.html*.

1. Now, to get a better idea of what the properties of the Rect Transform component mean, change the Pos X and Pos Y values to `0`, which will center our object around the object's anchors; you can then double-click on the object in the Hierarchy tab to center the camera at its new position:



2. Our object's anchors are visible from the Scene tab via four small rectangles, creating an X shape in the center of our Scene tab, if you have

the Title Text object selected (double-click on it to center the object on the screen).

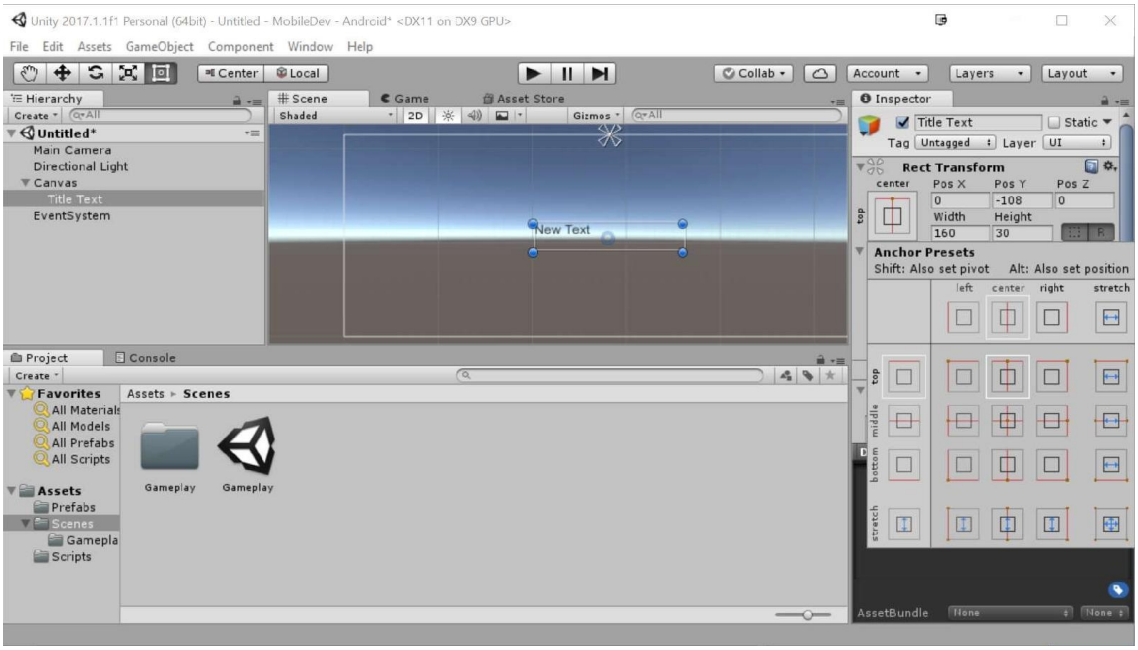
Like I mentioned previously, note that the white box that is displayed above for the Canvas may look different on your screen based on the Aspect Ratio you've set from the Game tab view (mine is set to Free Aspect, so it scales based on that to fill the space). If you go to the Game tab, you can select them from the drop-down menu on the left-hand side.

Anchors

Found inside the Rect Transform component, anchors give you the ability to *hold on* to a corner or part of the canvas so that if the canvas were to move and/or change, the pieces of the UI will stay in an appropriate place. These specify the edges of your element using a percentage of your parent's size. For example, if we set the Min X property to `0`, the UI element would stick to the left edge of its parent.

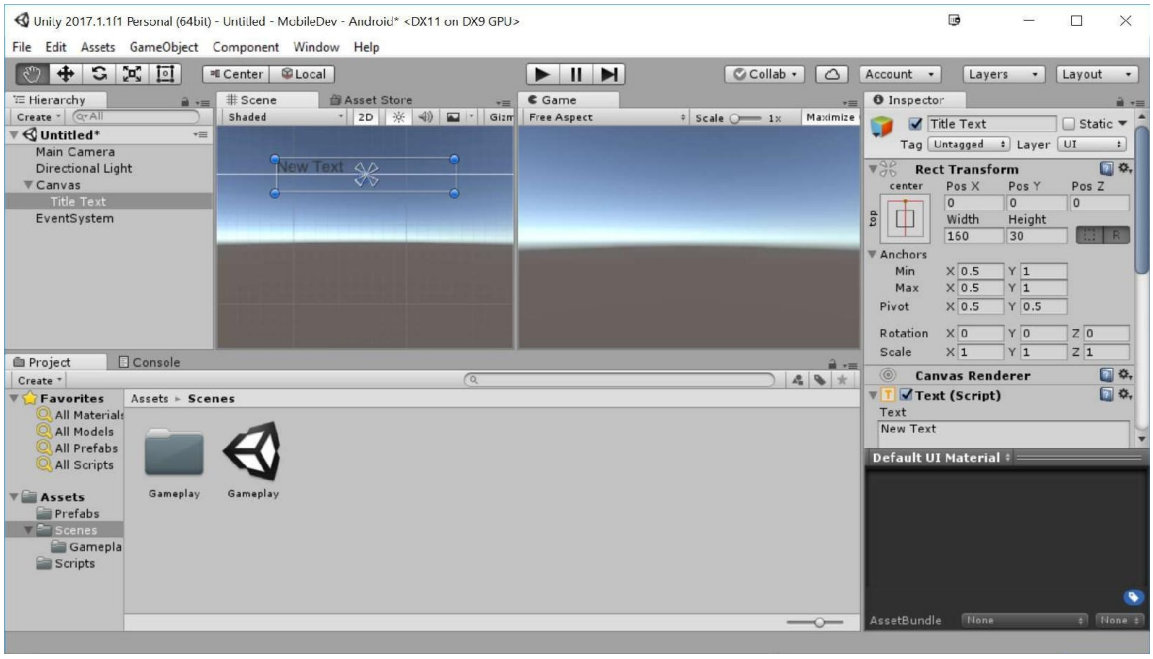
The properties above the anchors are your position relative to the anchor that has been set. This can be quite useful when it comes to things such as supporting multiple resolutions without scaling the art assets created. In our case, we will want to have our title position itself relative to the top of the camera:

1. Click on the Anchor Presets menu in the upper-left corner of the Rect Transform component (the box to the left of the Pos X and Width values). From there, it shows some of the most common anchor positions used in games for easy selection. In our case, we will want to pick the top-center option:



2. Note that after selecting it, the Pos Y value changes to another number (in

my case, `-108`). This is saying that our object is positioned `108` units below the anchor's y position (in screen space, 1 unit is 1 pixel). If we change the Pos Y value to `0`, the object would be centered along the *y* axis' anchor, which would have the object be placed with half of it off the screen, which is not good, as you can see in the following example:



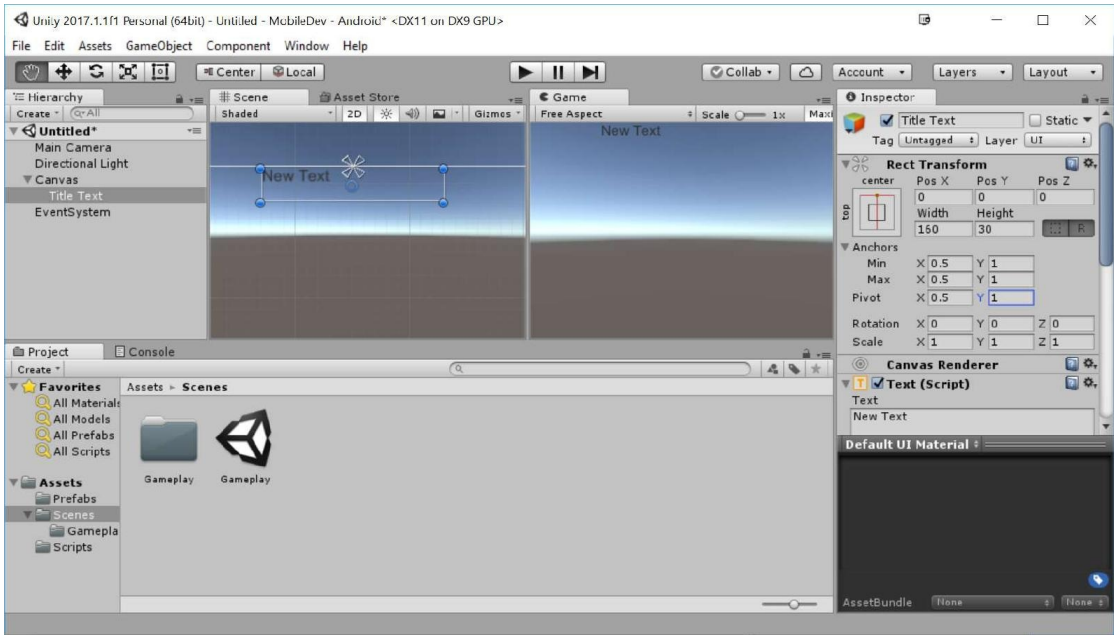
I placed the Game tab next to the Scene tab to make it easier to see the issue; you can do this by dragging and dropping the Game tab to the edge of the screen.

To reset any layout changes, you may go to the Layout menu in the top right of the screen and select Default.

However, if we changed our `Title Text` object's Pos Y to `-15` (subtracting half its Height value), it would be positioned correctly. However, hardcoding this value will be an issue if we decided we want to change the Height later on, as we would have to remember to adjust this again. It would be a lot nicer if we had something to make Pos Y at `0` the edge of the map relative to our height, and, thankfully, we have the Pivot property to fix that.

3. Next, change the Pivot Y property to `1` and then change Pos Y to `0` if you

changed it previously and it doesn't change automatically:

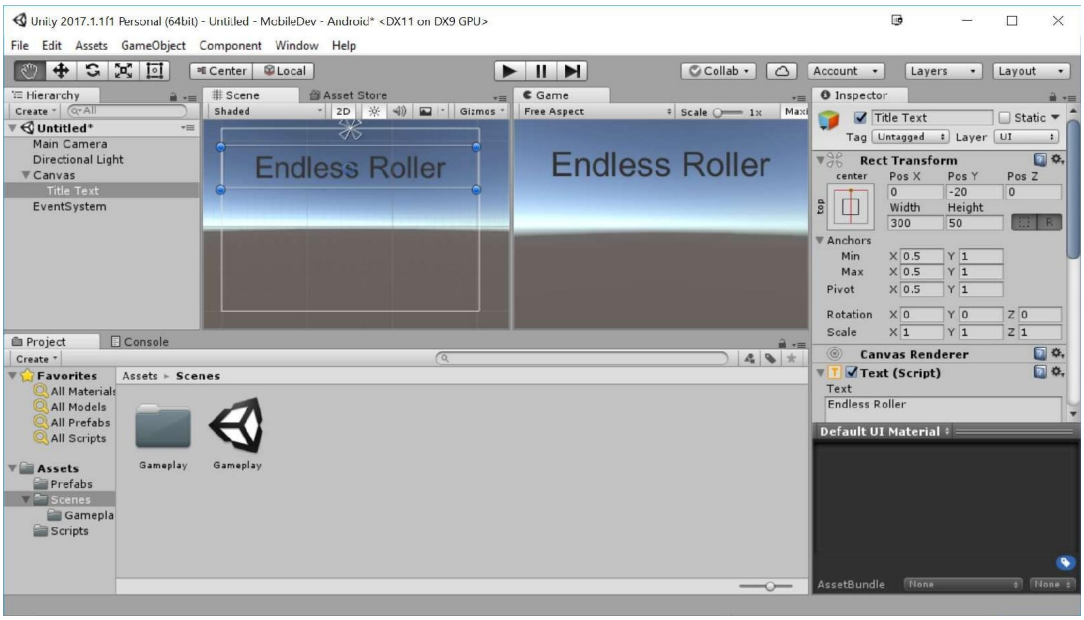


Pivots

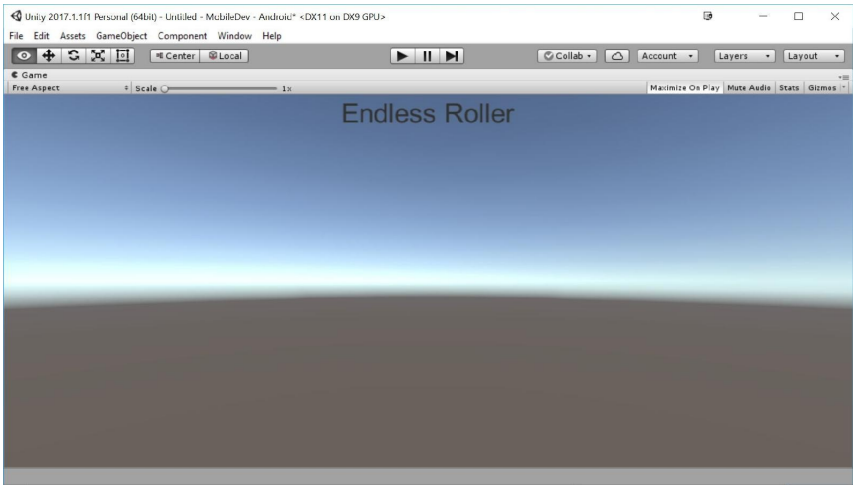
Pivots are markers that note where we want things to be in relation to our object. This means that objects will be moved, rotated, and/or scaled via this position. To note how this changes the way things react, try changing the Rotation Z property with a Pivot Y of `0`, `0.5`, and `1`, and note the differences in how things are rotated.

Note that it is possible to set the Pivot, Position, and Anchors of an object via the Anchors Preset menu I mentioned previously if you hold down the Alt + Shift keys while clicking it. This way, all of the steps we discussed will happen all at once, but it's a good idea to get a foundation on what everything means before jumping straight into using the shortcuts.

1. Now that we have our object positioned correctly, let's give it some visual flair. Select the `Title Text` object from the Hierarchy tab and then move over to the Inspector tab and scroll down to the Text component. From there, change the Text property to `Endless Roller` and set the Alignment property of the object to the center vertically and horizontally. Afterward, change the Font Size to `35`. Note that now the text doesn't show up because we've made the text too big for the Rect Transform that we defined.
2. With that in mind, scroll up to the Rect Transform and change the Width to `300` and Height to `50`. We will also want it to be offset from the top of the world, so let's change Pos Y to `-30` to give it a little offset:



Now, this looks great for this resolution; however, if we were to play the game at a larger resolution, it may look like this:



This can be good if you're trying to do a HUD in your game, but for the title screen, it's usually a good idea to have things be larger; so with that in mind, we will use the Canvas Scaler component to adjust how the screen will change based on the resolution we give it.

3. Select the `Canvas` object from the Hierarchy component and then change UI Scale Mode to Scale with Screen Size in the Inspector from Canvas Scaler.

   The key property here is the Reference Resolution. This is the

resolution that we want to base our menu on—if the resolution is bigger, it will scale up; if smaller, it will scale down. You will likely have a resolution in mind based on your mockups or an image file you've made; however, for reference, the following are some of the most common screen resolutions as at the time of writing this book:

Sample Apple device resolutions:

Device Name	Resolution
iPhone X	2436 x 1125
iPhone 7 Plus/8 Plus	1080 x 1920
iPhone 7/8	750 x 1334
iPhone 6 Plus/6S Plus	1080 x 1920
iPhone 6/6S	750 x 1334
iPhone 5	640 x 1136
iPod Touch	640 x 1136
iPad Pro	2048 x 2732
iPad 3/4	1536 x 2048
iPad Air 1 & 2	1536 x 2048
iPad Mini	768 x 1024
iPad Mini 2 & 3	1536 x 2048

Sample Android device resolutions:

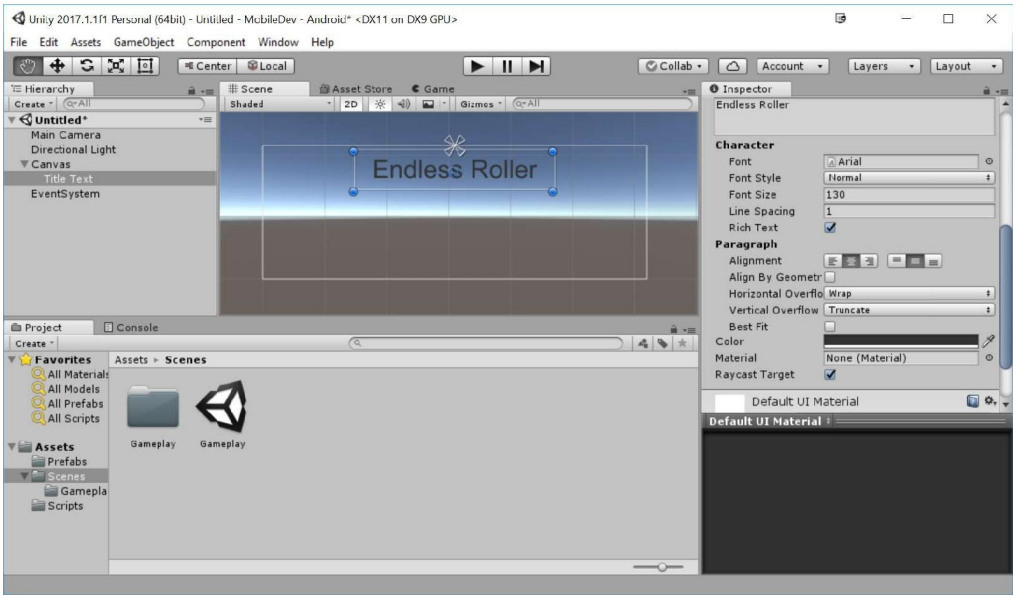
Device Name	Resolution
Samsung Galaxy S8/S8+	2960 x 1440
Google Pixel 2 XL	2560 x 1312
Nexus 6P	1440 x 2560
Nexus 5X	1080 x 1920
Google Pixel/Pixel 2	1080 x 1920
Google Pixel XL/Pixel 2 XL	1440 x 2560
Samsung Galaxy Note 5	1440 x 2560
LG G5	1440 x 2560
One Plus 3	1080 x 1920
Samsung Galaxy S7	1440 x 2560
Samsung Galaxy S7 Edge	1440 x 2560
Nexus 7 (2013)	1200 x 1920
Nexus 9	1536 x 2048
Samsung Galaxy Tab 10	800 x 1280
Chromebook Pixel	2560 x 1700

To see an up-to-date listing of cell phone screen resolutions, check out: *http://mobiphonespec.com/cellphone-screen-resolution-by-size.php*.

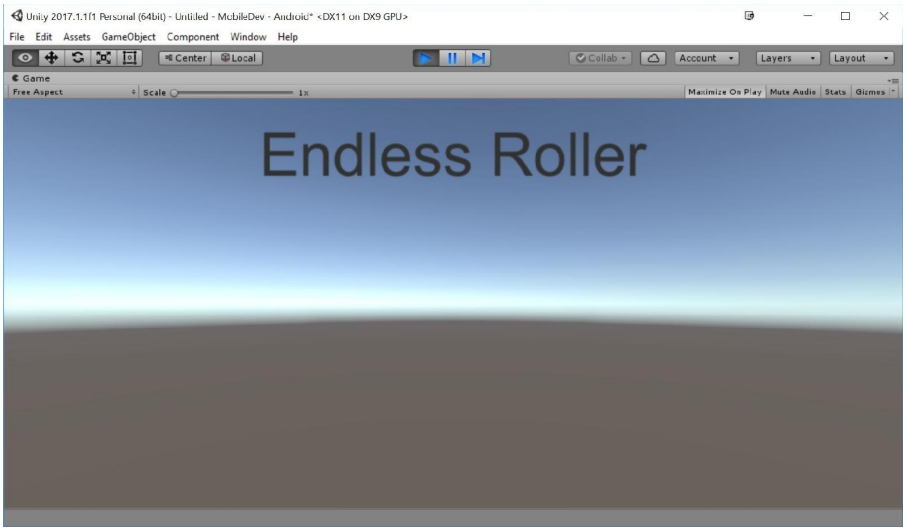
I am using a One Plus 3 and an iPhone 6S Plus, and both of them has a 1920 x 1080 resolution, so I think that would be a good place to start. However, if you are creating art assets, it's a good idea to create the UI at the largest resolution you plan on supporting and then build for other resolutions from there.

Unity has some of the most common resolutions built-in, which can be seen/changed from the drop down in the Game tab view mentioned previously.

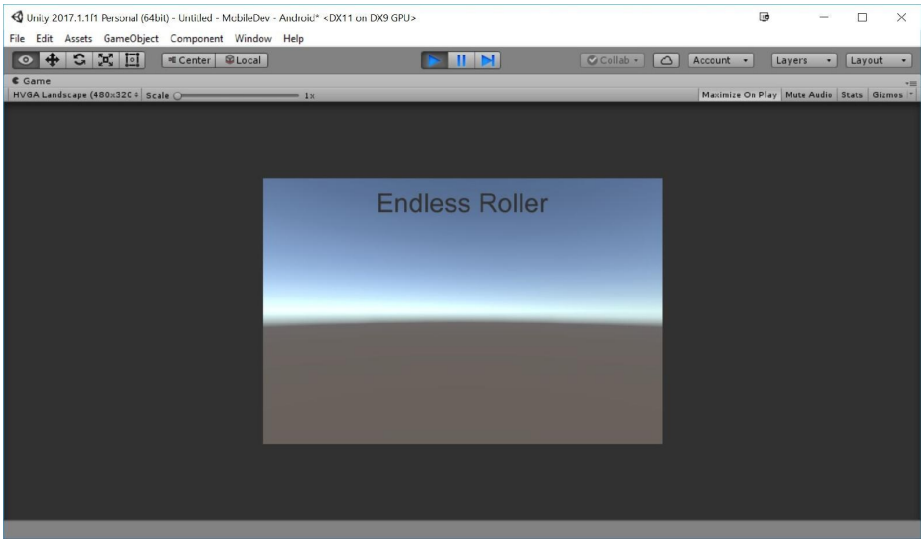
4. In the Inspector view, go to the Canvas Scaler component and change the Reference Resolution to `1920 x 1080` if it isn't there already.
5. Next, under Match, move it all the way over to Height. This will ensure that when the height of our screen changes, that's when we will scale.
6. Next, let's make the text a bit larger. Select the `Title Text` object and from the Rect Transform change the Width to `1000` and Height to `200`, and then change the Text component's Font Size to `130`:



7. Now, if we play the game with a higher resolution, it will display our title nicely, scaling up to fit the larger size that we have:



8. Go to the Game view control bar and pick a smaller resolution, such as the HVGA Landscape (480x320), and you'll note that the text will scale down to fit nicely as well:

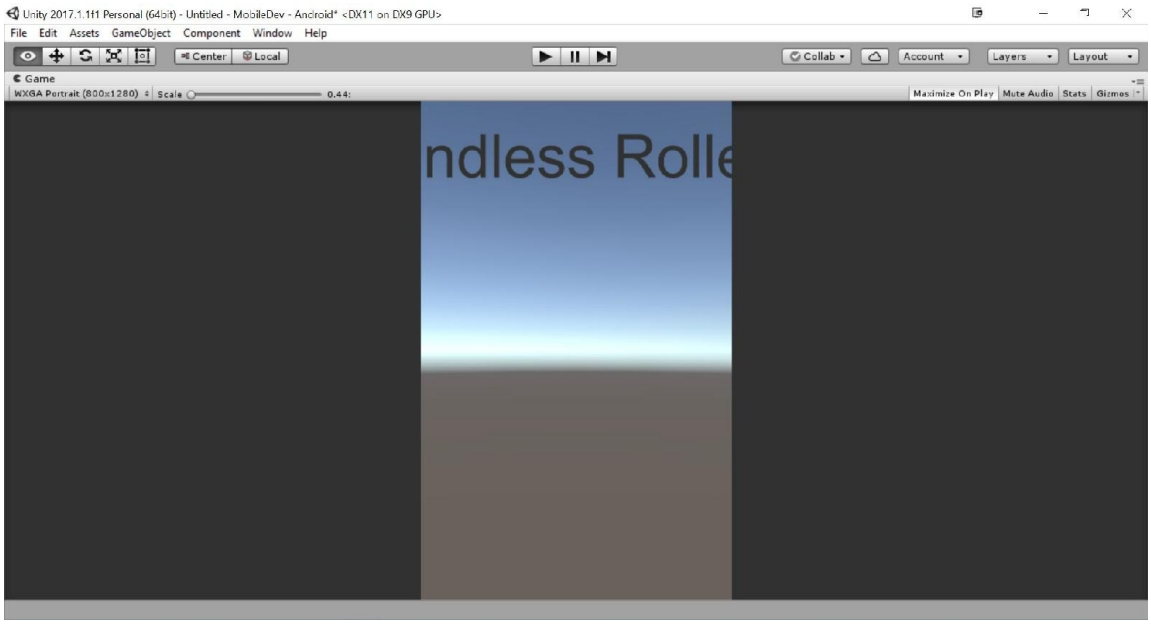


For more information on the Canvas Scaler, check out: http://docs.unity3d.com/Manual/script-CanvasScaler.html.

Selecting different aspect ratios

Like I mentioned previously, in the Game view, if we go to the control bar and select the first option, there is a drop-down menu where we can pick different resolutions to test our game, so we can find potential issues before exporting it to our devices. There are a number of them built-in for us by default, but we can also make our own using the + button at the bottom. I suggest that you make two new selections for your phone in the landscape and for portrait mode at the resolutions you are trying to reach (in my case, 1920 x 1080 and 1080 x 1920).

1. So, at this point, we can see that at a landscape ratio, it works fairly well, but let's try a portrait one:

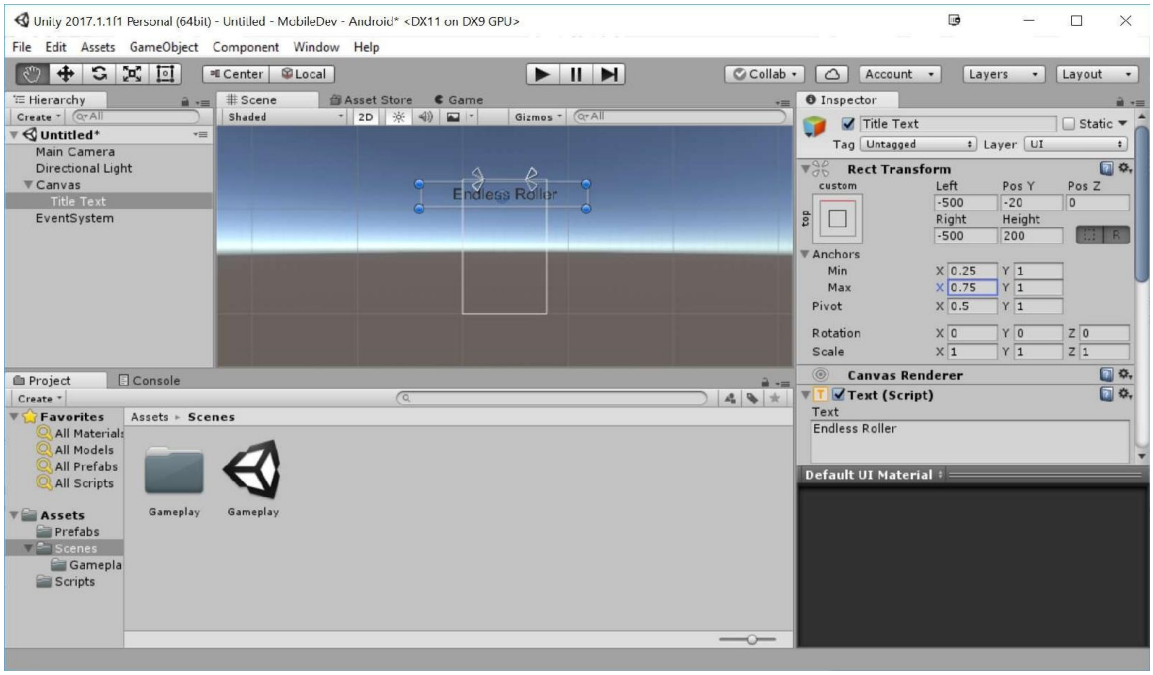


Ooops. Currently the text is overflowing past the bounds of the screen. Looks like we will have to fix that.

2. Select the `Title Text` object, and check the Best Fit property in the Inspector tab under the Text component.

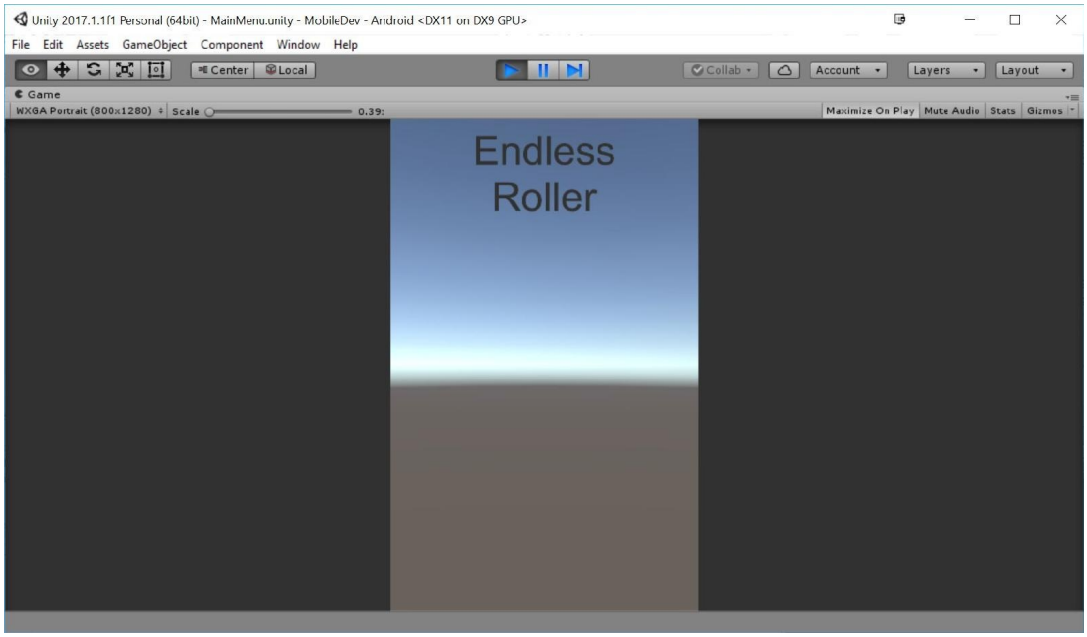
    This automatically scales the text to fit the space we have if the width and height were to change, which they currently don't, but we will change that next.

3. Now, go to the RectTransform component, and under Anchors, change the Min X to `0.25` and Max X to `0.75`:



   Note that now our RectTransform changed the Pos X and Width values. They have now been replaced with Left and Right properties, which are currently set to `-500` and `-500`. This means that the area being taken up by this is `-500` units away from our anchor at 25% and `-500` units away from our max anchor at 75%. We want the screen to resize to be at those anchors, so we will change both the Left and Right values to `0`.

4. Save our scene as a new file inside the `Scenes` folder called `MainMenu`, and then play the game:



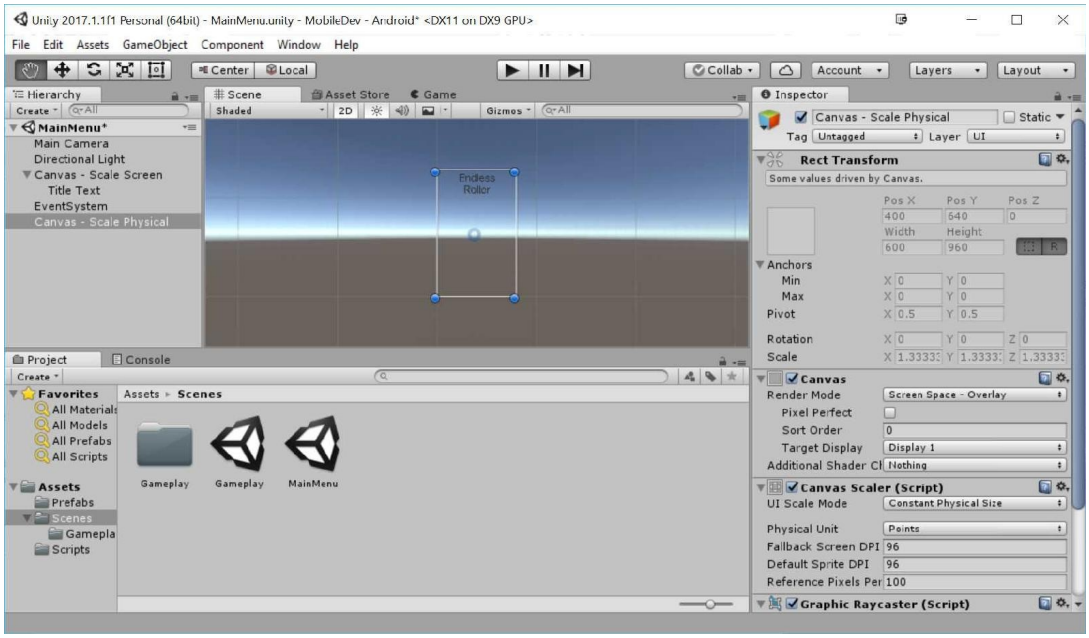
As you can see in the preceding screenshot, the text now fits a lot nicer. You'll also note that no matter what resolution we are using, this text takes up a good size of its fitting of a game's title.

Working with buttons

Now that we have the text displaying correctly, let's add the ability to move from the main menu into the game properly. However, unlike our title, for things that we want our players to touch, it's a good idea to make these buttons the same size in each device, as our fingers are the same size, no matter what device we are using. To show a possible solution for this, we will create a new Canvas using a different scaling technique.

1. We will first rename our current Canvas object to `Canvas - Scale Screen`. This way, we can easily tell whether we are using the correct canvas for this or not.

2. Now that we have that one ready, we can create our new one. Go to the top menu bar and then select GameObject | UI | Canvas. Rename this new Canvas to `Canvas - Scale Physical`. Then, under the Canvas Scaler component, change UI Scale Mode to Constant Physical Size:

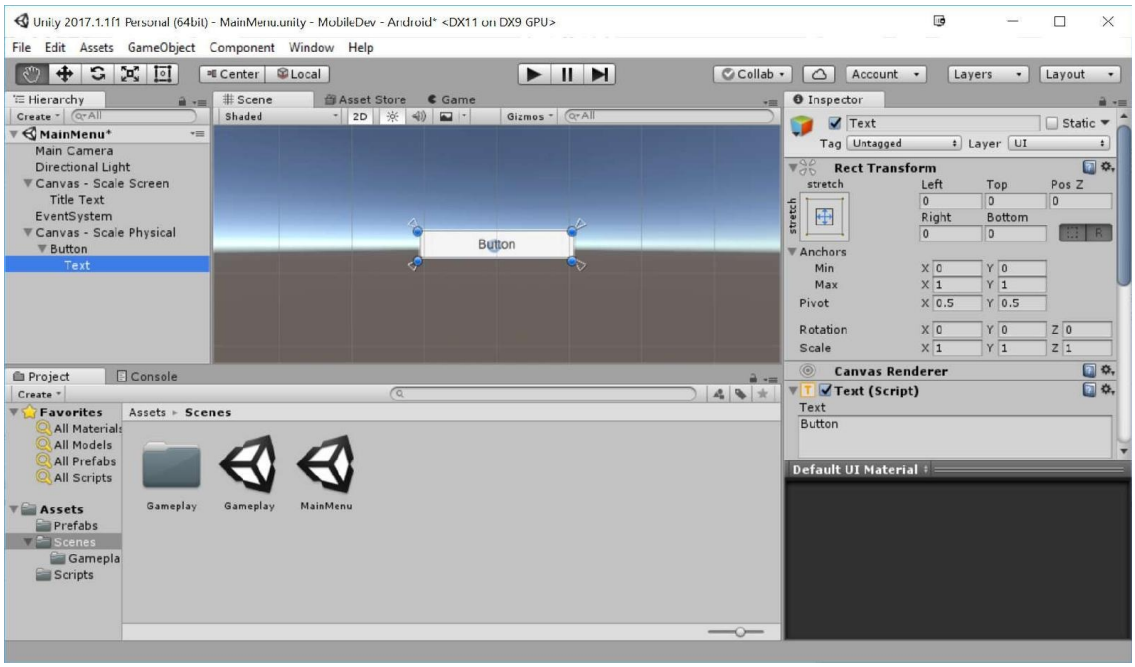


Using this method, Unity will attempt to scale the size of this canvas so that each element has the same physical size, regardless of the resolution. Since we're going for buttons that we intend to press with

our fingers, this makes a lot of sense.

3. Now, with this canvas selected in the Hierarchy view, go to the menu and select GameObject | UI | Button to create a new button inside of this canvas.

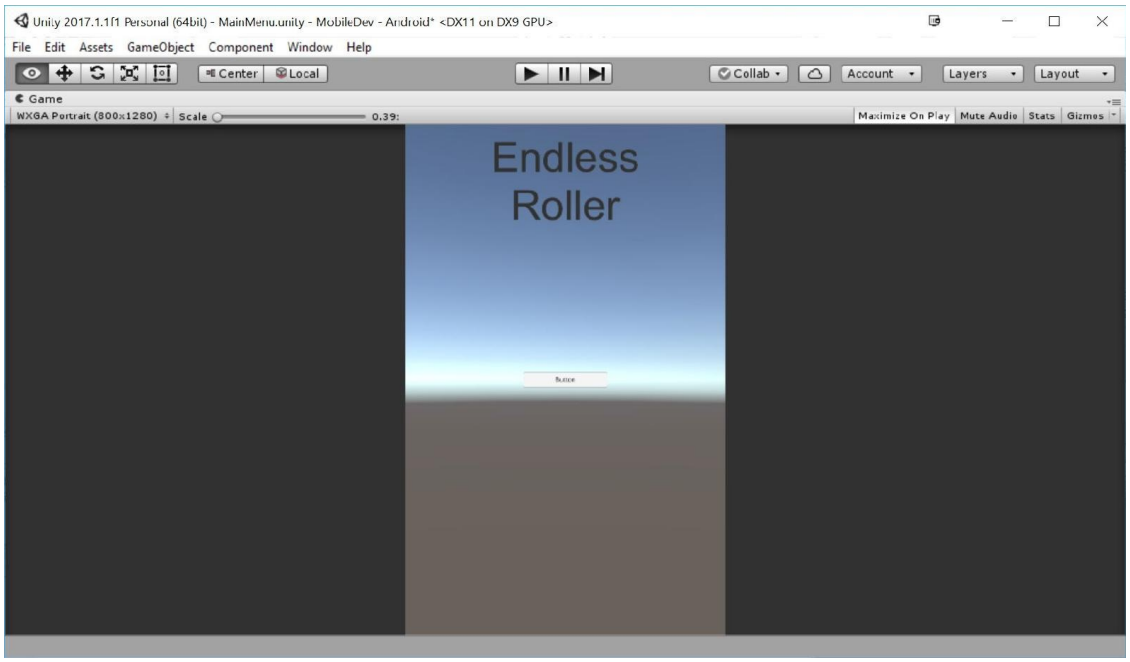
   At this point, you will see a new child object to Canvas called Button and if you were to extend that object, you'll see that it has a Text child also:



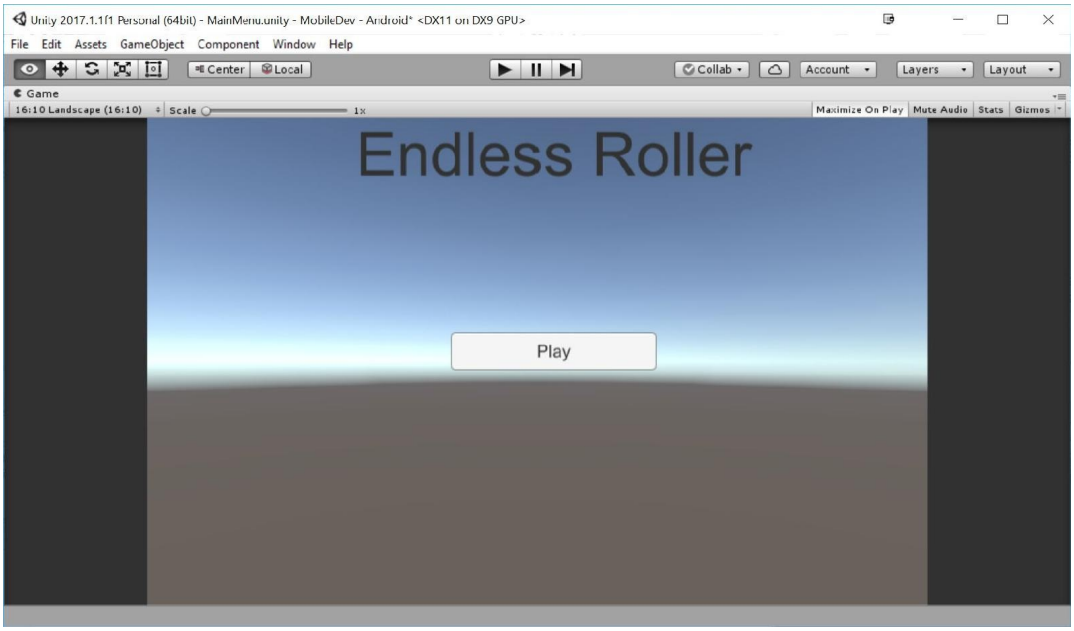
   The next question is, what size should our buttons be? Google suggests in their Material guidelines that at least 48 x 48 density-independent pixels should be used (dp for short), whereas Apple at their **Worldwide Developers Conference** (**WWDC**) recommends at least 44 dp x 44 dp. Either way, that comes somewhere around 8mm x 8mm or 0.3 inches x 0.3 inches.

To read the material guidelines, check out: https://material.io/guidelines/layout/metrics-keylines.html#metrics-keylines-sizing-by-increments.

   If you were to look at the game right now at our standard resolution, you may be a bit scared due to the size of the button right now, depending on the resolution:



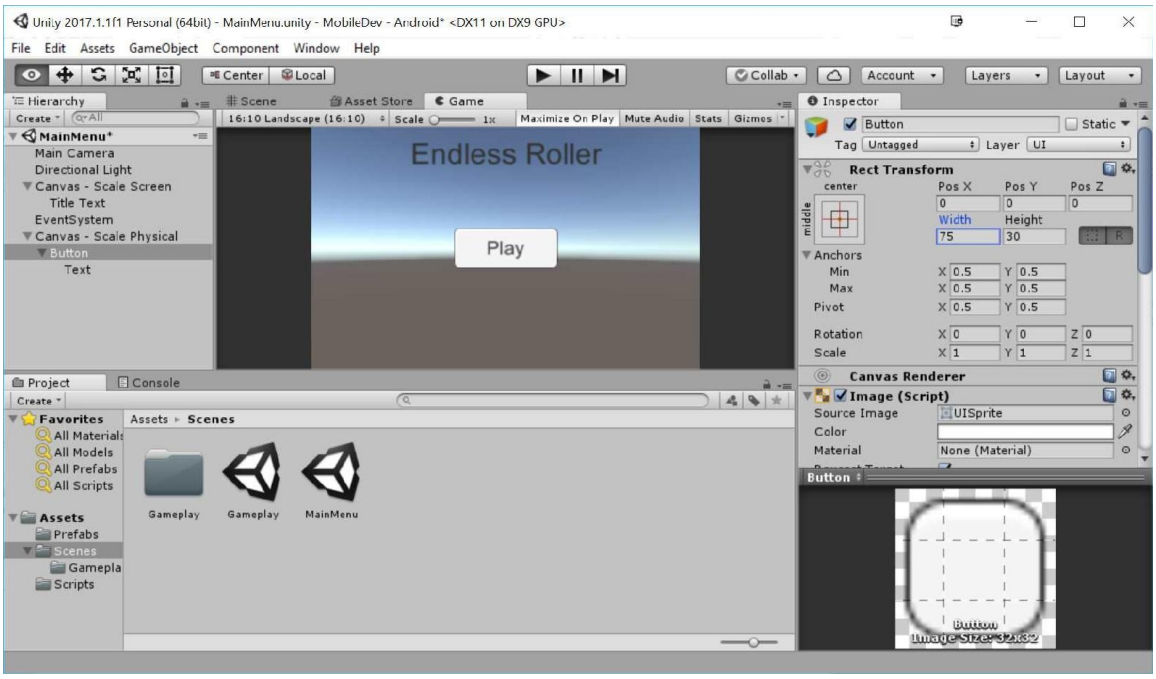
That's because our button size is assuming that the **dpi** (**dots per inch**) value is 96 when on devices such as the OnePlus 3, and the iPhone 6/7/8 Plus is around 400, or four times larger than what it is now. For right now, I'll change the Aspect Ratio value to 16:10 Landscape to see something closer to what we'll be using on our device when we play there:



If you're interested in finding out what the DPI for your device

is, check out: *http://dpi.lv/.*

4. From the Hierarchy tab, expand the `Button` object and change the button's Text child's Text component's text value to `Play`.
5. The size of the button is quite large for the space available, so let's change the `Button`'s Rect Transform component's Width to `75`:



We now have a button, but it doesn't actually do anything yet. Let's fix that now.

6. Let's create a script to contain the functionality that we want. From the Project view, open up the `Scripts` folder and let's create a new C# script called `MainMenuBehaviour`.

7. Once your IDE has opened, use the following code:

```
using UnityEngine;
using UnityEngine.SceneManagement; // LoadScene

public class MainMenuBehaviour : MonoBehaviour
{
    /// <summary>
    /// Will load a new scene upon being called
```

```
    /// </summary>
    /// <param name="levelName">The name of the level we want
    /// to go to</param>
    public void LoadLevel(string levelName)
    {
        SceneManager.LoadScene(levelName);
    }
}
```

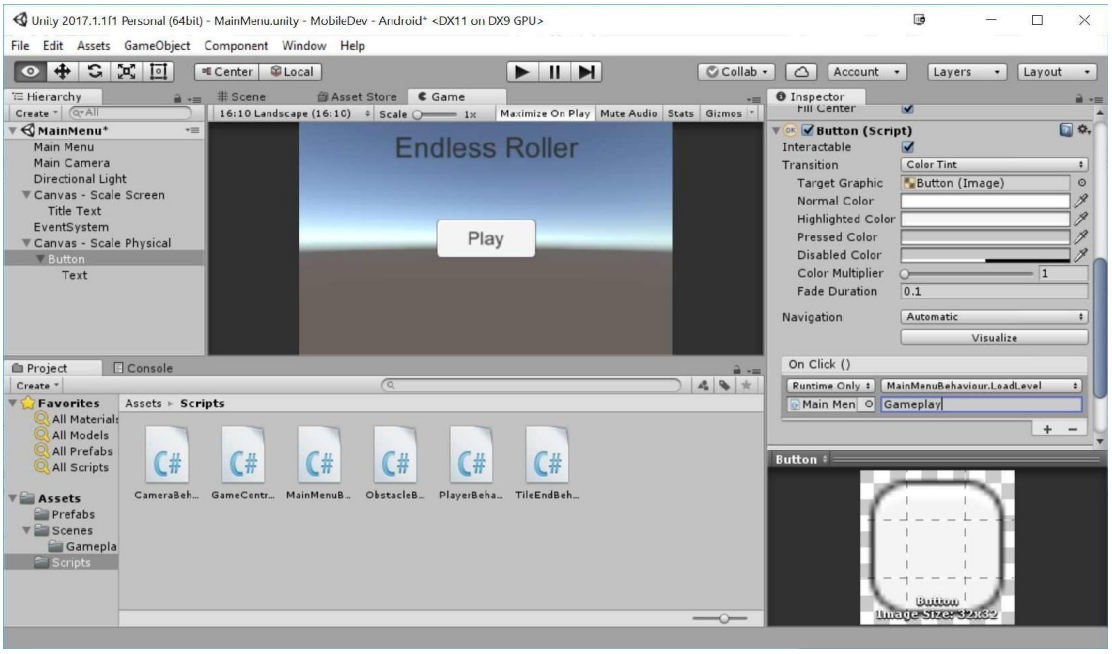
The `LoadLevel` function will load a level based on the name that we provide to it making use of Unity's Scene Manager, which we added using a statement at the top of our code so that we would have access to that namespace.

8. Save the script and go back to the Unity editor. To call Unity's UI Events from the editor, we will need to have a game object with the `MainMenuBehaviour` component attached to it to call this function from. We could use one of the currently existing objects, but we'll just create a new object, making it easier to be found in the future.

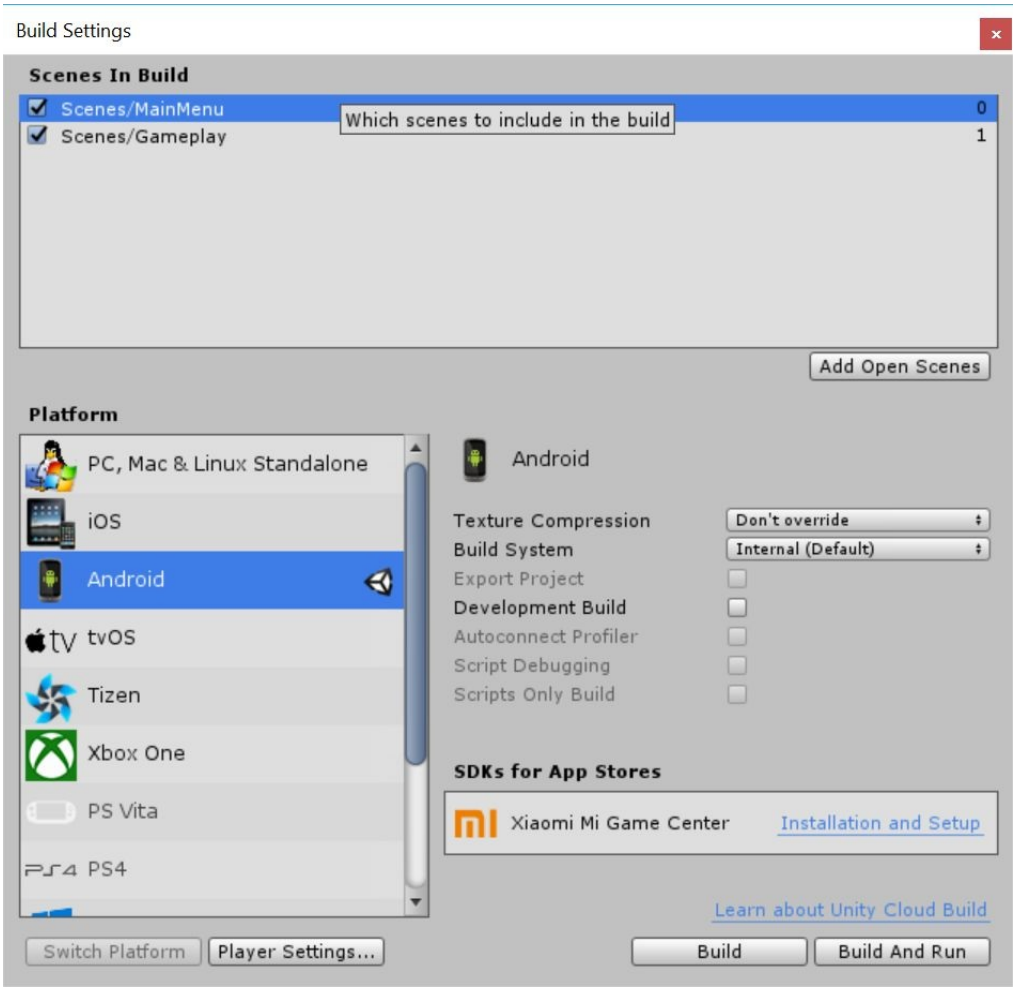
9. With that in mind, create an empty game object in your scene called Main Menu and then add the `MainMenuBehaviour` script to it. Then, drag and drop it to the top of the Hierarchy tab to make it easier to access in the future and reset its position for the sake of neatness.

10. Select your Play button, and go to the Inspector tab and scroll down to the Button component from there. Then, in the On Click () section, click on the + icon to add something for our button to do.
11. Then, drag and drop the Main Menu object from the Hierarchy tab to the area that currently says None (Object), which is added to the list.

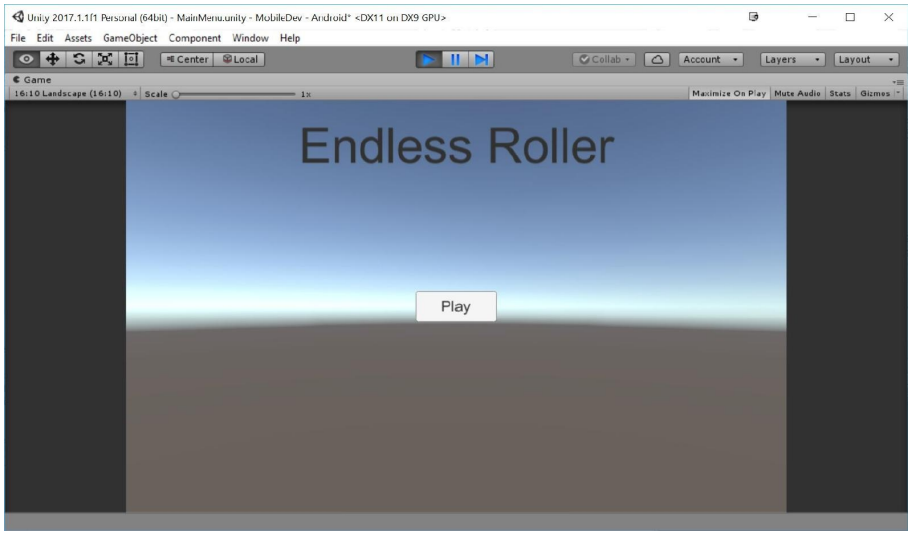
12. Click on the dropdown that currently says No Function and then select MainMenuBehaviour.LoadLevel. Then, in the textbox that appears below, type in the name of our game's level, `Gameplay`:



13. Lastly, open the Build Settings like we did before by going to File | Build Settings and add our MainMenu to the list at index 0 by selecting Add Current, and then dragging the MainMenu level to the top so that it will be the level that starts off when we start the game:

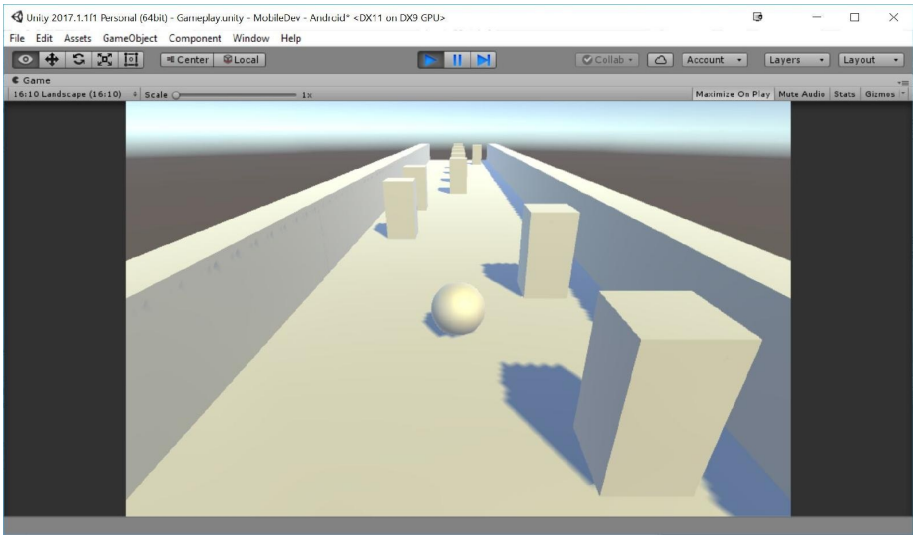


14. Save your project and scene and click on the Play button:



At this point, our main menu is working well, and we can get into the game

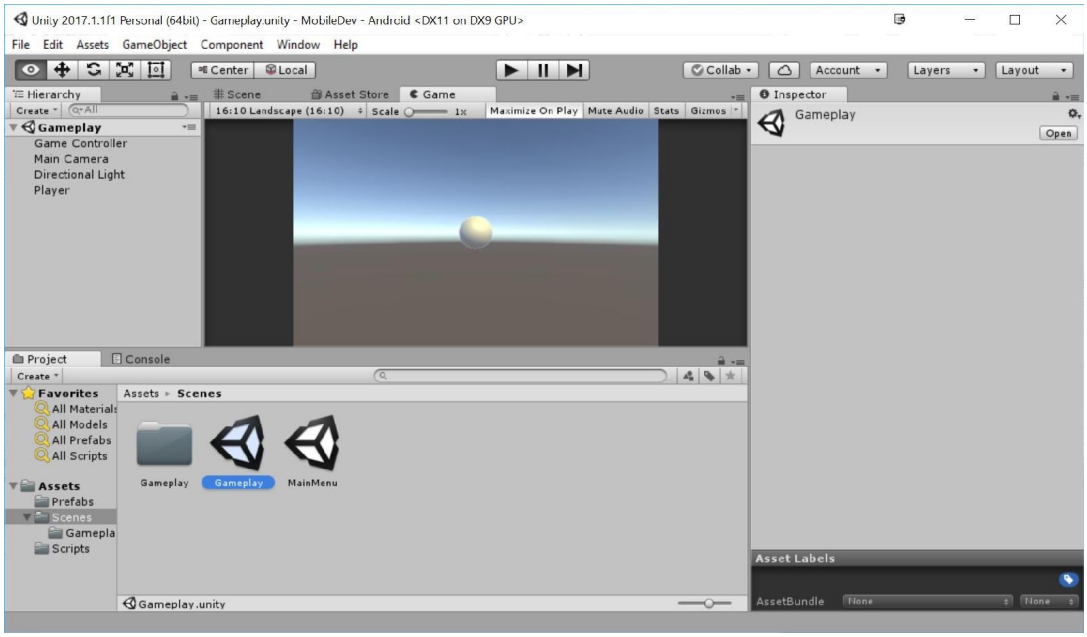
without any issues by clicking on the Play button:



Adding a pause menu

Now that we have the main menu, we will move on next to building something else that most games will need, a pause menu. In PC games, this will likely be triggered by the *Esc* key, whereas in a mobile game, it typically needs its very own button. We will make it so that this project supports both.

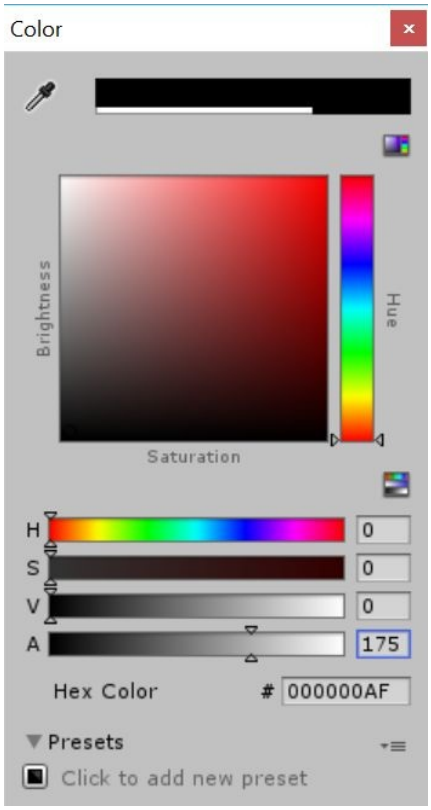
1. To start off, let's open up the Gameplay scene by going to the Project window and then opening the `Assets/Scenes` folder and double-clicking on Gameplay, thus saving the MainMenu level if you didn't do so already:



   Before we create a button to open our pause menu, let's go ahead and create the pause menu first that we'll be opening.

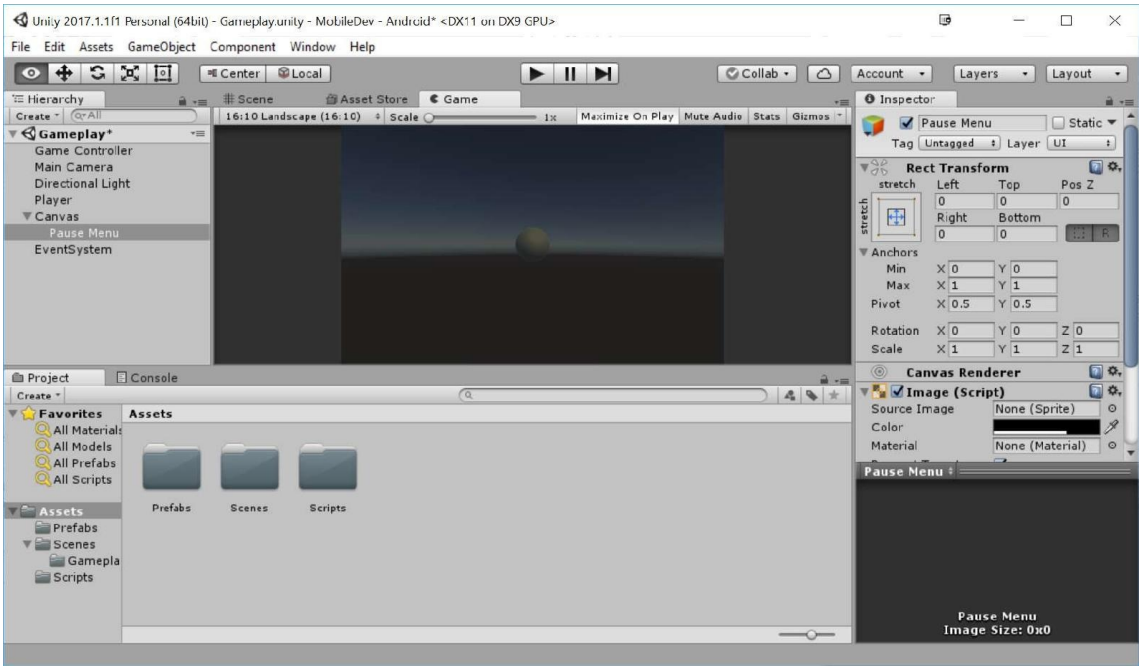
2. The first thing we'll do is dim our screen when we enter the pause menu. An easy way to do that is having an image scale to cover our entire screen, which is what the `Panel` object does by default. We can create it by selecting Game Object | UI | Panel. Note that this creates a Canvas and an EventSystem object in addition to the Panel because one doesn't exist already.

3. Rename Panel to Pause Menu. Then, we'll change the Color property of the object's `Image` component to black with a higher transparency by increasing the alpha channel (A) to `175`:



   The Image component works in a similar manner to the Sprite Renderer for 2D games with information on an image to draw and the color to use for it.

4. The current image has a thin border, which I'm not a fan of, in this case. You may keep it if you'd like, but I'm going to remove it and change the Source Image to None (Sprite) by selecting the current one and pressing the *Delete* key:



Now that we have this, we will need to populate the menu with content. In this case, we will have a text object saying that the game is paused and some buttons allowing the player to resume, restart, or return to the main menu.

5. Let's create another panel to hold our pause menu contents. We want this panel to be a child of our Pause Menu object, so we can do this easily by going to the Hierarchy window, right-clicking on the `Pause Menu`, and selecting UI | Panel.

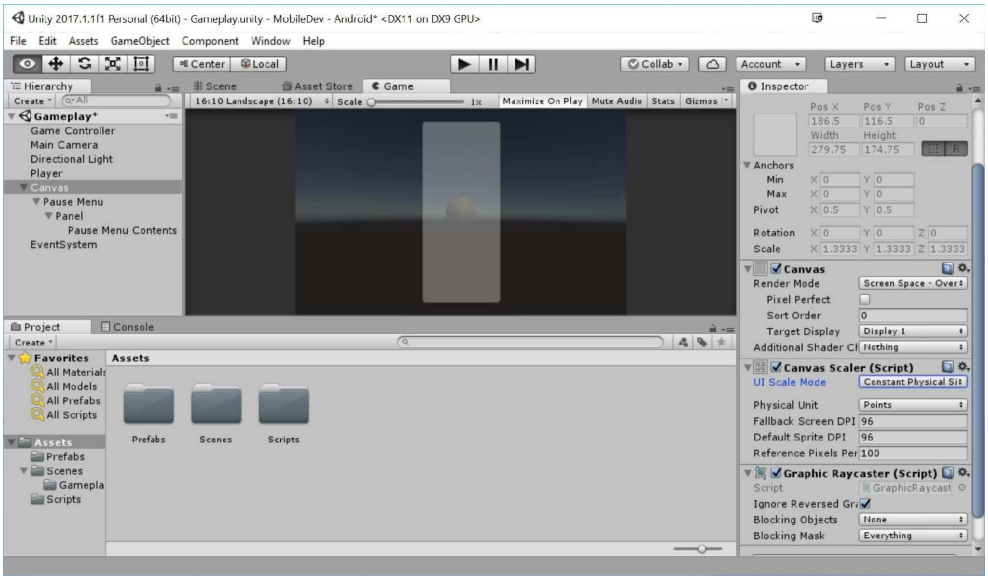
   Now, for this panel, I don't want it to take up the entire screen, so I will use another component to modify its size based on the resolution we receive. In this case, I will use an Aspect Ratio Fitter component.

6. From the Inspector window, scroll all the way down and then select Add Component and start typing in `Aspect`. From there, select Aspect Ratio Fitter and then press the *Enter* key.

7. Afterward, go to our newly added component and change the Aspect Mode to `Fit in Parent` to ensure that the panel will always fit within our screen and set the Aspect Ratio to `0.5`. This means that it will be twice as high as it is wide (width over height, which means ½—`0.5`).

If you go to the Game window and switch aspect ratios, you'll note that the menu will stay in a similar shape.
For more information on the Aspect Ratio Fitter component check out:
https://docs.unity3d.com/Manual/script-AspectRatioFitter.html.

8. This is good, but I don't want to have the panel stuck directly to the edge of our screen, so we will make this object invisible by clicking on the checkmark by the Image component.
9. Then, right-click on the Panel object and create another panel by going to UI | Panel. Rename this new object as Pause Menu Contents and then change the Rect Transform component's left, right, top, and bottom values to 10 to give us a border around the screen.
10. We will use physical buttons like last time, so let's move to the Canvas object, and under the Canvas Scaler component, change the UI Scale Mode to Constant Physical Size:



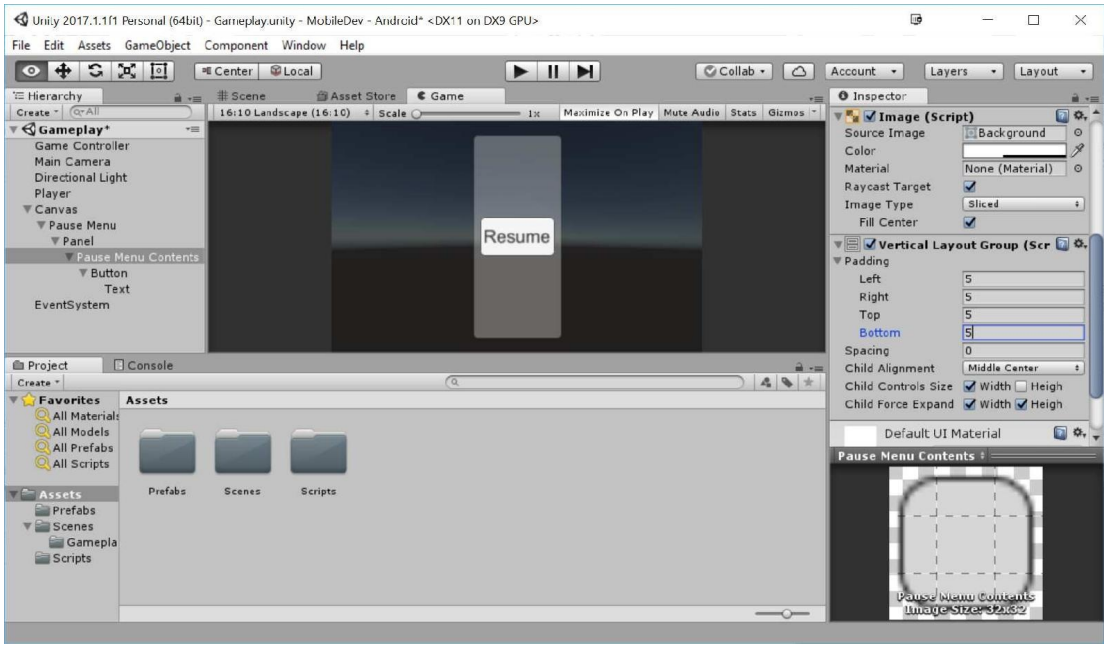
We could place everything manually like we did previously by hand, but in this case, we may want to use another feature that Unity's UI System has: Layout Groups.

Layout Groups will make resize children of an object with that component automatically to fit the area of the parent. There are several different layout groups, including grid-based, horizontal, and

vertical. In our case, the menu will probably be vertical.

For more information on Unity's way of automatically creating layouts, check out: *https://docs.unity3d.com/Manual/UIAutoLayout.html.*

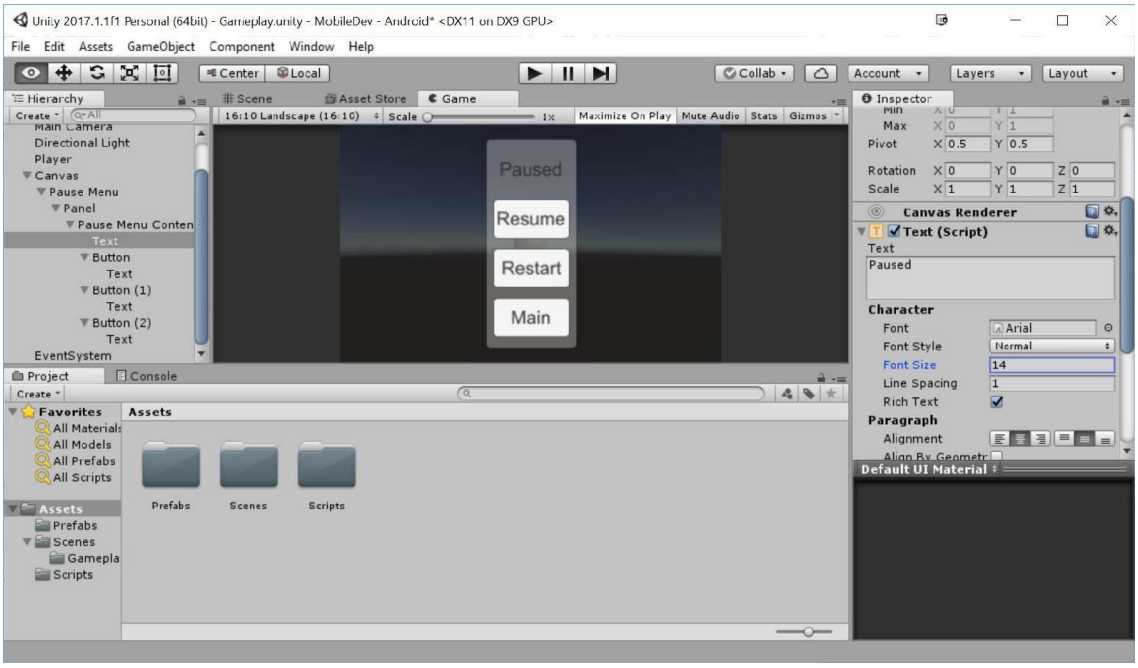
11. Select the Pause Menu Contents object in the Hierarchy window and then switch to the Inspector window. From there, scroll all the way down to the Add Component option and select it. Type in Vertical Layout Group and select it.
12. Let's create some children to fit in our menu. From the Hierarchy window, right-click on the Pause Menu Contents object and select UI | Button.
13. This creates a button, but you'll note that it looks pretty much like a normal button. Let's open up its child Text object and change the text to Resume.
14. Afterward, select the Pause Menu Contents object and under the Inspector window go to the Vertical Layout Group (Script) component and change the Child Alignment to Middle Center. Then, change the Child Control Size to have Width toggled.
15. Then, in the Vertical Layout Group component, click on the arrow to the left of the Padding property to open it up and then set all of the sides to 5:



This will add 5 pixels of padding in each direction within all of the children of the layout group.

16. Now, duplicate this button twice and change the text to Restart and Main Menu.

17. Next, right-click on the Panel object and select UI | Text. Change the object's text to Paused and change its alignment to be centered and increase the size until it's somewhere you like. Note how the order in which the child is changes the order it is displayed. With that in mind, drag the Text object to the top:

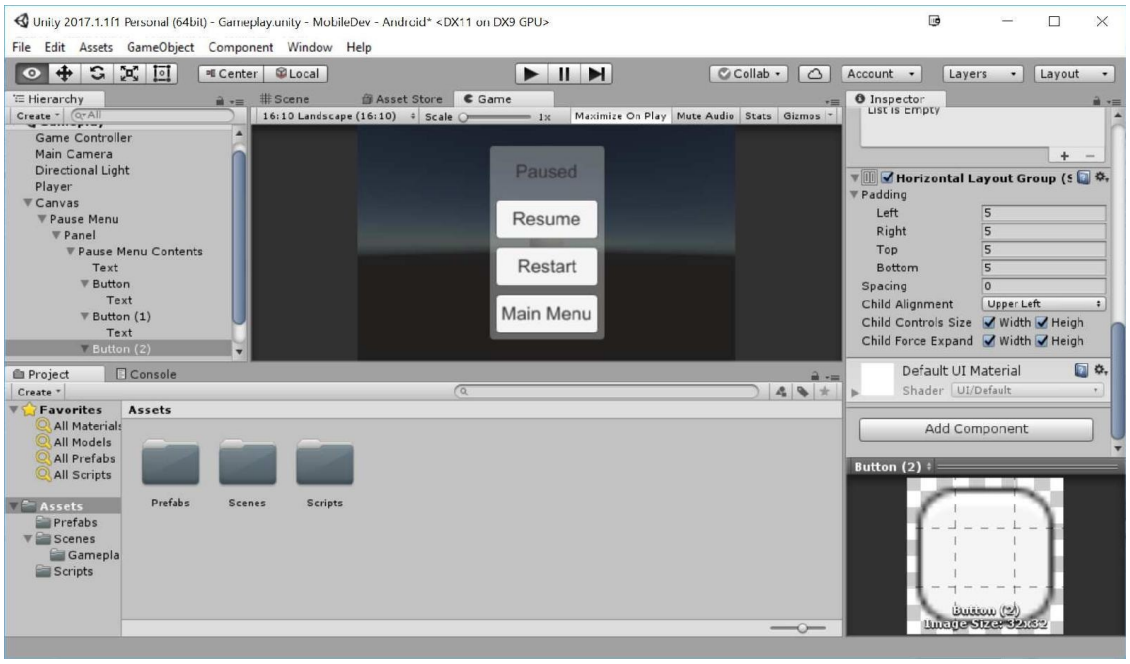


This looks nice, but there's also a lot of spacing here, and we can't see entirety of the Main Menu on this button. So, if we'd like, we can inste condense the contents of our menu to just fit what we have there.

18. To do this, we can go to Hierarchy and select the Pause Menu Contents object and then add a Content Size Fitter component. Once it is added, we will change the Horizontal Fit and Vertical Fit to Preferred Size.

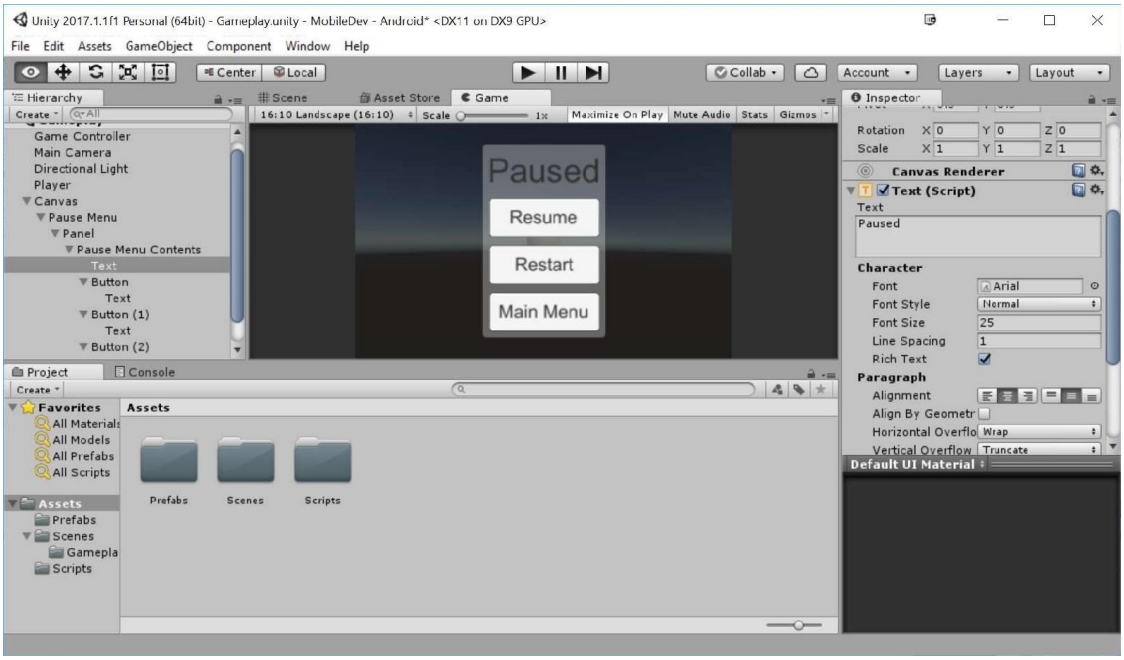
19. This will scrunch all the buttons together, so we can change the Vertical Layout Group's Spacing property to 5 and add some space between the buttons.
20. Now, to make sure that the buttons fit no matter what size we have, select each Button and add a Horizontal Layout Group (Script) to them. From there, check the Child Controls Size for Width and Height and add a

padding of 5. Once you create the first Horizontal Layout Group (Script), you can right-click on it and select Copy Component, then go to the other two buttons, right-click on the Inspector tab at the Rect Transform component and then select the Paste component:



In this case, the Child Controls Size property will make it so that the button will increase its size to ensure that it fits the size of the text provided.

21. We will also go to the paused Text object and increase the Text component's Font Size to 25 to fill out the area and emphasize that we are in the pause menu:



22. Now that we have the buttons themselves, let's not actually make them do something. In the Project window, open up the `Scripts` folder and create a new C# Script called `PauseScreenBehaviour` and double-click on it to open up the IDE of your choice.
23. Once it's opened, use the following code:

```
using UnityEngine;
using UnityEngine.SceneManagement; // SceneManager

public class PauseScreenBehaviour : MainMenuBehaviour
{

    private static bool paused;

    [Tooltip("Reference to the pause menu object to turn on/off")]
    public GameObject pauseMenu;

    /// <summary>
    /// Reloads our current level, effectively "restarting" the
    /// game
    /// </summary>
    public void Restart()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    /// <summary>
    /// Will turn our pause menu on or off
    /// </summary>
    /// <param name="isPaused"></param>
    public void SetPauseMenu(bool isPaused)
    {
        paused = isPaused;
```

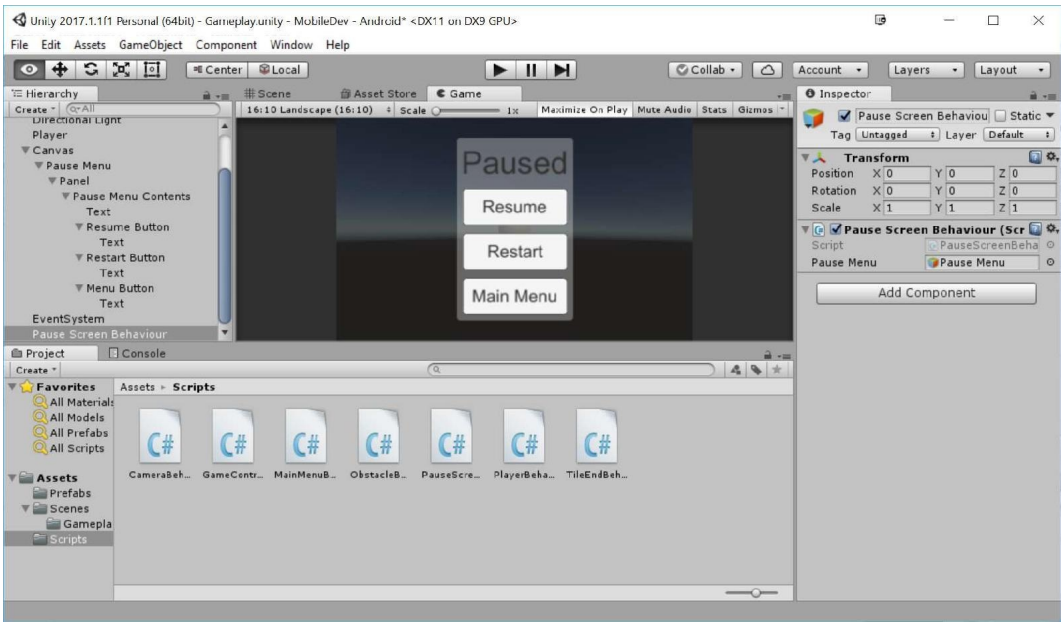
```
        // If the game is paused, timeScale is 0, otherwise 1
        Time.timeScale = (paused) ? 0 : 1;
        pauseMenu.SetActive(paused);
    }

    void Start()
    {
        paused = false;
    }

}
```

In this script, we will first use a `static` variable, which is called `paused`. When we declare a variable `static`, we ensure that there will only ever be one of those variables inside of this class, which all instances will share. One of the advantages of this is that we can access the property in other scripts using the class name followed by a period and then the attribute's name (in this case, `PauseScreenBehaviour.paused`). We will use this concept later on when we want to open the menu through code.

We then have two public functions, which we will call via the UI elements. First, we have a `Restart` function, which will use Unity's Scene Manager to return us to the current level loaded, effectively restarting the game. It is important to note that `static` variables do not reset when restarting in Unity, so that's why I set paused to false in the `Start` function to ensure that when we come to the level, it is unpaused.

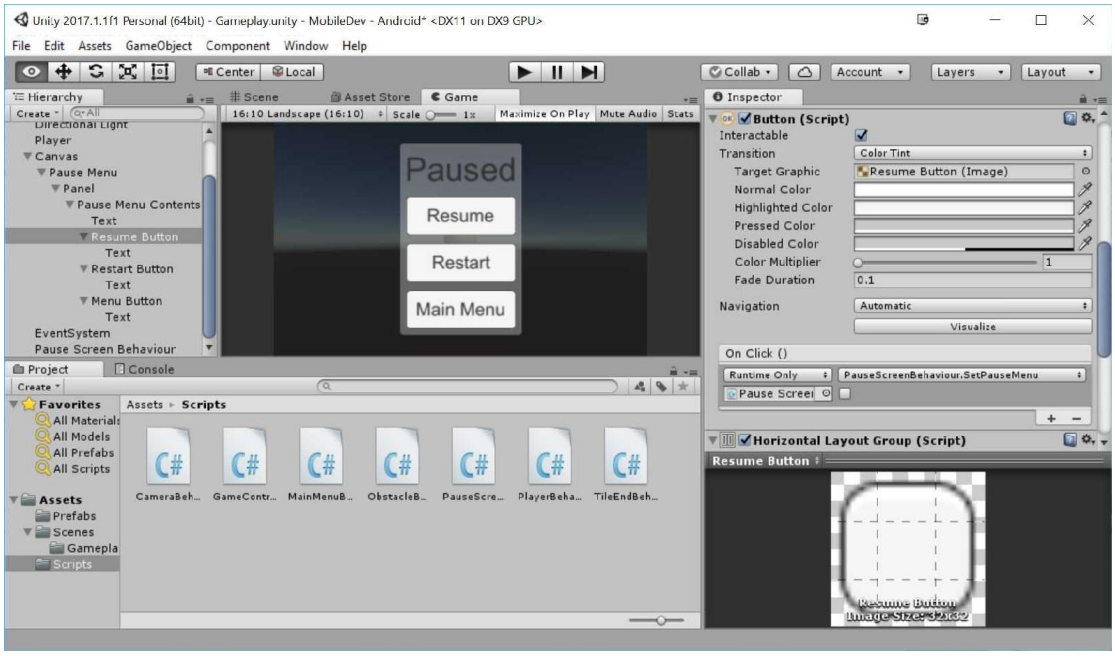
Finally, I have a `SetPauseMenu` function, which will turn the pause menu on or off based on the value of `isPaused`. It also sets the `Time.timeScale` property, where `0` means that nothing will happen and `1` means normal time. This property will modify the `Time.deltaTime` variable, effectively canceling out movement that we have as long as we use it.

24. Save your script and dive back into Unity.
25. Now, let's dive into the three `Button` objects and rename them to Resume button, Restart button, and Main Menu button for clarity's sake.
26. Then, we'll create a new empty game object by going to GameObject | Create Empty. We'll name it to Pause Screen Behaviour (Script) and then attach the Pause Screen Behaviour component to it. Next, assign the Pause Menu variable to the Pause Menu game object in the Hierarchy tab:



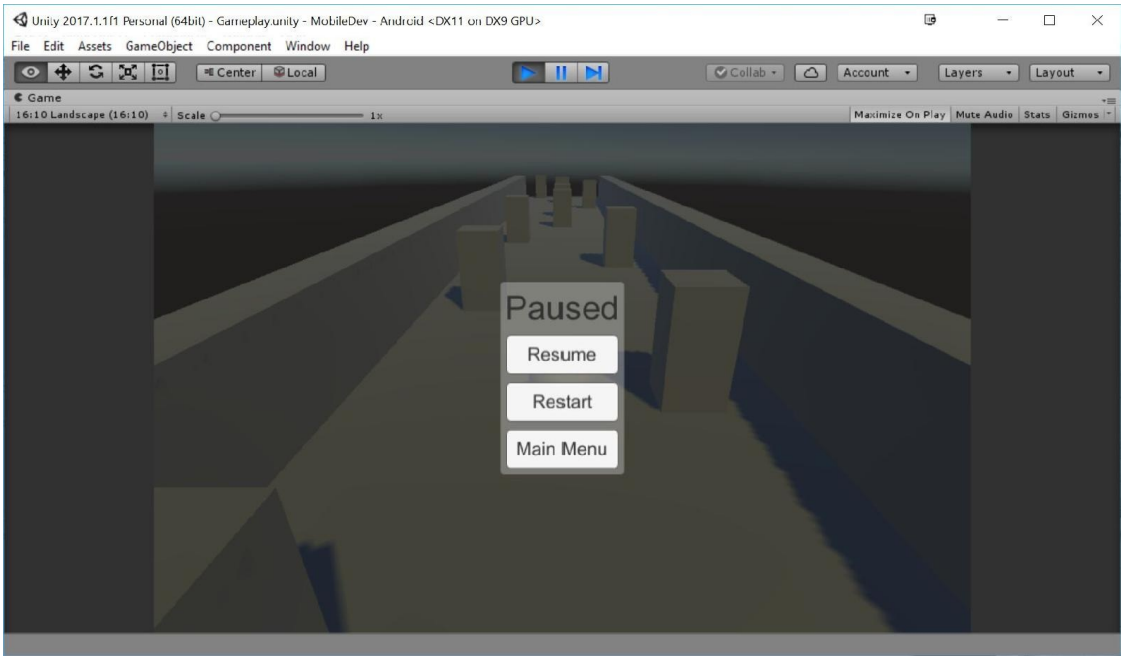
27. Now that we have the script, we can now change the buttons to actually do something. Go to the Inspector window with it selected and go to the Button component's On Click () section and click on the + button to add an action to occur.

28. Drag and drop the Pause Menu Behaviour (Script) object from the Hierarchy window into the box on the bottom-left side of the On Click () action in the Inspector window. Next, go to the dropdown and select Pause Menu Behaviour | SetPauseMenu. By default, it's on false due to not being checked, so this should work for us:



29. Likewise, do the same for the Restart button object, this time calling the `Restart` function.

30. Next, do the same for the `Menu Button` object, except call `LoadLevel`, and put the name of our main menu level in the string place (`MainMenu`, in my case).
31. Save our game and go ahead and run the game:



As you can see in the preceding screenshot, if we restart the game, it works

correctly--we can go to the main menu and Resume continues the game.

Pausing the game

At this point, we have some issues. Once the menu is gone, there is no way to get it back, the game should start unpaused, and the game should actually pause. Let's tackle these issues next:

1. Open up the `PlayerBehaviour` script and replace the bottom of the Update function with the following (changes are highlighted, note the removal of the original way of calling `rb.AddForce`):

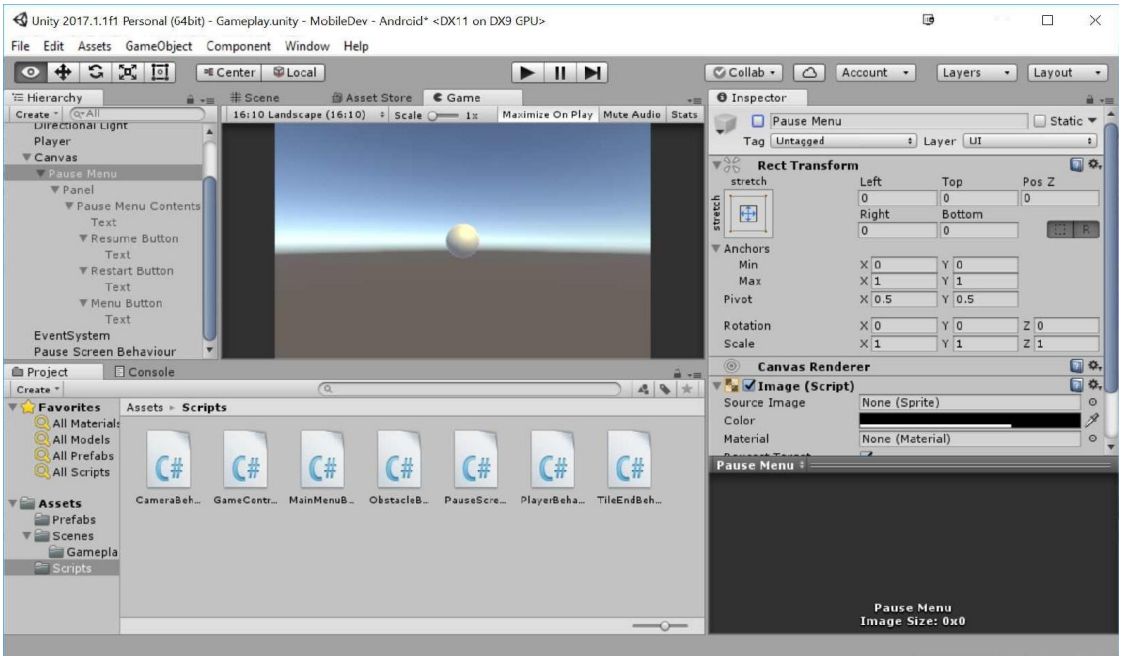
```
/// <summary>


    /// Update is called once per frame


    /// </summary>
    void Update () {


        // If the game is paused, don't do anything
        if (PauseScreenBehaviour.paused)
            return;







        // Movement in the x axis


        float horizontalSpeed = 0;







        // Check if we are running either in the Unity editor // or in a standalone
build.
        #if UNITY_STANDALONE || UNITY_WEBPLAYER || UNITY_EDITOR
```

```
        // Check if we're moving to the side horizontalSpeed =
Input.GetAxis("Horizontal") *

                              dodgeSpeed;




        // If the mouse is held down (or the screen is tapped // on Mobile)

        if (Input.GetMouseButton(0))

        {

            horizontalSpeed = CalculateMovement(Input.mousePosition); }




        // Check if we are running on a mobile device #elif UNITY_IOS || UNITY_ANDROID




        // Move player based on direction of the accelerometer horizontalSpeed =
Input.acceleration.x * dodgeSpeed;

        // Check if the screen has been touched if (Input.touchCount > 0)

        {

            // Store the first touch detected.

            Touch touch = Input.touches[0]; //horizontalSpeed =
CalculateMovement(touch.position); SwipeTeleport(touch);
```

```
            TouchObjects(touch);


        }


        #endif


        var movementForce = new Vector3(horizontalSpeed, 0, rollSpeed);




        // Time.deltaTime is the amount of time since the // last frame (approx. 1/60
seconds) movementForce *= (Time.deltaTime * 60);


        // Apply our auto-moving and movement forces rb.AddForce(movementForce);


    }
```

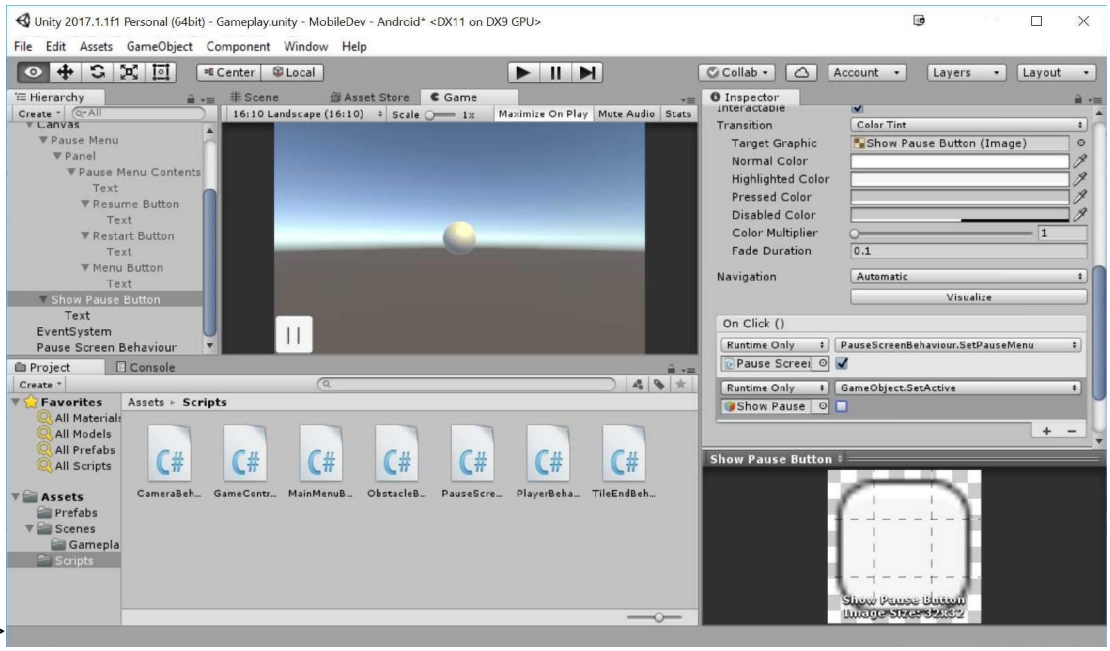
First of all, if the game is paused, we will not do anything within the function. Also, instead of just adding a force always, we now multiply that value by `Time.deltaTime`, which is how much time has elapsed since the previous frame. This makes it so that as computers get better, our code will work the same.

2. Now, the game, by default, should be unpaused, so let's go ahead and select the Pause Menu object in the Hierarchy view and then click on the active button in the Inspector view to disable it:



3. In addition to this, we will need a way to turn it on. On PC games, this is usually the *Esc* button, but, for mobile, we will instead have a button that players can click to turn on the menu. Go ahead and right-click on the Canvas object in the Hierarchy view and select UI | Button.

4. Rename the new object as Show Pause Button and use the Anchor Presets option to place the object on the bottom left of the screen (use *Alt* + *Shift* to set Pivot and Position as well). Then, change the Width to `30`, as it won't need to be that large.
5. Open up the Text and change the Text component's Text property to `| |` to make it look like a pause button.
6. Go back to the Show Pause Button object and create an On Click () event using the SetPauseMenu function on the Pause Menu Behaviour object. Then, click on the checkbox to set it to pause.
7. Now, we want to remove this pause button when we click. We could do this through code, but just to show that we can also do this via the editor, go ahead and click on the + again to add another action on On Click (). Next, drag and drop the actual Show Pause Button and then call the GameObject | SetActive function. Then, uncheck it to turn the object off:



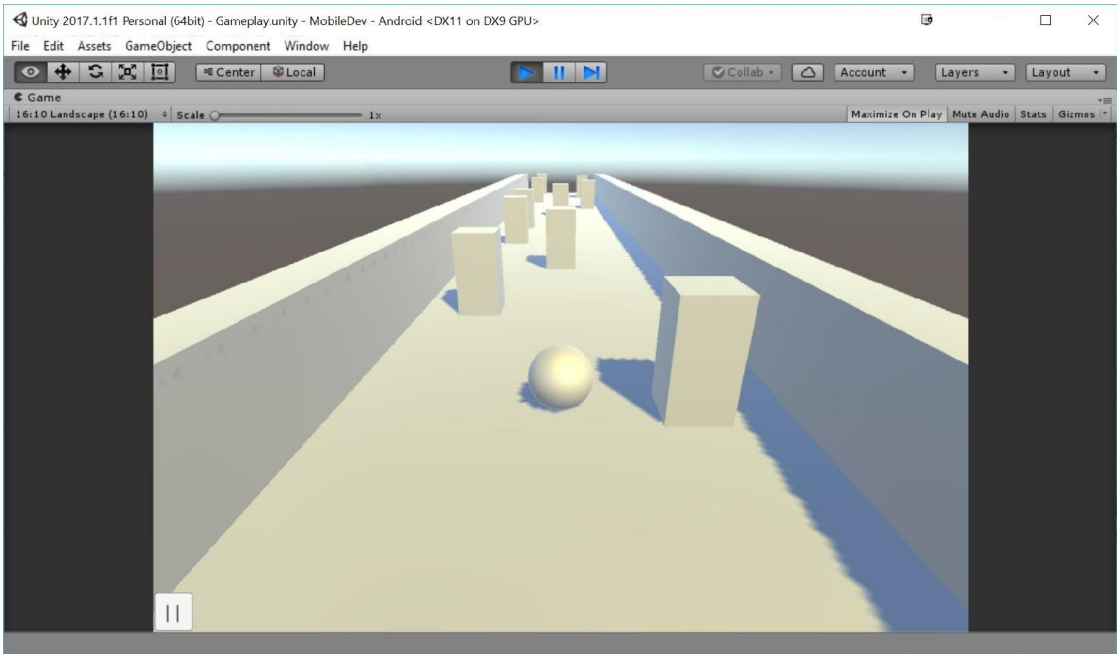
8. Go back to the Resume Button and add another event to its button to turn the Show Pause Menu button back on when we leave using SetActive like we used in the previous step.

    As mentioned previously, one problem that won't be apparent now unless you restart the level is the fact that `static` variables will keep their values each time we reload the game. In our case, we set `paused`, which turns our `Time.timeScale` to `0`. Thankfully, we can fix this fairly easily.

9. Lastly, open up the `PauseMenuBehaviour` and update the `Start` function to have the following:

```
void Start()

    {

        paused = false;

        SetPauseMenu(false);

    }
```

10. Save your script and the scene, and then play the game:



The Pause menu now works correctly.

Summary

With that, we've gotten a good foundation to build on when creating UI elements for a mobile game. We first covered how to create a title screen making use of Buttons and Text objects. We then covered how to use Panels and Panels, Button, Text, and Layout Groups to make your menus adapt to the size of your elements. We also touched on Layout Groups and how they can arrange our objects to fit in a pleasing manner. Finally, we integrated the pause menu into our game itself and made it work with everything in our project. We will be revisiting these concepts in later chapters, so keep these explanations in mind.

In the next chapter, we will dive into monetization and take a look at just how we can add Unity Ads to our project.

Advertising Using Unity Ads

When working on mobile titles, one needs to think about how they are going to sell their game. Deciding on how to best sell a game can be difficult. Of course, you can sell your game for a price, and there is a possibility that it will be successful, but you'll be limiting your audience numbers to a much lower amount. This could work well for a niche game, but if you're trying to make a game with a broad appeal where you want to get as many players as possible to play your title, you may have some issues.

Having a price on the game can be a major hurdle in getting those initial customers who can share the game via word of mouth and contribute to having more people play your game. To solve this potential issue, you do have the option of making your game free. Afterward, you can give players the opportunity to purchase things or show advertisements when playing the project.

That's not to say that having a bunch of advertisements in a free game is the best option either. Having too many ads, or even the wrong kind of ads, can drive users away, which can be even worse. Many developers each have their own opinions on whether it's a good idea to use ads or not, but that's not the purpose of this chapter. In this chapter, we will look into different options available to us for advertising over the course of our game and show how to implement them, should you choose to add the content to your game.

Chapter overview

In this chapter, we will integrate Unity's Ad framework into our project and learn how to create both simple and complex versions of advertisements.

Your objectives

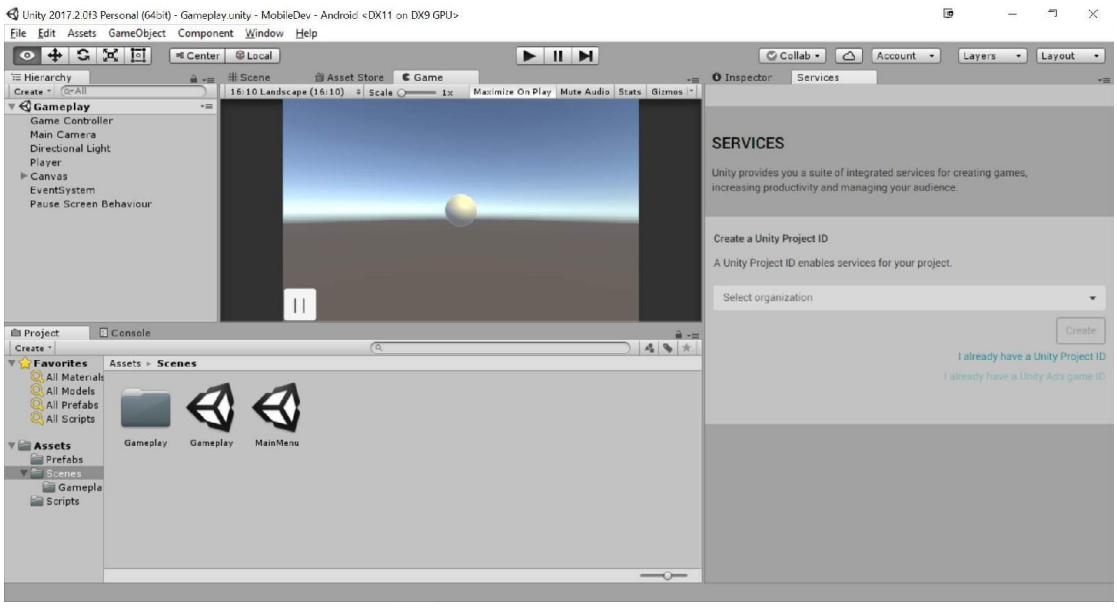
This chapter will be split into a number of topics. It will contain a simple step-by-step process, from beginning to end. The following is the outline of our tasks:

- Unity Ads setup
- Creating a simple ad
- Adding in ad callback options
- Opt-in advertisements with rewards
- Integrating a cooldown timer

Unity Ads setup

Unity Ads is a video ad network for iOS and Android that can monetize your existing player base by showing ads. Unity Ads offers video ads that can be shown as either rewarded or non-rewarded placements. Before we can enable Unity Ads, we must first enable Unity's Services suite; so, let's do that first. To activate Unity Services, you have to link your project to a Unity Service Project ID, which is how Unity can tell the difference between the different projects you are creating.

1. Open the Services window by going to Window | Services or by clicking on the button that has a cloud on it in the toolbar on the right-hand side. If you are working offline, you may be asked to sign in. When you do so, you should see something similar to the following:

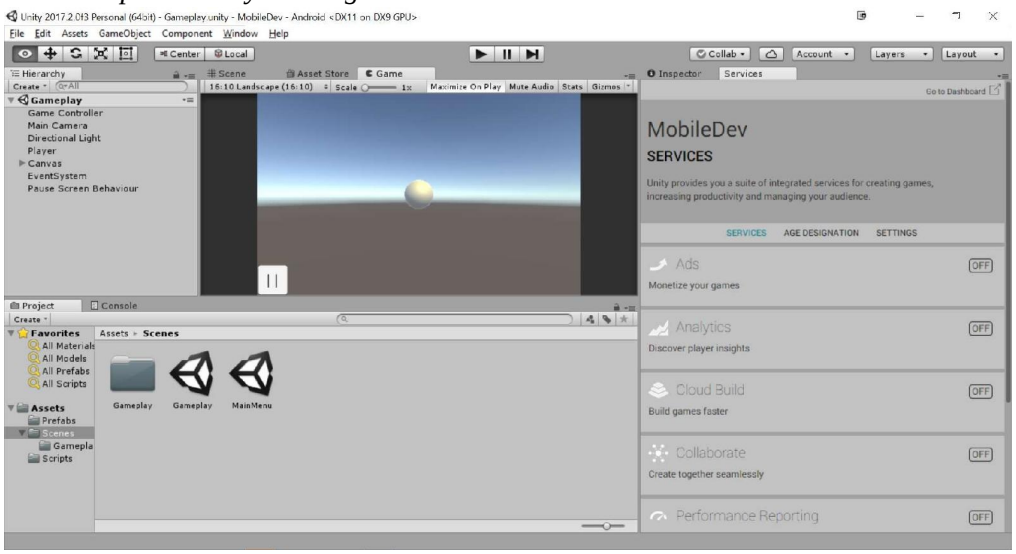


2. Assuming that you haven't worked with Unity Services before, you will need to create an Organization and Project Name. Click on the dropdown and select your username and then click on the Create button.

   The project name is automatically created according to the name of your project when you first created, but you can change this in the

Settings section of the Services window.

Unity automatically creates an organization using your account username; however, if you need to make another one, you can do so at https://id.unity.com/organizations.



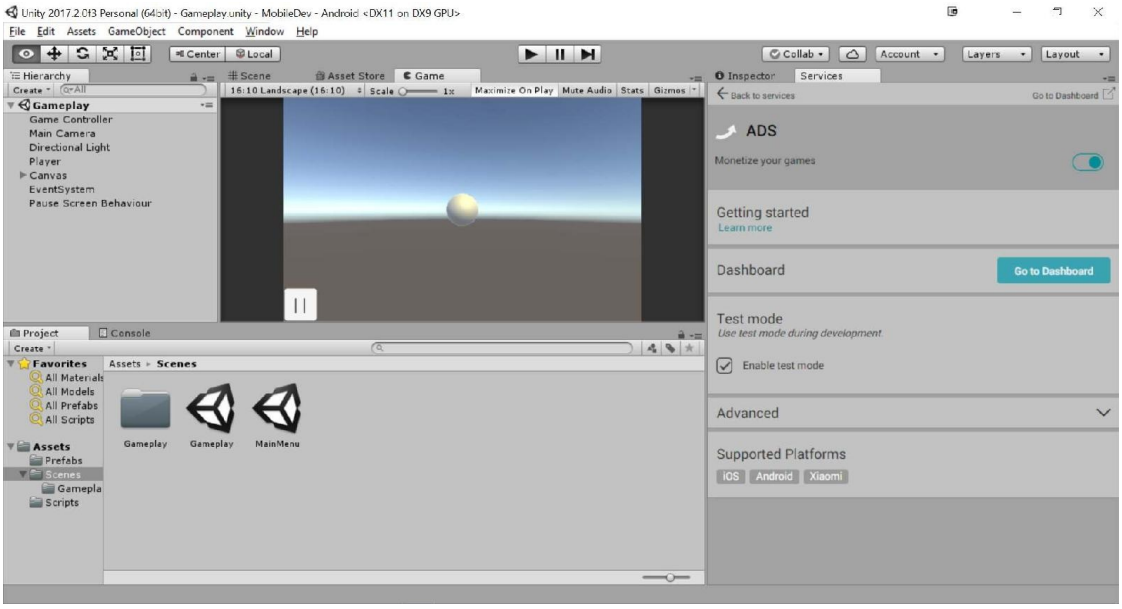
This should open up the Services window which contains a number of that we will be using over the course of this book.

3. At the top of the menu that pops up, you'll see a button called Ads. Go ahead and click on it to enter the menu. From there, click on the toggle on the top-right corner to turn Ads on. You'll then be asked questions about your game. If your game is not directed to children, go ahead and click on the Continue button, otherwise click on the check and then click on Continue.

*When you indicate whether your game is designed for children under the age of 13 years, ads will not be behaviorally targeted to users in your game. Behavioral targeting can yield higher **effective cost per thousand impressions (eCPMs)** by showing ads that are more relevant to your users, but its use is prohibited with users under the age of 13 due to **Children's Online Privacy Protection Rule Act (COPPA)** regulations. For more info on this, check out htt ps://forum.unity.com/threads/age-designation.326930/.*

4. Ads should be toggled on at this point. Then, click on the checkbox for

Enable test mode. This will ensure that ads displayed are just for testing:



It is against Unity Ad's terms of service to distribute live ads to beta testers. If they are to click on or install any of the advertised games, their activity will be monetized and the automated fraud system would flag the game for fraud and disable it. That's why we keep Test mode enabled until the game is ready to launch.

At this point, we have finished setup and can proceed to actually adding ads to our project.

Displaying a simple Ad

As mentioned previously, Unity Ads has two different types of ads that we can display: simple and rewarded. Simple ads are easy to use, hence the name, and allow users to have simple full-screen interstitial ads. This can be really useful for moving between levels or perhaps for when the player wants to restart the game. Let's see how we can implement that feature now.

1. To get started, it would be a good idea for us to have all of the Ad-related behavior to share a script, so we will create a new class called `UnityAdController` by going to the Project window, opening the `Assets/Scripts` folder, and selecting Create | C# Script.
2. Open up the file in the IDE of your choice, and use the following code:

```
using UnityEngine;

#if UNITY_ADS // Can only compile ad code on supported platforms

using UnityEngine.Advertisements; // Advertisement class

#endif

public class UnityAdController : MonoBehaviour
{

    public static void ShowAd()
    {

        #if UNITY_ADS

        if (Advertisement.IsReady())
        {
            Advertisement.Show();
        }

        #endif

    }
}
```

The preceding code does a number of things. We first state that we are using the `UnityEngine.Advertisments` namespace to get access to the `Advertisement` class.

We've also created a static method called `ShowAd`. We made this static so that we can access the function without actually having to create

an instance of this class in order to call this function. We then check whether an advertisement is ready, and if it is, we will then call the `Show()` function to display it on the screen.

Note that in all of the code where we make use of the `Advertisement` class, I've added in a `#if` statement and closed it with a `#endif`. This is a directive that will only compile if the defined symbol is defined. Basically, this code will only be compiled if Unity supports ads on it, so our project will still compile for both PC and mobile devices.

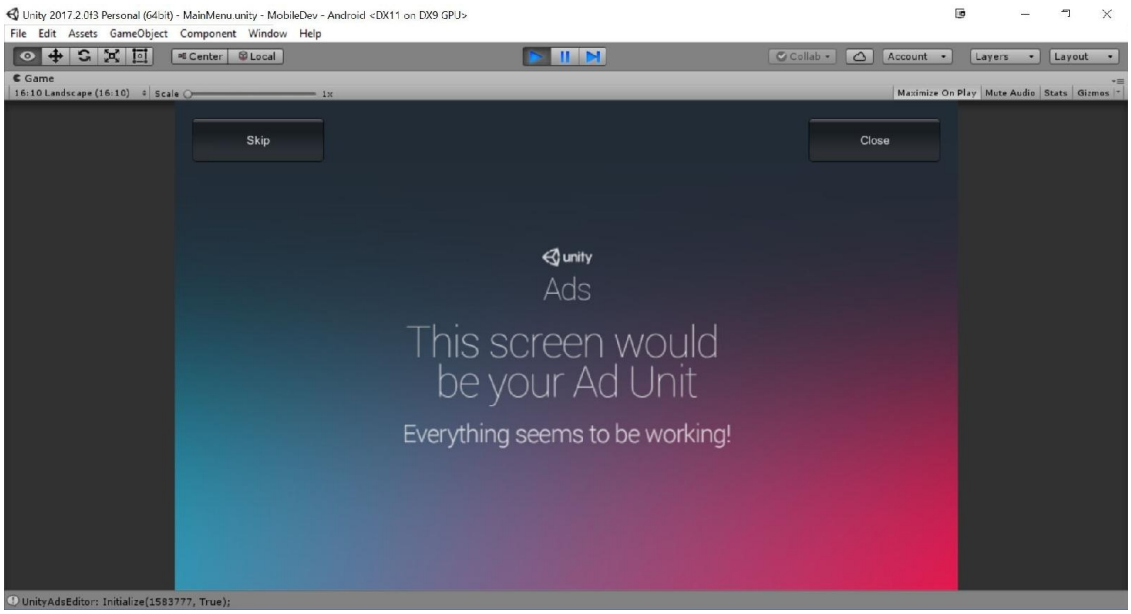
For more information on `#if`, `endif`, and other conditional directives, check out https://msdn.microsoft.com/en-us/library/ew2hz0yd.aspx.

3. Save your script and then open up the `MainMenuBehaviour` file and add the following highlighted code:

```
using UnityEngine;
using UnityEngine.SceneManagement; // LoadScene

public class MainMenuBehaviour : MonoBehaviour
{
    /// <summary>
    /// Will load a new scene upon being called
    /// </summary>
    /// <param name="levelName">The name of the level we want
    /// to go to</param>
    public void LoadLevel(string levelName)
    {
        SceneManager.LoadScene(levelName);

#if UNITY_ADS

        if (UnityAdController.showAds)
        {
            // Show an ad
            UnityAdController.ShowAd();
        }

#endif
    }
}
```

This will have an advertisement play each time we call the `LoadLevel` function, if it is supported. We also added in a new parameter with a default value. The nice thing about this is that we can optionally decide when we want to show an ad. For instance, we may want to make it so that when we restart the game we don't play an ad.

4. Now let's see this in action. Play the game, pause the game, and click on the Main Menu button:



As you can note in the preceding screenshot, the ad works correctly. This screen is what is shown when playing the game in the editor. It has buttons to allow us to test whether a player skips or watches a video in full. When we disable Test mode, we will then see live video ads.

If this does not work and/or show up, check the Player Settings menu you learned about previously and ensure that your current platform is set to Android or iOS.

Utilizing ad callback options

The code we wrote for the `LoadLevel` function works perfectly fine when we go to the main menu of the game; however, if we dive into the game itself from the main menu, the game will still be going on in the background with the ad blocking the player from playing the game.

When running your app on an actual mobile device, the Unity Player will pause while Unity Ads are shown. However, if you are testing in the Unity Editor, the game is not paused while the placeholder ads are shown. However, we can simulate that behavior ourselves using the `Advertisement.ShowOptions` class.

We will pause the game when an ad is shown and then resume once the ad is finished. To do so, perform the following steps:

1. Let's first open up the `UnityAdController` class and update it to the following:

```
using UnityEngine;

#if UNITY_ADS // Can only compile ad code on support platforms

using UnityEngine.Advertisements; // Advertisement

#endif

public class UnityAdController : MonoBehaviour
{

    public static void ShowAd()
    {

        #if UNITY_ADS

        // Set options for our advertisement
        ShowOptions options = new ShowOptions();
        options.resultCallback = Unpause;

        if (Advertisement.IsReady())
        {
            Advertisement.Show(options);
        }

        // Pause game while ad is shown
        PauseScreenBehaviour.paused = true;
        Time.timeScale = 0f;

        #endif

    }
```

```
    #if UNITY_ADS

    public static void Unpause(ShowResult result)
    {
        // Unpause when ad is over
        PauseScreenBehaviour.paused = false;
        Time.timeScale = 1f;
    }

    #endif

}
```

The preceding code shows us the `ShowOptions` class for the first time. This class contains options that can be passed to `Advertisements.Show` to modify the behavior of the advertisement we want to play, the most important being `resultCallback`. This basically says that whenever we close, skip, or fail playing the advertisement, we want to call the `Unpause` function. Now, we will be diving into more detail on this function in a later section, but in the meantime, we can make use of this to resume the game.

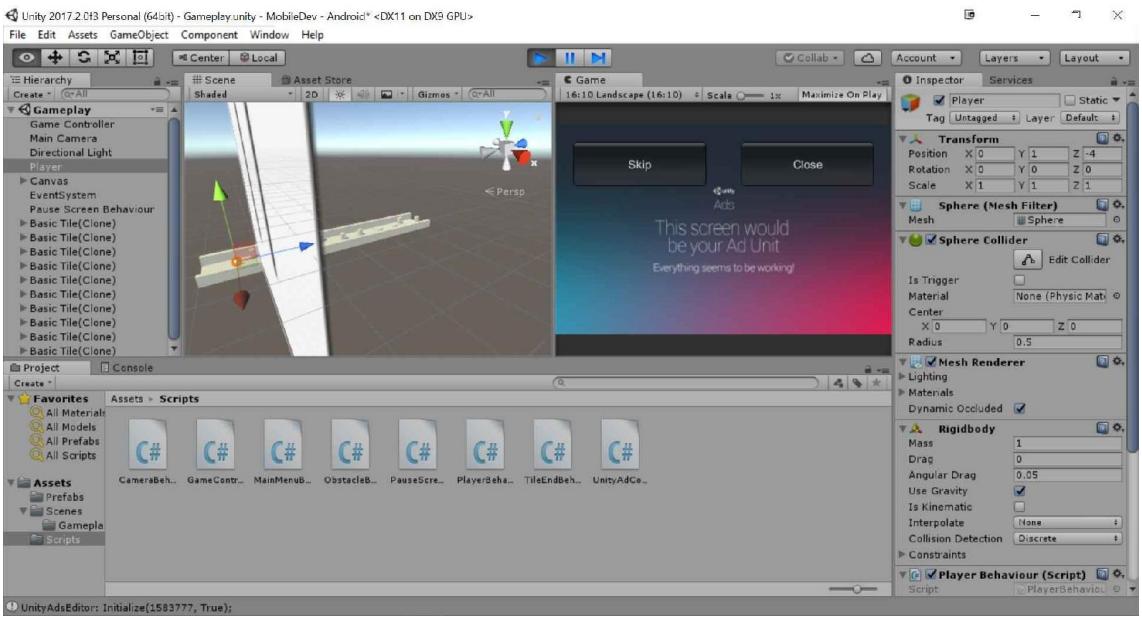
For more information on the `ShowOptions` class, check out https://docs.unity3d.com/ScriptReference/Advertisements.ShowOptions.html.

2. To start off, we will make it so that the `PauseMenuBehaviour` doesn't override this new change. So, we will replace the `Start()` function with the following:

```
void Start()
    {
        paused = false;

        #if !UNITY_ADS // If not using ads, just start the game

        SetPauseMenu(false);

        #endif
    }
```

Performing the preceding snippet is important because otherwise the game will immediately be turned off when the level loads in the Start function after we tell the game to pause, which is called after the level loads. This is needed for the PC version of the game, as there is nothing else for unpausing the static value.

3. Save our scripts and start the game up again:



With that, when we transition from the main menu to the game, we will pause the game until we are ready to jump in.

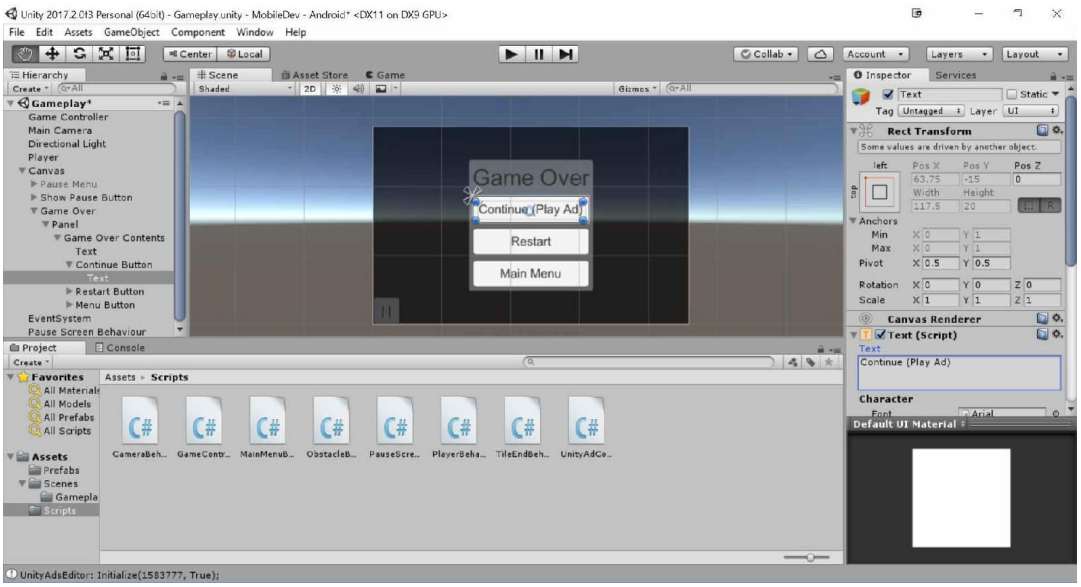
Opt-in advertisements with rewards

According to AdColony, the most recommended form of mobile game ads from 58% of mobile developers is the rewarded video ad. By that, we're referring to making ads an opt-in experience where players choose to see an ad and they receive some kind of bonus in return. That way, users feel it's a choice for them to see the ad and they feel compelled to see it because they will get something out of it.

Rewarded ad placements typically yield higher **effective Cost Per 1000 Impressions** (**eCPMs**) since they offer more engagement from users by allowing them to opt-in before watching an ad in exchange for some in-game reward. In our game, we could add the choice of restarting the game or seeing an ad to continue the game.

If you're interested in learning more about why reward ads are recommended, check out https://www.adcolony.com/blog/2016/04/26/the-top-ads-recommended-by-mobile-game-developers/.

1. So, let's create a Game Over menu by first going to the Hierarchy tab and expending the Canvas if not done so already. Then, select the Pause Menu object and duplicate it by pressing *Ctrl + D*. Rename this new object Game Over and then toggle it on so that we can see it. To make it easier to see, feel free to toggle the 2D mode we used previously when creating the UI elements of our game.
2. Next, expand the `Game Over` object and the Panel child, then change the Pause Menu Contents object's name to `Game Over Contents` and change the child Text object's Text component to say `Game Over` instead.
3. Now, change the Resume button to say `Continue (Play Ad)` and change the button object's name to `Continue Button`:



4. We'll first need to update the `ObstacleBehaviour` script to handle it; add the following highlighted code:

```
using UnityEngine;
using UnityEngine.SceneManagement; // LoadScene
using UnityEngine.UI; // Button

public class ObstacleBehaviour : MonoBehaviour
{

    [Tooltip("How long to wait before restarting the game")]
    public float waitTime = 2.0f;

    void OnCollisionEnter(Collision collision)
    {
        // First check if we collided with the player
        if (collision.gameObject.GetComponent<PlayerBehaviour>())
        {
            // Destroy (Hide) the player
            collision.gameObject.SetActive(false);

            player = collision.gameObject;

            // Call the function ResetGame after waitTime
            // has passed
            Invoke("ResetGame", waitTime);
        }
    }

    /// <summary>
    /// Will restart the currently loaded level
    /// </summary>

    void ResetGame()
    {
        //Bring up restart menu
        var go = GetGameOverMenu();
        go.SetActive(true);
```

```
        // Get our continue button
        var buttons = go.transform.GetComponentsInChildren<Button>();
        UnityEngine.UI.Button continueButton = null;

        foreach (var button in buttons)
        {
            if (button.gameObject.name == "Continue Button")
            {
                continueButton = button;
                break;
            }
        }
        if (continueButton)
        {
            #if UNITY_ADS
            // If player clicks on button we want to play ad and
            // then continue

    continueButton.onClick.AddListener(UnityAdController.ShowRewardAd);
            UnityAdController.obstacle = this;
            #else
            // If can't play an ad, no need for continue button
            continueButton.gameObject.SetActive(false);
            #endif
        }
    }

    private GameObject player;

    /// <summary>
    /// Handles resetting the game if needed
    /// </summary>
    public void Continue()
    {
        var go = GetGameOverMenu();
        go.SetActive(false);
        player.SetActive(true);

        // Explode this as well (So if we respawn player can continue)
        PlayerTouch();
    }

    /// <summary>
    /// Retrieves the Game Over menu game object
    /// </summary>
    /// <returns>The Game Over menu object</returns>
    GameObject GetGameOverMenu()
    {
        return GameObject.Find("Canvas").transform.Find("Game
        Over").gameObject;
    }


    public GameObject explosion;

    /// <summary>
    /// If the object is tapped, we spawn an explosion and
    /// destroy this object
    /// </summary>
    void PlayerTouch()
    {
        if (explosion != null)
        {
            var particles = Instantiate(explosion, transform.position,
```

```
            Quaternion.identity);
            Destroy(particles, 1.0f);
        }

        Destroy(this.gameObject);
    }
}
```

In this instance, we are removing the destruction of our player and are instead hiding them so that we can re-enable it later if we would like to. We also destroy what the player hit. So, if we do restart the game, then the player will be able to start off from right where he initially began. With that in mind, we also created a `Continue` function that will set up the game to be continued if we need to do so.

5. Open up the `UnityAdController` script and update it by adding the following functions:

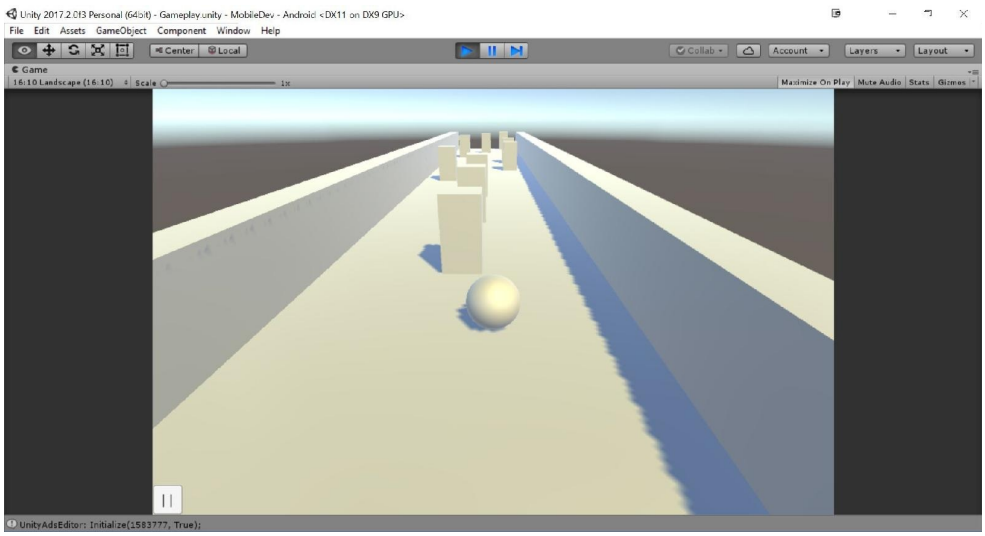
```
public static void ShowRewardAd()
    {
        #if UNITY_ADS

        if (Advertisement.IsReady())
        {
            // Pause game while ad is shown
            PauseScreenBehaviour.paused = true;
            Time.timeScale = 0f;

            var options = new ShowOptions { resultCallback =
            HandleShowResult };
            Advertisement.Show(options);
        }

        #endif
    }

    // For holding the obstacle for continuing the game
    public static ObstacleBehaviour obstacle;

    private static void HandleShowResult(ShowResult result)
    {
        #if UNITY_ADS

        switch (result)
        {
            case ShowResult.Finished:
                // Successfully shown, can continue game
                obstacle.Continue();
                break;
            case ShowResult.Skipped:
                Debug.Log("Ad skipped, do nothing");
                break;
            case ShowResult.Failed:
                Debug.LogError("Ad failed to show, do nothing");
                break;
        }
```

```
        #endif
        // Unpause when ad is over or when called
        PauseScreenBehaviour.paused = false;
        Time.timeScale = 1f;

    }
```

6. Save your scripts.
7. Click on the `Game Over` object and disable it, save our scene, and then dive into the game.

At this point, when we die in the game, we'll be given a Game Over screen:

If we click on Continue (Play Ad), we will have an ad play. If the player skips it, nothing will happen, but if they watch all the way through it should bring us back into the game as if nothing happened:



With that, our ad system is working correctly.

Adding in a cooldown

Ads are great for developers; however, according to Unity's Monetization FAQs, each user is only able to view 25 ads per day. With that in mind, we will likely want to make it so that players can only trigger ads every once in a while. This also has the benefit of making players want to come back to our game after a period of time.

For more information on Unity's Monetization FAQs, check out https://unityads.unity3d.com/help/faq/monetization.

We will now program it so that our Continue option will only work once in awhile, perhaps with a short delay that we can easily customize if we'd like:

1. To get started, let's dive into our `UnityAdController` script. We need to add a new variable to it, as you can see in the following highlighted code:

```
using System; // DateTime
using UnityEngine;

#if UNITY_ADS // Can only compile ad code on support platforms

using UnityEngine.Advertisements; // Advertisement

#endif

public class UnityAdController : MonoBehaviour
{

    // Nullable type
    public static DateTime? nextRewardTime = null;


    public static void ShowAd()
    {
        // Rest of function below....
```

The `nextRewardTime` variable is of the `DateTime` type, which we haven't talked about previously. Basically, it's a structure that represents a point in time where we can compare it to other times and it's built into the .NET framework. We'll use this to store the time (if any) that needs to pass before the player is able to play another ad. Note that `DateTime` is part of the System namespace. That is why we added the

`using System;` line in the preceding code as well.

For more information on the `DateTime` class check out: https://msdn.microsoft.com/en-us/library/system.datetime(v=vs.110).aspx.

You may notice the `?` next to the type of this variable. When we do this, we create what's called a nullable type. The advantage of using them is that they can be null in addition to having normal values. We do this so that we don't have to fill in a default value just for the sake of having one.

For more information on nullable types, check out https://www.tutorialspoint.com/csharp/csharp_nullables.htm.

2. We also need to update the `ShowRewardAd()` function as follows:

```
public static void ShowRewardAd()
{
    #if UNITY_ADS

    nextRewardTime = DateTime.Now.AddSeconds(15);

    if (Advertisement.IsReady())
    {
        // Pause game while ad is shown
        PauseMenuBehaviour.paused = true;
        Time.timeScale = 0f;

        var options = new ShowOptions { resultCallback =
        HandleShowResult };
        Advertisement.Show(options);
    }

    #endif
}
```

Now when we show a reward ad, we set `nextRewardTime` to 15 seconds from when the function is called. Of course, we can just as easily set this to minutes or hours using the `AddMinutes` and `AddHours` functions.

3. Save your script and then open up the `ObstacleBehaviour` script. To start off with, we'll use two new types, so we will need to add the following new using statements to our script:

```
using System; // DateTime
```

```
using System.Collections; // IEnumerator
```

4. Afterward, we will need to modify the bottom part of the `RestartGame()` function to have the following bolded changes:

```
// Rest of RestartGame above...

        if (continueButton)
        {
            #if UNITY_ADS
            // If player clicks on button we want to play ad and
            // then continue
            StartCoroutine(ShowContinue(continueButton));
            #else
            // If can't play an ad, no need for continue button
            continueButton.gameObject.SetActive(false);
            #endif
        }
}
```

Now, instead of just adding a listener to this button, we have replaced it with calling a `StartCoroutine` function, which takes in a function that we haven't written yet, but will get to in a second. I think it's probably a good idea to talk a little bit about coroutines first.

A coroutine is like a function that has the ability to pause execution and continue where it left off after a period of time. By default, a coroutine is resumed on the frame after we start to `yield`, but it is also possible to introduce a time delay using the `WaitForSeconds` function for how long you want to wait before it's called again.

5. In there, use the following script for the `ShowContinue` function:

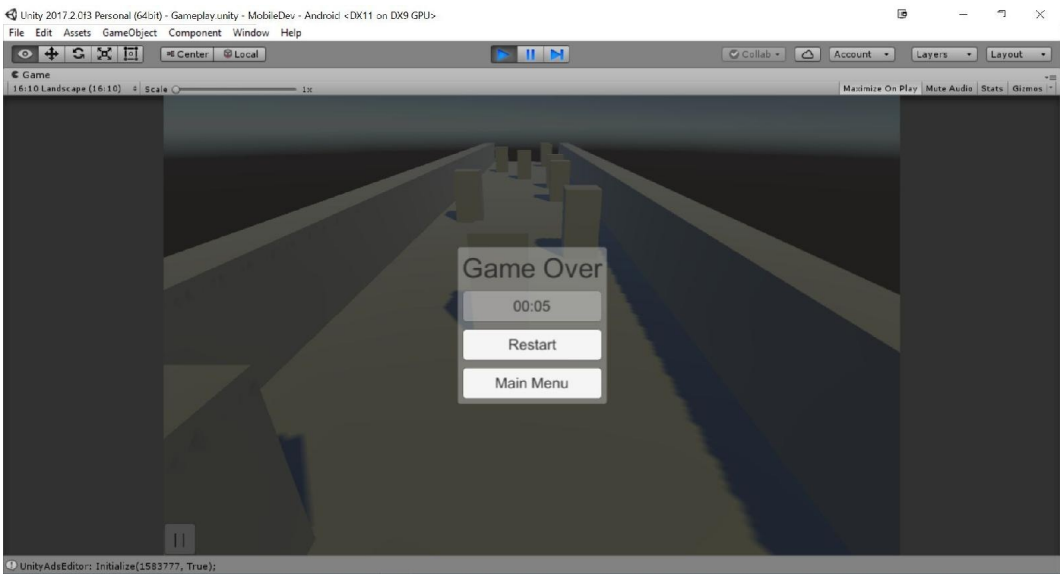
```
public IEnumerator ShowContinue(UnityEngine.UI.Button contButton)
{
    while (true)
    {
        var btnText = contButton.GetComponentInChildren<Text>();

        // Check if we haven't reached the next reward time yet
        // (if one exists)
        if (UnityAdController.nextRewardTime.HasValue &&
            (DateTime.Now <
                UnityAdController.nextRewardTime.Value))
        {
            // Unable to click on the button
            contButton.interactable = false;

            // Get the time remaining until we get to the next
            // reward time
            TimeSpan remaining = UnityAdController.nextRewardTime.Value
                            - DateTime.Now;
```

```
            // Get the time left in the following format 99:99
            var countdownText = string.Format("{0:D2}:{1:D2}",
                                remaining.Minutes,
                                remaining.Seconds);

            // Set our button's text to reflect the new time
            btnText.text = countdownText;

            // Come back after 1 second and check again
            yield return new WaitForSeconds(1f);
        }
        else
        {
            // It's valid to click the button now
            contButton.interactable = true;

            // If player clicks on button we want to play ad and
            // then continue
        contButton.onClick.AddListener(UnityAdController.ShowRewardAd);
            UnityAdController.obstacle = this;

            // Change text to its original version
            btnText.text = "Continue (Play Ad)";

            // We can now leave the coroutine
            break;
        }
    }

}
```

This coroutine will do a number of things, starting off by entering a `while (true)` loop. Now, usually, this is a very bad thing, as it would cause an infinite loop, but we break out of the loop if we have no reward time set or if we've passed the time set in the `nextRewardTime` variable. If not, we will figure out how much time is left before that time has passed and will change the button's text to display it. We then use the `WaitForSeconds` function to pause execution and come back after 1 second has passed.

If you're interested in learning more about the behind the scenes aspects of how coroutines work, Cyrill Nadezhdin has written a neat article on it at http://blog.nadezhdin.org//articles/Unity-coroutines-demystified.

6. Save all of our scripts and dive back into Unity and play the game:



Upon restarting the game once, you'll see that if we try to do so again we are brought to a delay screen. After the time gets down to 0, the player will then be able to continue once again.

Summary

With that, we've gotten a good foundation on how to add ads to our game. Hopefully, you can see how easy it is to implement and can think of new ways to engage players to have the best experience possible. Over the course of this chapter, we discovered how to set up Unity Ads. We then saw how we could create simple ads and then learned how to react to the player's actions with ad callback options. Afterward, we saw how we could add in rewards for players using opt-in advertisements to the game and added a cooldown to the system to make the game less annoying for players.

While this is a valid way to monetize our games, we will dive into the other most popular form of in-game monetization in the next chapter: in app purchases.

Implementing In-App Purchases

As mentioned in the preceding chapter, there are many options out there when it comes to selling your game on a mobile platform. If you decide to go free-to-play, in addition to showing ads, there is also the ability to sell people additional content and/or advantages through the use of selling in-app purchases. This can be a way to engage users of your game to convert themselves from a free player to a paid one.

Typically, these can be options such as removing ads or offering themes to players, but you can also do things such as unlock new levels and add additional content that people addicted to your game will be clamoring to give you more of their time. Alternatively, you can also think of your IAPs as items that players will want to buy in order to enhance their gameplay experience, such as power-ups and upgrades.

For more tips and tricks on improving your freemium strategy, I suggest that you check out an article by Pepe Agell at https://www.chartboost.com/blog/2015/04/inapp-purchases-for-indie-mobile-games-freemium-strategy/.

Chapter overview

In this chapter, we will integrate Unity's **In-App Purchase** (**IAP**) system into our project and take a look at how to create an IAP that is for consumable content as well as permanent unlocks.

Your objectives

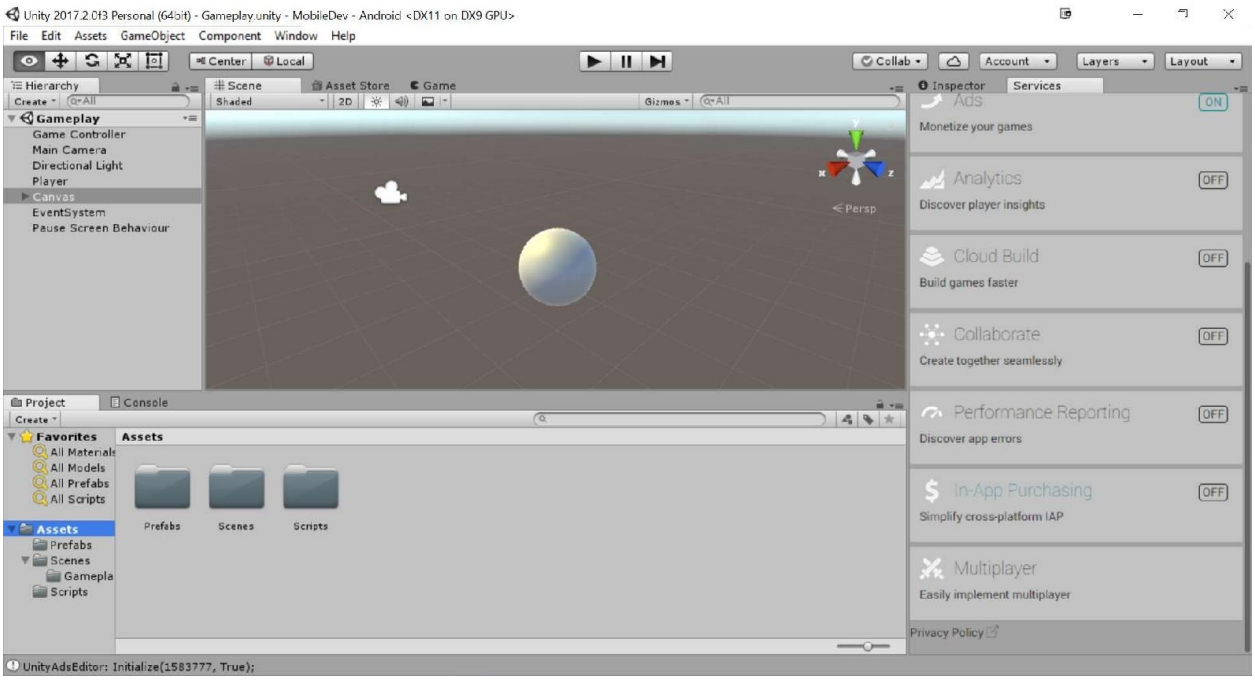
This chapter will be split into a number of topics. It will contain a simple step-by-step process from beginning to end. The following is the outline of our tasks:

- Setting up Unity IAP
- Creating a first purchase
- Adding button to restore purchases
- Configuring purchases for the stores of your choice

Setting up Unity IAP

Unity IAP is a service that allows us to sell a variety of different items to players within our game projects and is currently supported by the iOS App Store, Mac App Store, Google Play, Windows Store, Amazon Appstore, and more, by default. So, using this, we can easily sell our items in many different places. We have already set up Unity Services in the preceding chapter, so this will be a lot easier to get going. Perform the following steps to add Unity IAP:

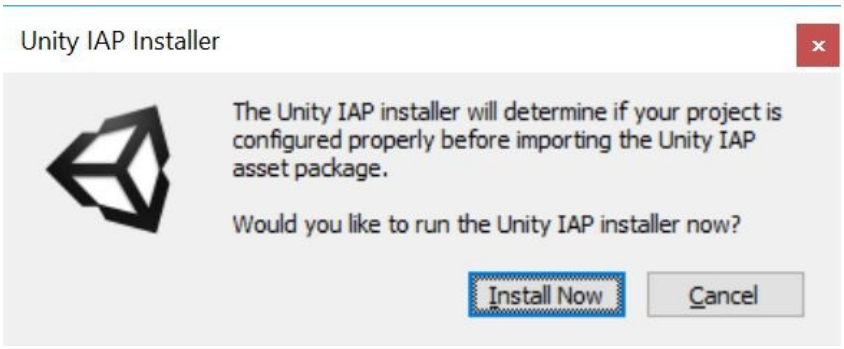
1. Open the Services window by going to Window | Services or by clicking on the Cloud button in the toolbar. Assuming that you are following along from the preceding chapter, we should already have services set up. If not, check out the Unity Ads Setup section in Chapter 5, *Advertising using Unity Ads*, for an explanation on how to do so.
2. From the Services menu, scroll down to the In-App Purchasing item. You will note that it's currently off:



3. Click on the In-App Purchasing button to open it up and then click on the Enable button.
4. From there, you'll see the button change to say Import instead. Unity IAP

requires us to import a package for us to create IAPs, so let's go ahead and click on the Import button and wait for it to finish.

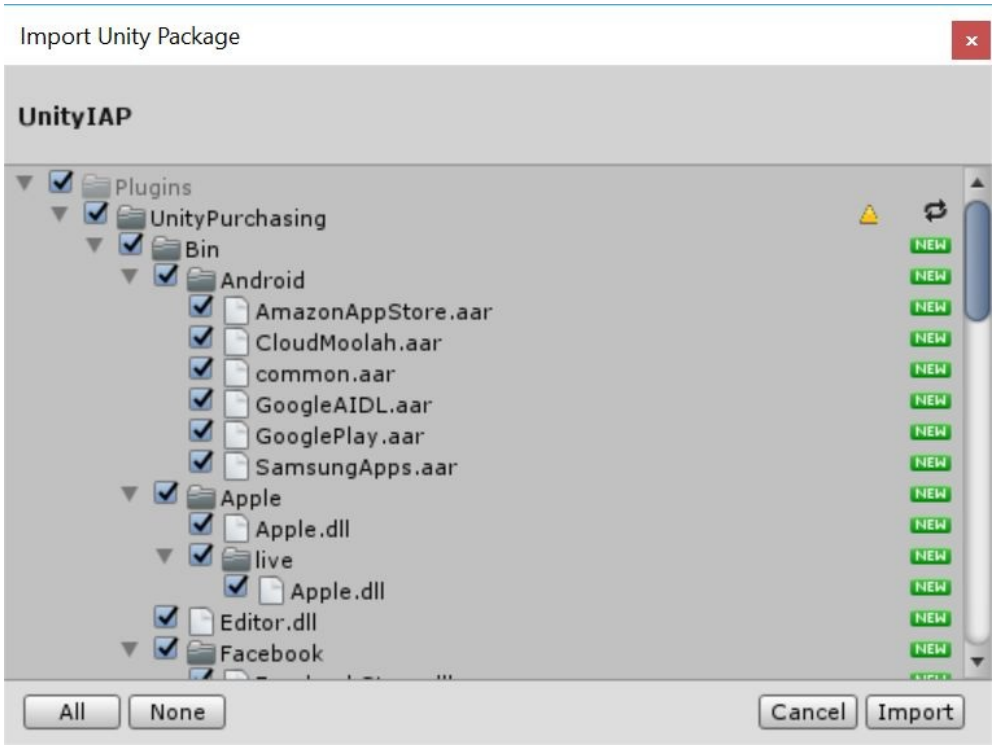
5. You'll note that when it finishes, you'll have a new folder named `Plugins` created and a popup saying that it needs to determine whether our project is configured properly and asking whether we want to install it:



6. Go ahead and click on the Install Now function and wait for it to finish.

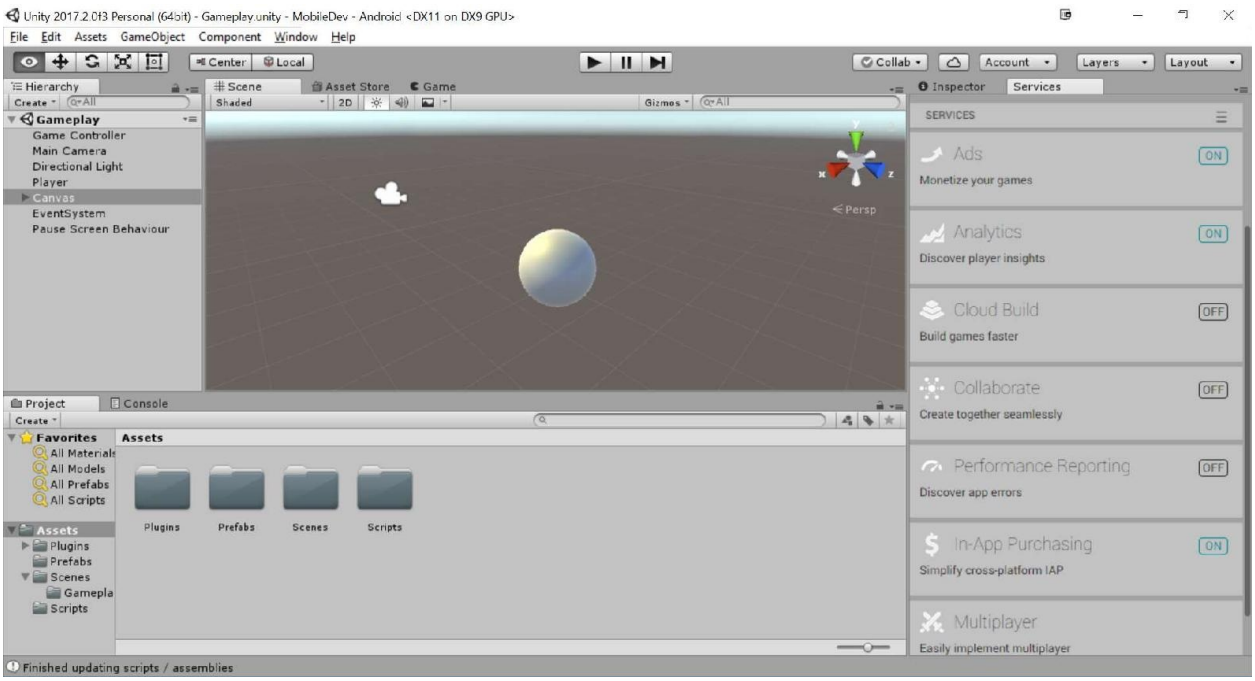
If, for some reason, you encounter any errors at this point in the Console window, try to first close and then open the Services window again and check whether Unity IAP is enabled. If that doesn't work, disconnect and reconnect it to the internet and then sign back into Unity Services and then re-enable Unity IAP.

If all goes well, you should see an Import Unity Package window coming up:



7. Click on the Import button and wait for it to be inserted into our project. It may say that an API Update is required. This is likely because the version of Unity that you're using is newer than the last update of the Unity IAP API. Go ahead and click on the I Made a Backup. Go ahead! button and wait for it to finish.

8. Once it's done, scroll up to the top of the Services tab and click on the Back to Services button, as follows:



Once you come back, you should note that both Analytics and In-App Purchasing are now set to ON.

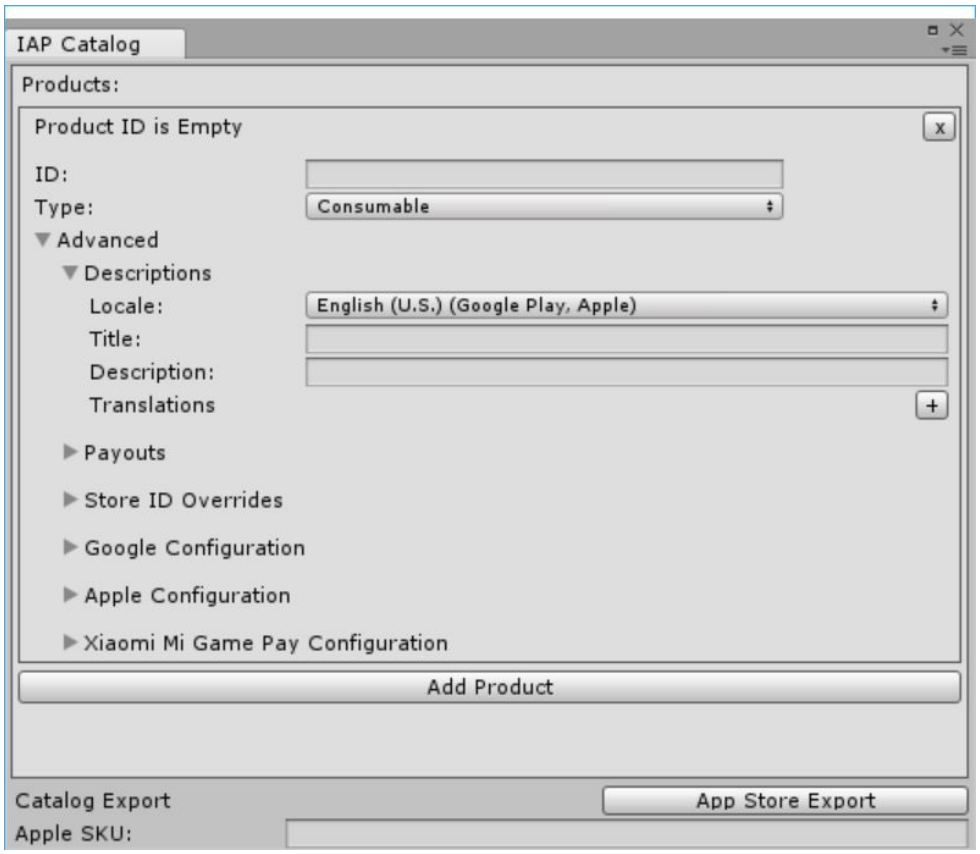
The IAP package is created externally from the main engine itself because the code is meant to be extremely flexible and can be updated to fit any policies that are needed and then we can just update the package instead of having to update it to the latest version of Unity, which can be very important when working on a large project.

Creating our first purchase

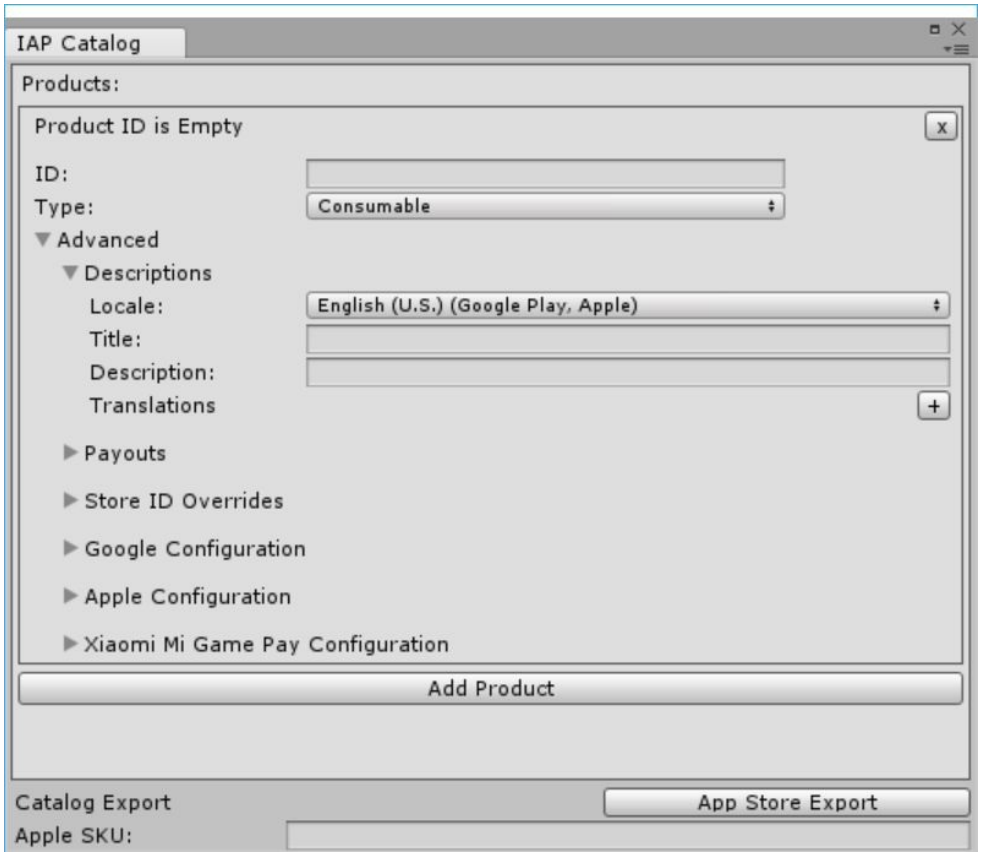
To make our first in-app purchase, we will make use of a feature of Unity that was just added to our project, Codeless IAP. It is called Codeless IAP because you do not need to write any code for the actual IAP transaction, just the script that defines what users get if they make the purchase. It's by far the easiest way to integrate in-app purchases in Unity games and a great way to start trying IAPs in our project.

One of the most common in-app purchases is the ability to disable advertisements in mobile games. Let's add that functionality now by creating a button that when clicked will do just that:

1. Let's first open up our Main Menu level by going to the Project window, opening the `Assets/Scenes` folder, and then double-clicking on the `MainMenu` file.
2. From there, let's click on the 2D button to go into 2D mode since we'll be working with UI.
3. We will first need to have something to sell; to do that, we will use the IAP Catalog, which we can access by going to Window | Unity IAP | IAP Catalog:



4. Now, the first thing we'll need to do is create an ID for our product, which is how we identify our product in different app stores. In our case, let's go with `removeAds`. Then, under Type, change it to Non Consumable:

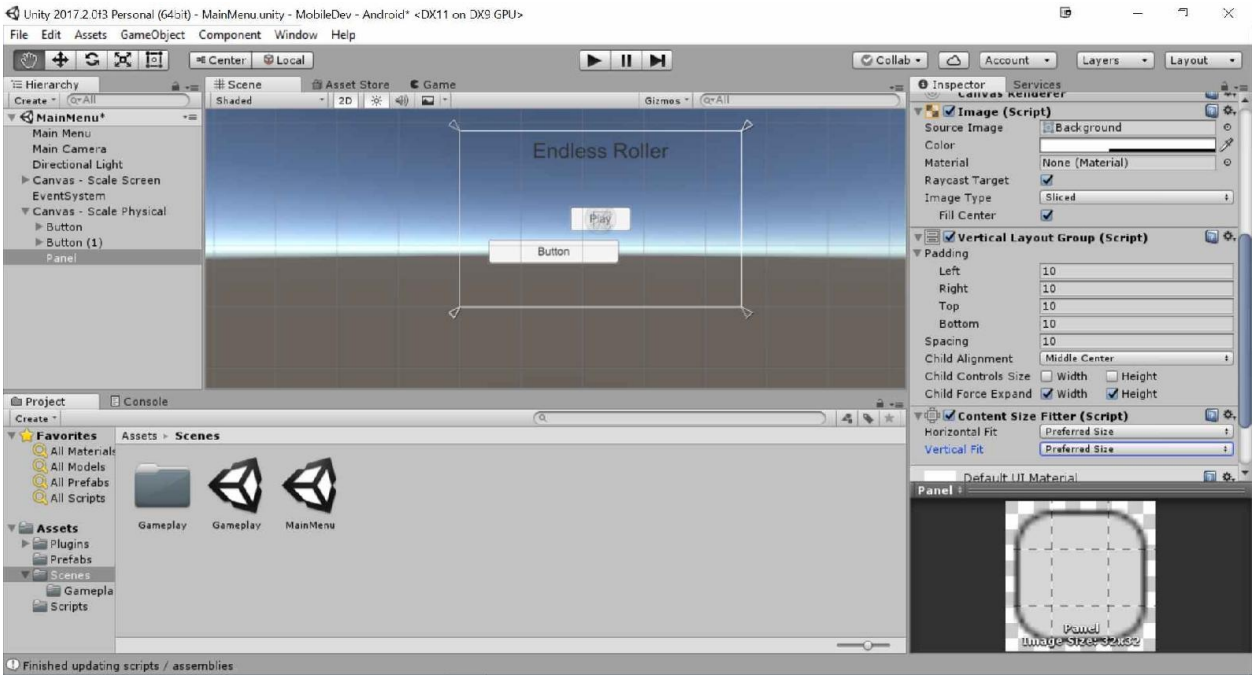


By non-consumable, we mean that the players only need to buy this once, and the game will keep that in mind for later purposes. The others are consumable, meaning that they are used for things that can be bought over and over again, such as special power ups and subscriptions. These give access to some kind of content for a period of time, possibly recurring until a user cancels them.

5. Next, we can close out of the IAP Catalog by clicking on X in the top-right corner of the window.

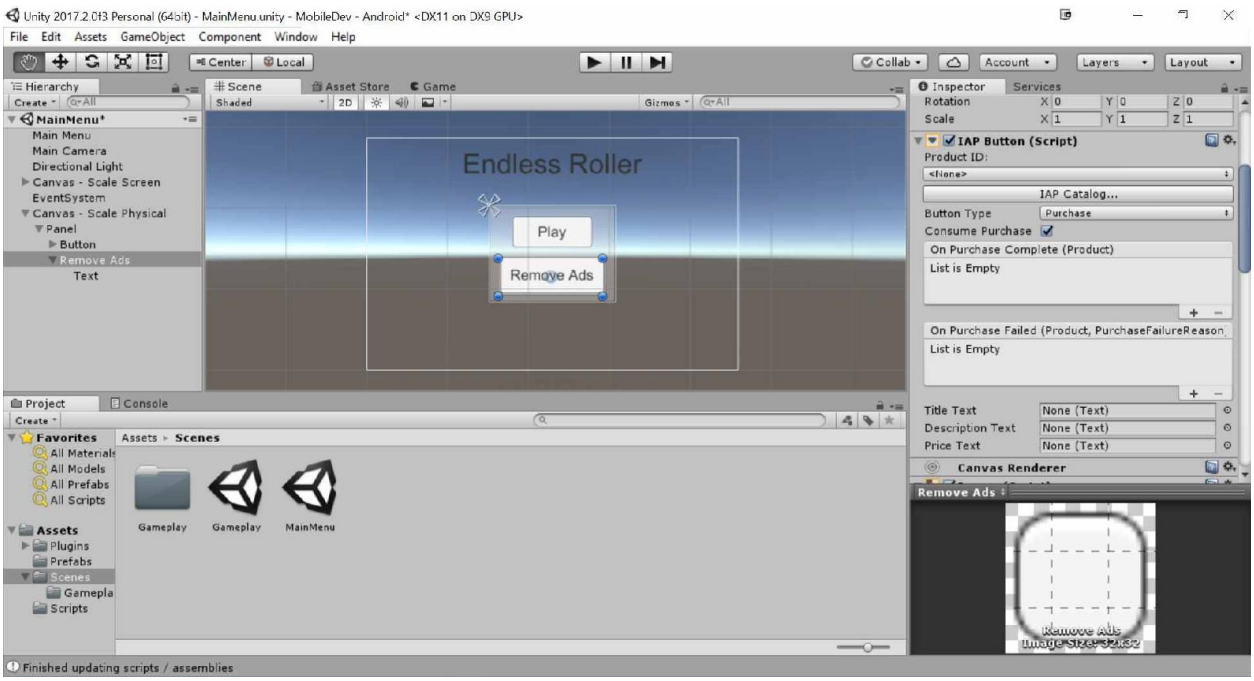
6. Select the Canvas - Scale Physical object in the Hierarchy window. From there, select Window | Unity IAP | Create IAP Button, and you should see a new button created in our scene.
7. Create a Panel object and have it fill the entire screen as we did before. Add a Vertical Layout Group (Script) component to it. From there, change the Child Alignment to Middle Center and set all of the Padding and Spacing to `10`.
8. Then, add a Content Size Fitter component and set the Vertical Fit and

Horizontal Fit field to `Preferred Size`:



9. Rename the newly added button to `Remove Ads` and then add a Horiztonal Layout Group component to it with all of the Padding set to `10` and check Width and Height to the Child Controls Size property. Also, add a Content Size Fitter component to it with both Fits set to Preferred Size. Then, in the child Text object's Text component, change the text property to show `Remove Ads` instead.

10. Finally, drag and drop the two buttons onto the Panel with the Play button first and the Remove Ads button below it, as follows:



11. Next, with the Remove Ads object selected, move to the Inspector tab and scroll down to the IAP Button component. Under Product ID:, select the dropdown and select `removeAds`. You'll note that the `IAP Button` class has an On Purchase Complete function that works similar to the On Click that we've used with Buttons in the past. With that in mind, we will need to create a function that will use this.

12. Go into the `UnityAdController` script and add the following variable:

```
public class UnityAdController : MonoBehaviour
{
    // If we should show ads or not
    public static bool showAds = true;

    // Nullable type
    public static DateTime? nextRewardTime = null;

    // Rest of UnityAdController...
```

    We will use this variable to check whether we should show ads or not.

13. Next, we will need to open up the `MainMenuBehaviour` script and replace it with the following:

```
using UnityEngine;
using UnityEngine.SceneManagement; // LoadScene
```

```
public class MainMenuBehaviour : MonoBehaviour
{
    /// <summary>
    /// Will load a new scene upon being called
    /// </summary>
    /// <param name="levelName">The name of the level we want
    /// to go to</param>
    public void LoadLevel(string levelName)
    {
        SceneManager.LoadScene(levelName);

        #if UNITY_ADS

        if (UnityAdController.showAds)
        {
            // Show an ad
            UnityAdController.ShowAd();
        }

        #endif
    }

    public void DisableAds()
    {
        UnityAdController.showAds = false;

        // Used to store that we shouldn't show ads
        PlayerPrefs.SetInt("Show Ads", 0);
    }

    virtual protected void Start()
    {
        // Initialize the showAds variable
        UnityAdController.showAds = (PlayerPrefs.GetInt("Show Ads", 1) == 1);
    }
}
```

14. Note that I made this function virtual, which means that inherited classes can also use this. With that in mind, we will also need to update Start function of the `PauseMenuBehaviour` to the following:

```
protected override void Start()
    {
        // Initalize Ads if needed
        base.Start();

        paused = false;

        // If no ads at all, just unpause
        #if !UNITY_ADS
            SetPauseMenu(false);
        #else

        // If we support ads but they're removed, unpause as well
        if (!UnityAdController.showAds)
        {
            SetPauseMenu(false);
        }

        #endif
```

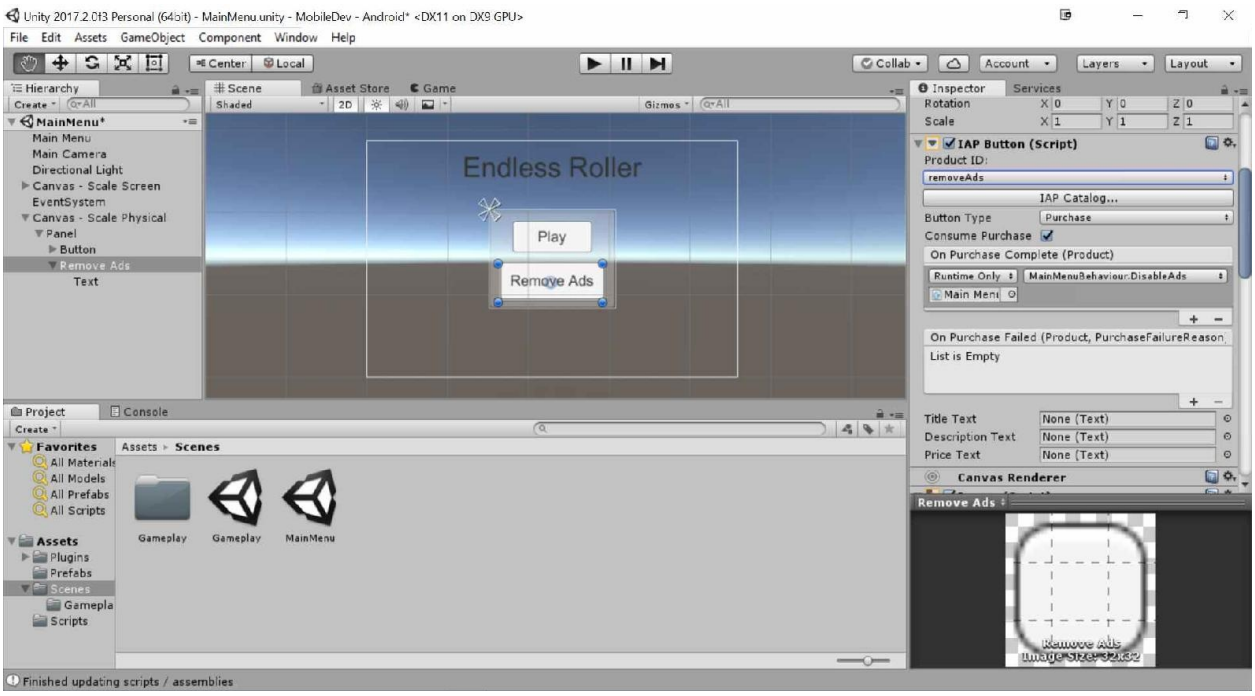
```
    }
```

The override keyword makes it so that it will replace the default behavior of Start. However, when we call `base.Start()` we are ensuring that the preceding content from `MainMenuBehaviour` will be called--in this case, we ensure that the `UnityAdController` has the correct value set.

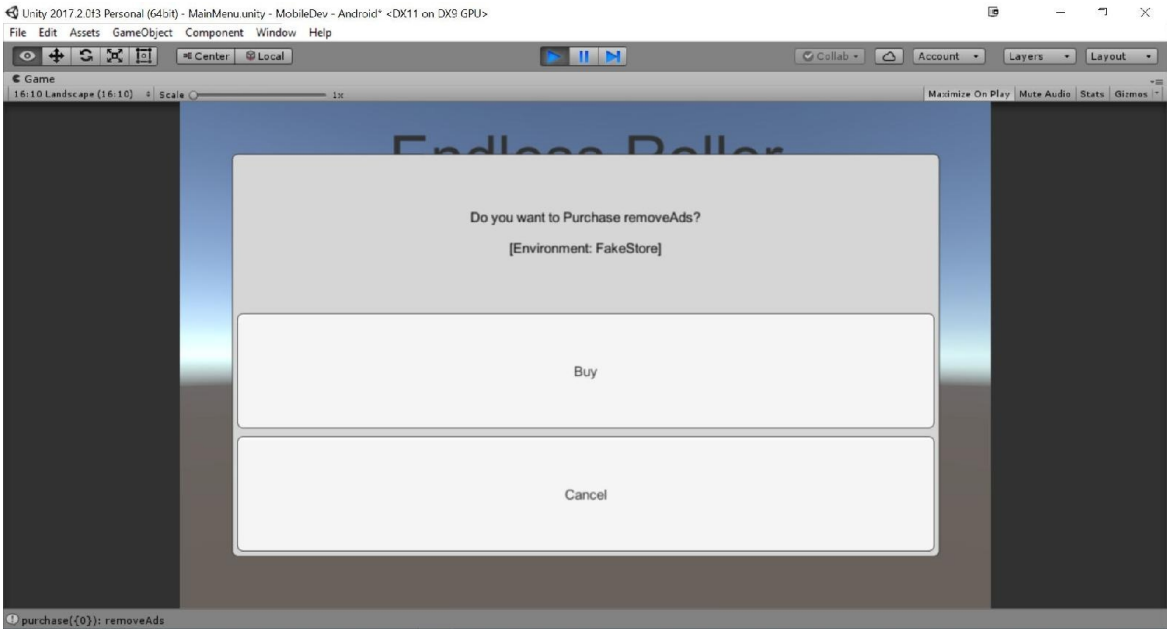
15. Finally, we will need to adjust the `ObstacleBehaviour` script to handle not playing ads as well. Update the `ShowContinue` function to use the following:

```
// Other code above...

                // Come back after 1 second and check again
                yield return new WaitForSeconds(1f);
            }
            else if (!UnityAdController.showAds)
            {
                // It's valid to click the button now
                contButton.interactable = true;

                // If player clicks on button we want to just continue
                contButton.onClick.AddListener(Continue);

                UnityAdController.obstacle = this;

                // Change text to allow continue
                btnText.text = "Free Continue";

                // We can now leave the coroutine
                break;
            }
            else
            {
                // It's valid to click the button now
                contButton.interactable = true;

                // More code below...
```

16. Save your script and dive into Unity.
17. From the Hierarchy window, select the Remove Ads button. Go into the Inspector tab and then scroll down to IAP Button (Script). Go ahead and click on the plus button underneath the On Purchase Complete option and then add the `Main Menu` object in the little box on the bottom of the side below the Runtime Only dropdown and then select Main Menu Behaviour | DisableAds from the dropdown to the right:

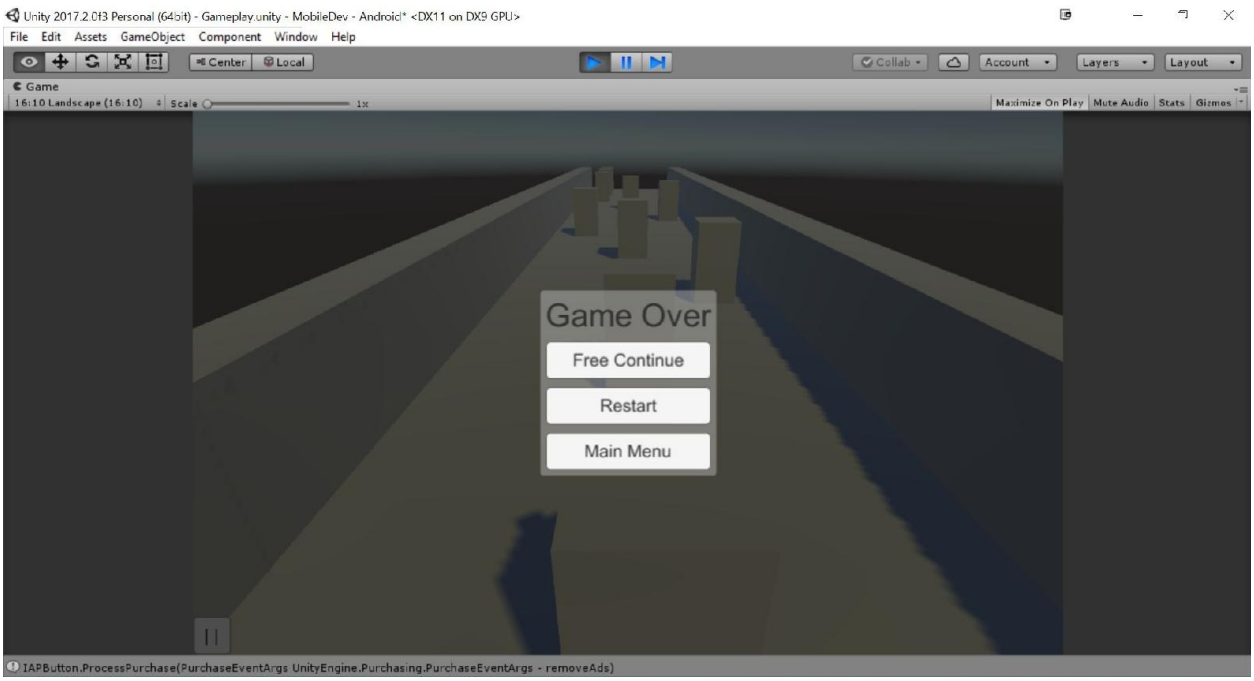


18. Now, save our scene and start the game:



Now, if we click on the Remove Ads button, it will ask whether we want to make the purchase. If we do, it will then make it so when we go into the game, there are no ads.

Likewise, now when we die, it will display a Free Continue button:



With that, we now have the Unity end of creating a simple purchase completed.

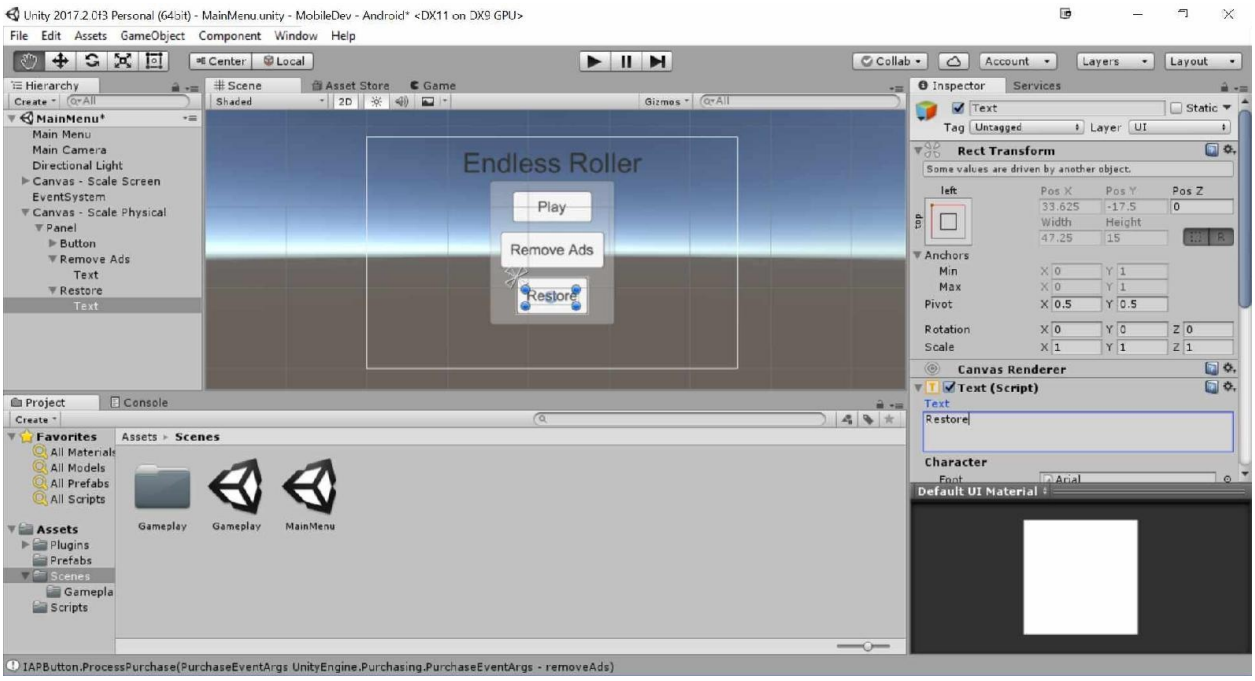
If you're interested in learning more about Codeless IAP, check out https://docs.unity3d.com/Manual/UnityIAPCodelessIAP.html.

If you'd like to dive into more customizable forms of IAPs, you may access the library directly. Information on that can be found at https://unity3d.com/learn/tutorials/topics/ads-analytics/integrating-unity-iap-your-game.

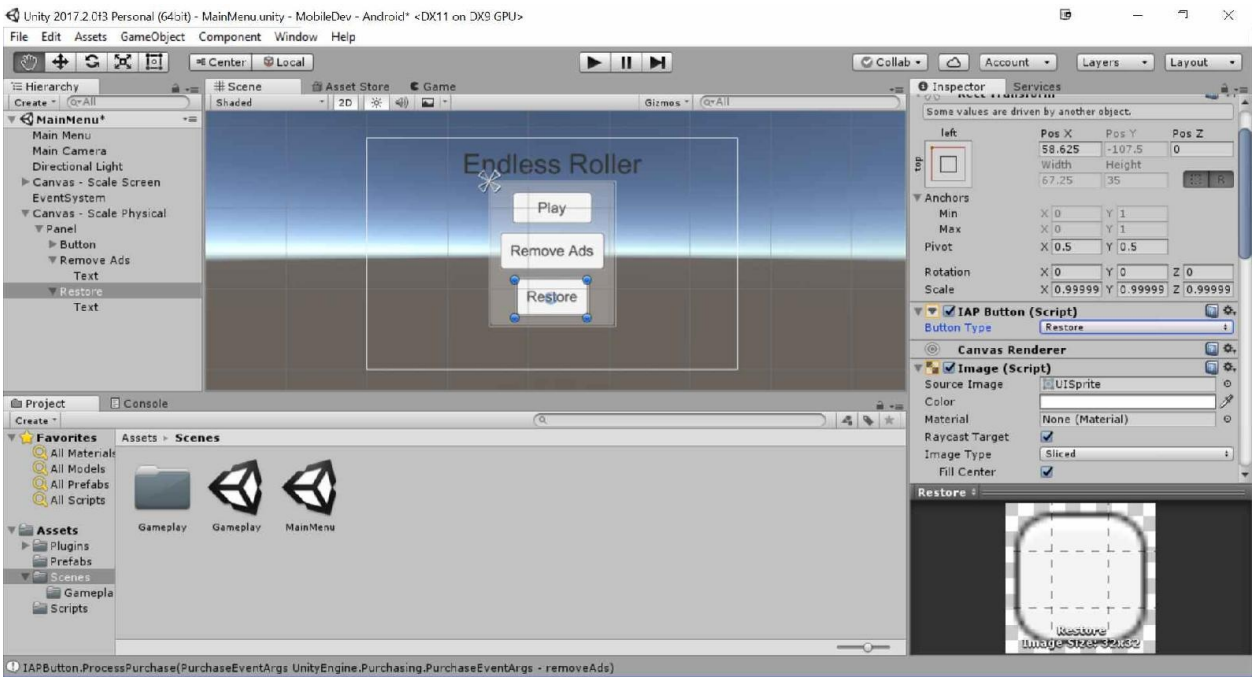
Adding button to restore purchases

On platforms that support it (Google Play and Universal Windows Applications, most notably), if you purchase something, uninstall, and then reinstall a game using Unity IAP, it automatically restores any products the user owns during the first initialization following reinstallation. For those on iOS, users must have the ability to restore their purchases via a button due to Apple requiring them to reauthenticate their password before it happens. Not doing so will prevent our game from being accepted on the iOS App Store, so it's a good idea to include this functionality if we wish to deploy there, as follows:

1. Go to the Hierarchy window and select the Remove Ads button. Once selected, duplicate it by pressing *Ctrl + D*.
2. Change the duplicate's name by selecting it and changing its name to `Restore` from Inspector.
3. From the Hierarchy tab, open up the Text object and change the text to say Restore as well:

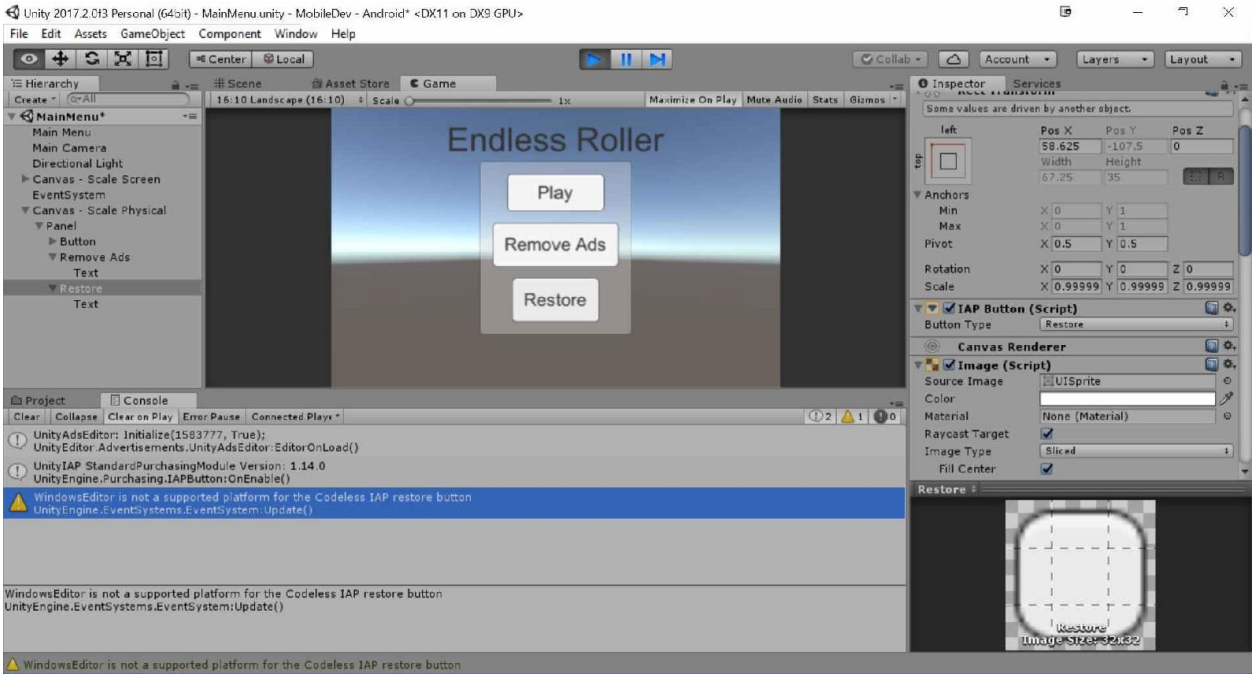


4. Now, select the Restore object, and then in the IAP Button component, go to Button Type and select Restore:



You should note that the properties change to only have this type.

5. Save your scene and jump into Unity:



When you start the game and try to click on the Restore button, you'll get a warning stating that this isn't a supported platform. So, with that in mind, we can adjust our game so that the button will only show up

if we are currently running on a supported platform.

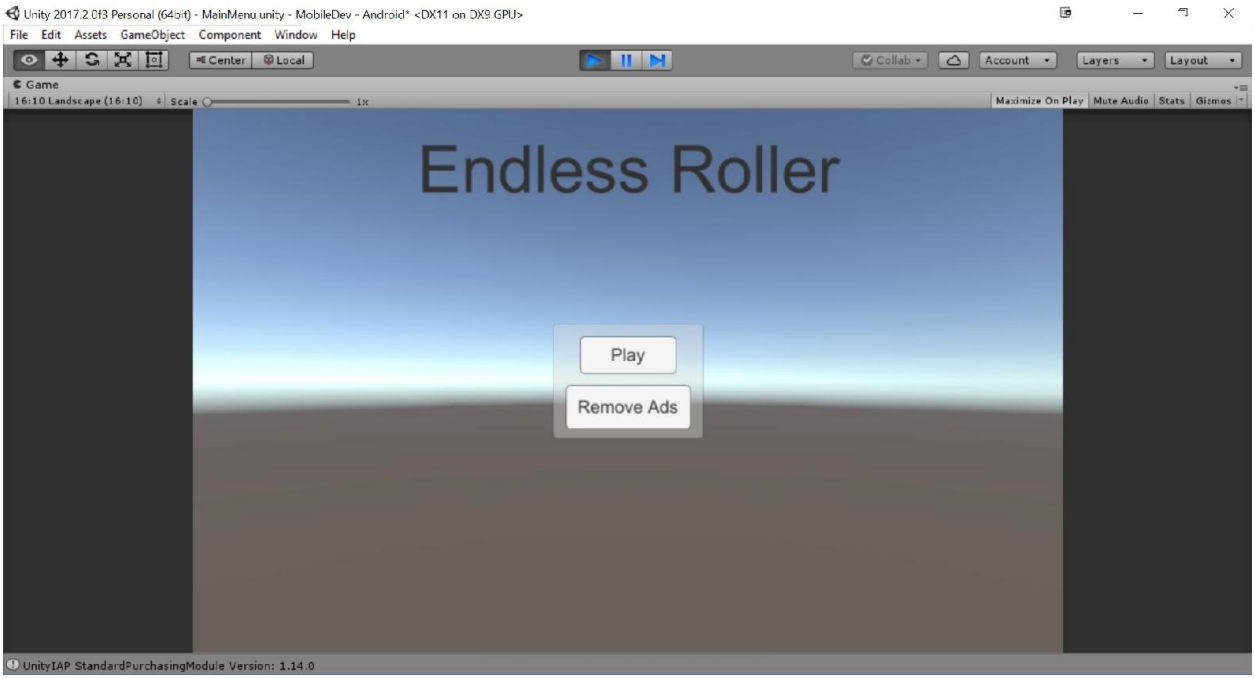
6. Go to the `Scripts` folder and create a C# script called `RestoreAdsChecker`. Once it's opened, use the following script for it:

```
using UnityEngine;
using UnityEngine.Purchasing;

/// <summary>
/// Will show or remove a button depending on if we can restore /// ads or not
/// </summary>
public class RestoreAdsChecker : MonoBehaviour
{

    // Use this for initialization
    void Start()
    {
        bool canRestore = false;

        switch (Application.platform)
        {
            // Windows Store
            case RuntimePlatform.WSAPlayerX86:
            case RuntimePlatform.WSAPlayerX64:
            case RuntimePlatform.WSAPlayerARM:

            // iOS, OSX, tvOS
            case RuntimePlatform.IPhonePlayer:
            case RuntimePlatform.OSXPlayer:
            case RuntimePlatform.tvOS:

                canRestore = true;
                break;

            // Android
            case RuntimePlatform.Android:
                switch (StandardPurchasingModule.Instance().appStore)
                {
                    case AppStore.SamsungApps:
                    case AppStore.CloudMoolah:
                        canRestore = true;
                        break;

                }
                break;
        }

        gameObject.SetActive(canRestore);
    }

}
```

This script goes through all of the stores listed in Unity's `IAPButton` class, and if they are something that can be restored, we set `canRestore` to `true`, otherwise it will stay as `false`. Finally, we will remove the object if we cannot restore it, without having to create specific things for different builds.

7. Save the script and dive back into Unity.
8. Attach our newly created `RestoreAdsChecker` component to our Restore button.

9. Save your project and start up the game:



On our PC build of the game, the Restore button doesn't show up, but if we export for iOS it will be on our device!

Configuring purchases for the stores of your choice

Unfortunately, we do not have enough room in the book to go step by step through the process for every store, but I do have pages that you can reference to go through the entire process for the following stores: Apple App Store and Mac App Store: https://docs.unity3d.com/Manual/UnityIAPAppleConfiguration.html.

Google Play Store: https://docs.unity3d.com/Manual/UnityIAPGoogleConfiguration.html.

Windows Store: https://docs.unity3d.com/Manual/UnityIAPWindowsConfiguration.html.

Amazon Appstore and Amazon Underground: https://docs.unity3d.com/Manual/UnityIAPAmazonConfiguration.html.

Samsung Galaxy: https://docs.unity3d.com/Manual/UnityIAPSamsungConfiguration.html.

Tizen Store: https://docs.unity3d.com/Manual/UnityIAPTizenConfiguration.html.

CloudMoolah Moo Store: https://docs.unity3d.com/Manual/UnityIAPMoolahConfiguringMooStore.html.

There are some potential issues when trying to publish to multiple Android IAP stores (such as Samsung and Google) with the same build. You can find information on resolving those issues at https://docs.unity3d.com/Manual/UnityIAPCrossStoreInstallationIssues.html.

Summary

In this chapter, we covered how to create in-app purchases, making use of Unity in your project. We first covered how to set up Unity's IAP system and then dived into using Codeless IAP to easily add a purchasable item to your game. We then created the functionality to restore our purchase if we uninstall and reinstall our game and went over where we can go to set up our purchases depending on the store we want to target.

Now, of course, having all these ways to make money isn't going to help us if no one is playing our game. In the next chapter, we will get social, learning how we can make use of social media to share our score and get other players interested in our title.

Getting Social

We now have all of the foundational things needed to get our game out into the world; it's mechanically working, and we've set up all of the monetization. Having all of the features that we have added to our projects is great, but if no one is playing your game, there's no reason to have them.

Word of mouth marketing is the most reliable way to get others to try your game, so providing people opportunities to share the game is an easy way to help others discover the project and something that we should really try to do, as marketing and getting your game out there is one of the hardest things to do as an indie developer.

Chapter overview

In this chapter, you will learn some of the different ways to integrate social media into your projects.

Your objectives

This chapter will be split into a number of topics. It will contain a simple step-by-step process from beginning to end. The following is the outline of our tasks:

- Adding in a score system
- Sharing high scores via Twitter
- Downloading and installing the Facebook SDK
- Logging to our game via Facebook
- Displaying Facebook name and profile picture

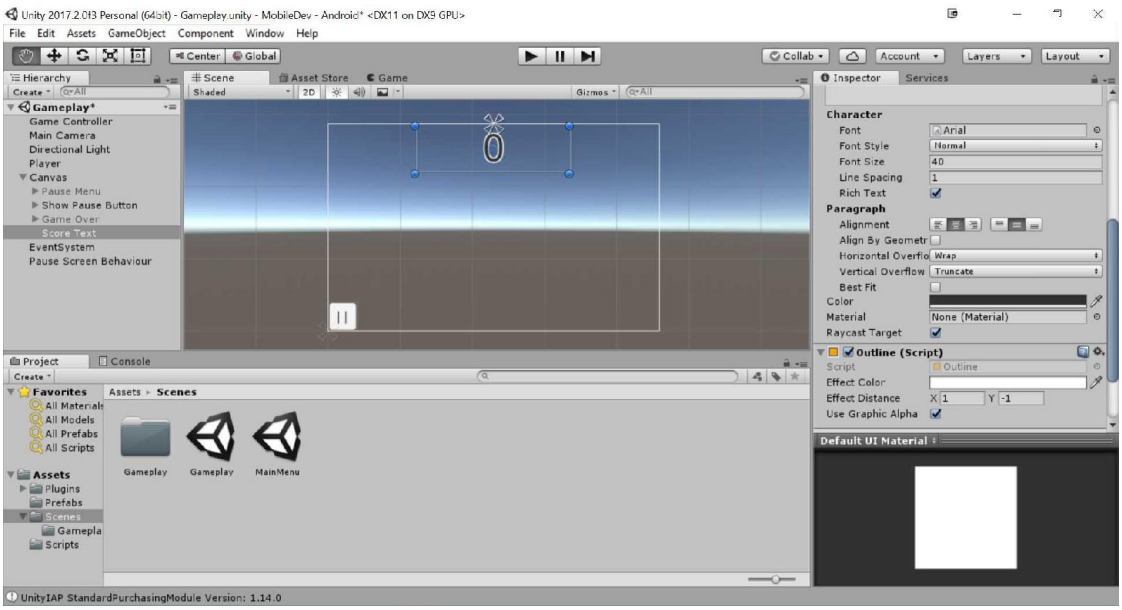
Adding a score system

In order to provide incentive to players to share our game with others, we need to provide a compelling reason to do so. Some people are very competitive and wish to be the best at playing a game, challenging others to do better than them. To help with that, we can allow our players to share a score value via social media. However, to do that, we'll first need to have a scoring system. Thankfully, it's not too difficult to do that, so let's add that in real quick:

1. Start off by opening the `Gameplay.scene` file located in the `Assets/Scenes` folder.

   > To show our players what their score is, we'll need to have some way to show it. In our case, the easiest way would be with a text object.

2. From the Hierarchy window, right-click on the Canvas object and select UI | Text.
3. Rename this object `Score Text` and use the Anchors Preset menu to the top-center, holding down *Shift* + *Alt* to set the pivot and position as well. Afterward, let's set the RectTransform component's Height property on the object to `50` to ensure that we have space to hold the score when we increase the size.
4. Next, in the Text component, change the Text property to `0` and set the Alignment to be centered. Afterward, set the Font Size to `40` so that it's easy to see.
5. To improve its readability, let's add another component, the Outline, by going to Add Component and then typing in `Outline` and pressing Enter.
6. From there, set the Effect Color field to `White`:



7. Next, open up the `PlayerBehaviour` script and add the following line at the top:

```
using UnityEngine.UI; // Text
```

8. Afterward, add the following code inside the class:

```
private float score = 0;

public Text scoreText;



public float Score

{

    get { return score; }

    set

    {

        score = value;
```

```
        // Update the text to display the whole number portion // of the score

        scoreText.text = string.Format("{0:0}" ,score); }

}
```

This makes use of C#'s `get`/`set` functions, which are implicit getters and setters. Basically, any time we get or set the `Score` variable, we will execute whatever is located between the `{}`. In our case, any time we set the `Score` variable, it will update our text for us.

This has an advantage over what a number of my students do, which is to update the value of the text every frame, which doesn't need to happen. We only need to update the text when the value changes, which makes it perfect for us to use in this situation.

For more information on the `get/set` accessors, check out https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/using-properties.

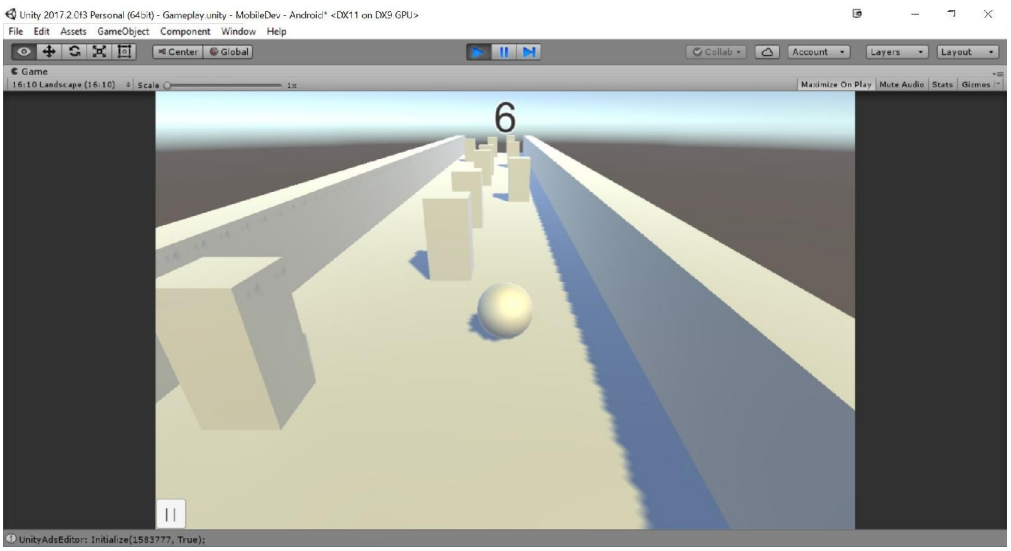
9. Then, update the class to have the following highlighted changes:

```
/// <summary> /// Use this for initialization /// </summary>

    new void Start ()

   {

        // Get access to our Rigidbody component rb = GetComponent<Rigidbody>();

       Score = 0;

   }
```

```
    /// <summary>

    /// Update is called once per frame /// </summary>

    void Update ()

   {

        // If the game is paused, don't do anything if (PauseMenuBehaviour.paused)
return;

        Score += Time.deltaTime;




        // Movement in the x axis float horizontalSpeed = 0;

   // Rest of Update here...
```

What we are doing here is resetting our score whenever the Player is created and increasing the value while the game isn't paused.

10. Save the script and dive back into Unity.
11. Select the Player object and drag and drop our Score Text object into the Score Text variable, as follows:



Now, as you can see, we have a score for our game, which updates as we play.

Sharing high scores via Twitter

Now that we have a scoring system, let's take a look at how we can share a high score using Twitter.

Twitter is an online news and social networking service where users post and interact with each other through messages that they call *tweets*, which are limited to 280 characters.

Twitter is a great option to start off with because it is very easy to add to our project by simply opening a specific URL:

1. Open the `PauseMenuBehaviour` script. Once inside there, we will add the following code inside the `PlayerMenuBehaviour` class:

```
#region Share Score via Twitter

// Web address in order to create a tweet
private const string tweetTextAddress =
                            "http://twitter.com/intent/tweet?text=";

// Where we want players to visit
private string appStoreLink = "http://johnpdoran.com/";

// Reference to the player for the score
public PlayerBehaviour player;

/// <summary>
/// Will open Twitter with a prebuilt tweet. When called on iOS or
/// Android will open up Twitter app if installed
/// </summary>
public void TweetScore()
{
    // Get contents of the tweet (in URL friendly format)
    string tweet = "I got " + string.Format("{0:0}", player.Score)
+ " points in Endless Roller! Can you do better?";

    // Open the URL to create the tweet
    Application.OpenURL(tweetTextAddress + WWW.EscapeURL(tweet +
                          "\n" + appStoreLink));
}

#endregion
```

   First of all, we will use a number of new things. You'll note that this new block of code starts and ends with `#region` and `#endregion`, respectively. What this does is allow us to expand and collapse this

portion of code inside Visual Studio. When we introduce longer code files, it can be convenient to be able to collapse or hide certain parts of your script so that you can focus only on the part of the file you're working on. Since this portion of code has nothing to do with the rest of the script, this is a good place for us to use it.

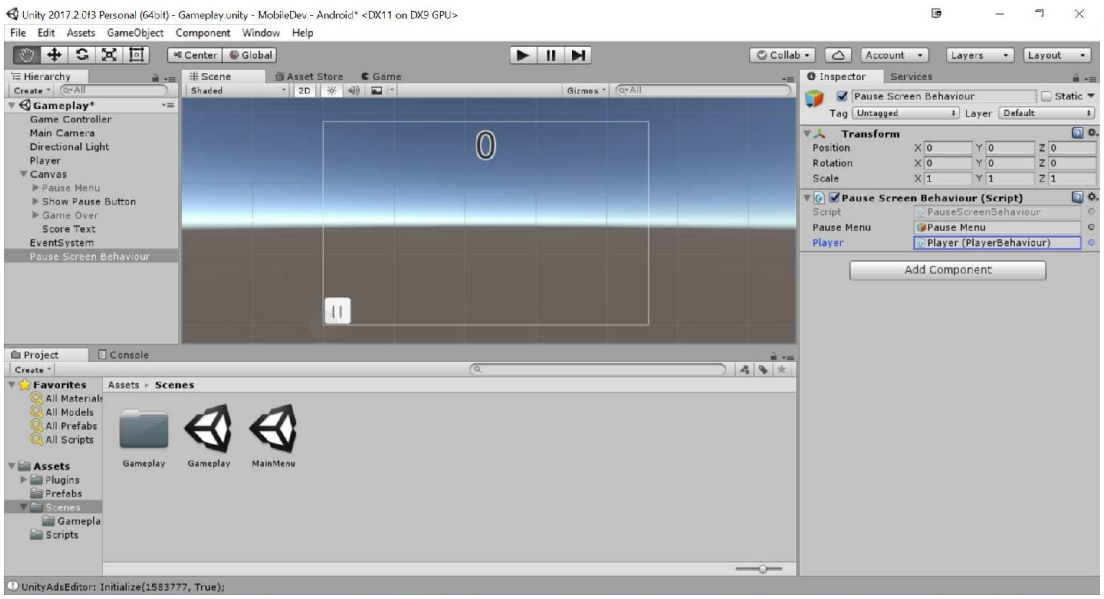
To open URLs inside of Unity, we will need to make use of the `Application.OpenURL` function and the `www` class.

For more information on Twitter's web Intents and the ways you can use it, check out https://dev.twitter.com/web/intents.

The `www` class is typically used to load content at runtime, but it also has the `EscapeURL` function, which will convert a string into a format that web browsers are comfortable with. For instance, the newline character will not be displayed by itself.

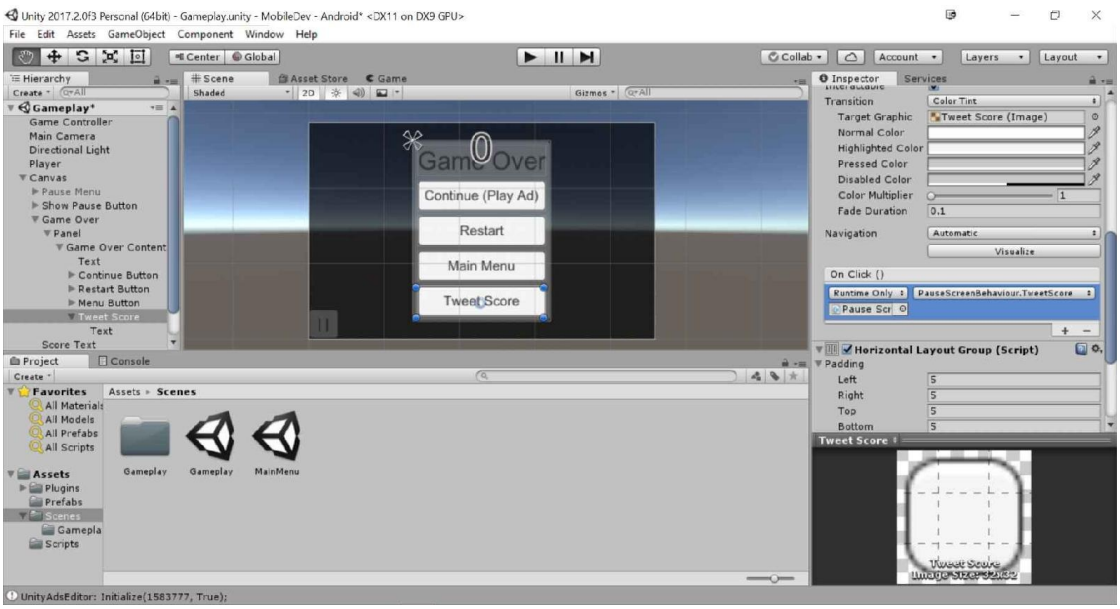
For more information on the `EscapeURL` function, check out https://docs.unity3d.com/ScriptReference/WWW.EscapeURL.html.

2. Save the script and dive back into Unity. From the Hierarchy window, select the Pause Menu Behaviour object and then set the Player property in the Inspector tab to our `Player` object by dragging and dropping the Player game object from the Hierarchy window onto the Player property in the Inspector:



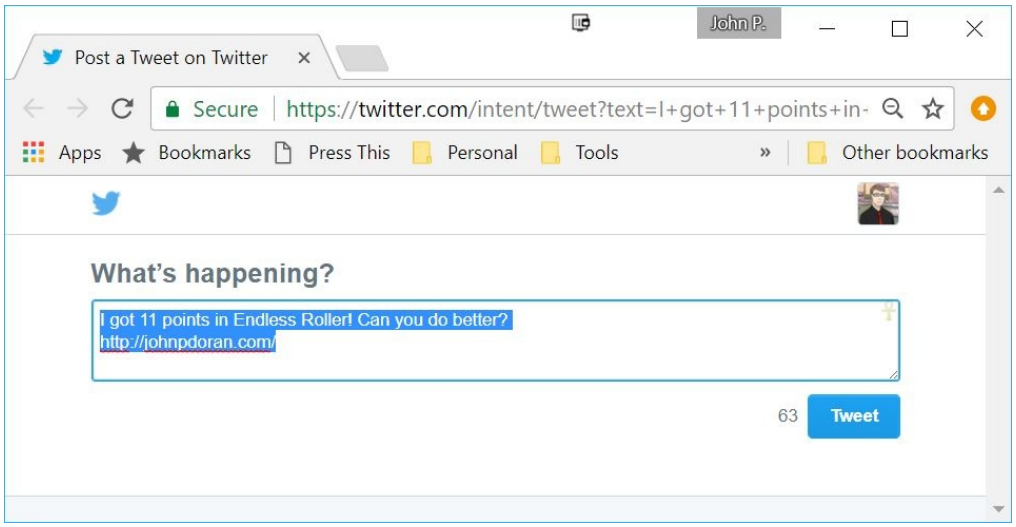
3. Now, we need to have a button for our Game Over screen to allow us to share our score.
4. Open up the Canvas object and toggle the Game Over object on by clicking on the checkmark beside its name in the Inspector window.

5. From there, expand the Panel and the Game Over contents. Select the Menu Button and duplicate it by pressing *Ctrl + D*. Next, change the name to `Tweet Score` and also update the text.
6. Afterward, select the Tweet Score button object and scroll down to the Button component. From there, change the function we are calling to the PauseScreenBehaviour | Tweet Score function:

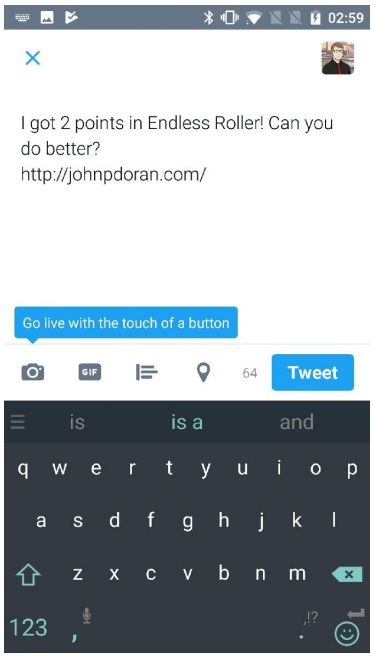


7. Select the Game Over object in the Hierarchy and disable it again. Next, save your scene and start the game.

Now when we fail the game, we can click on the Tweet Score button and our browser will open on PC:



However, on our mobile devices, it will open up the Twitter app:



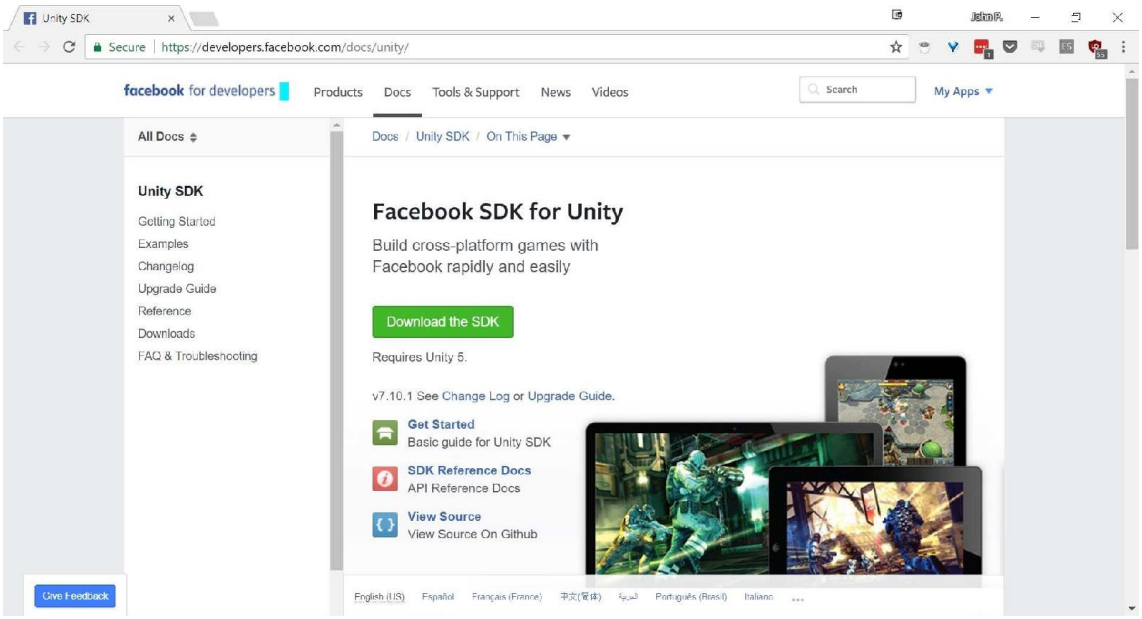
With that, you learned just how easy it is to share something using Twitter.

For those who are interested in doing more than this with Twitter, they do have their own API for Unity, which will allow you to let users log in to your game using Twitter if you'd like to do that instead of Facebook, which we will be doing later on. If you're interested in looking into it, you can find more information at https://dev.twitter.com/twitterkit/unity/overview.

Downloading and installing Facebook's SDK

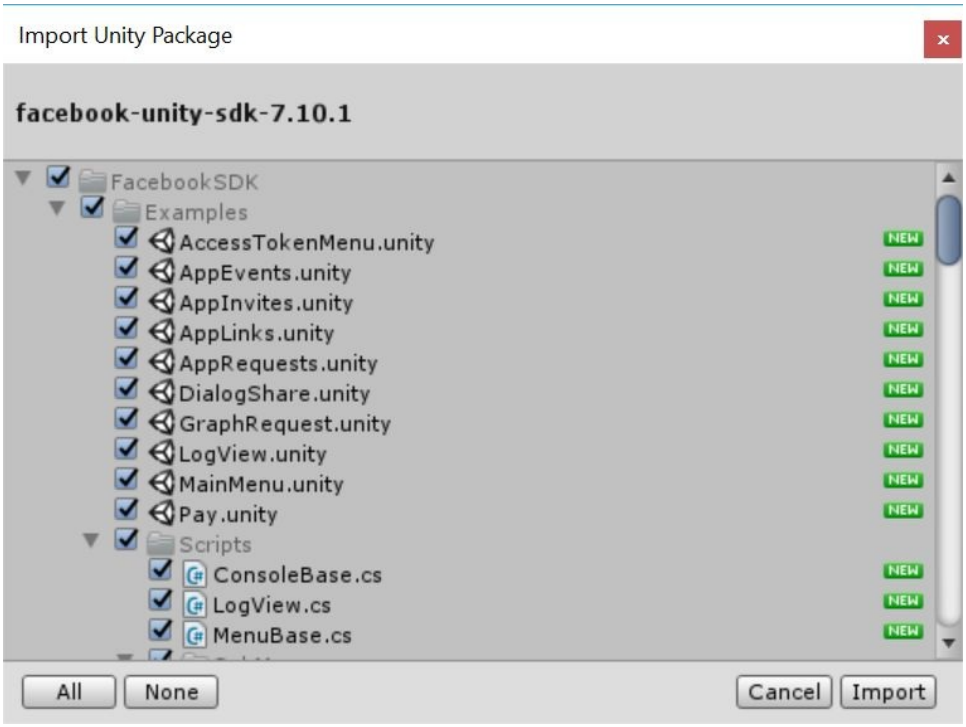
We couldn't have a chapter on social networking without mentioning Facebook. Facebook has its own SDK that can be used with Unity.

1. Open up your web browser and visit: https://developers.facebook.com/docs/unity/:



2. Click on the Download the SDK button and wait for it to finish downloading. Once it is downloaded, unzip it and then open up the `facebook-unity-sdk-7.10.1` folder. Then, open up the `FacebookSDK` folder and you'll see a single file, `facebook-unity-sdk-7.10.1.unitypackage`.

3. Double-click on the `unitypackage` file, and you should have a window pop up:

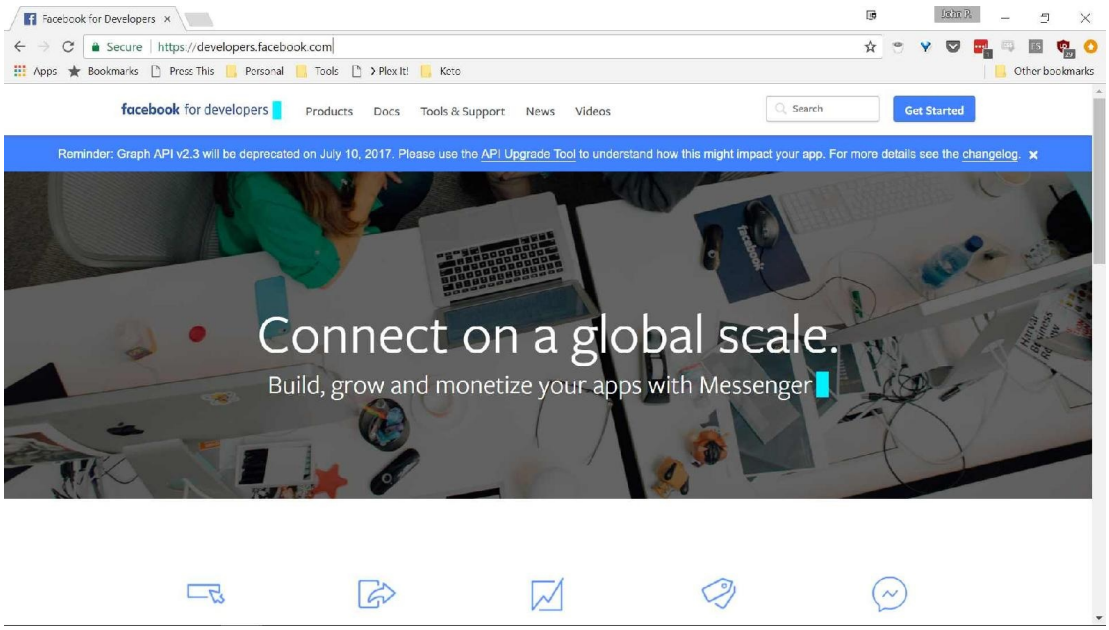


If this does not work, you can also go to Assets | Import Package | Custom Package and then find the folder that you unzipped the file to and open it that way.

4. Click on the Import button and wait for it to finish loading.

   Now, in order to use the Facebook API, we will first need to have a Facebook App ID, so let's do that next.

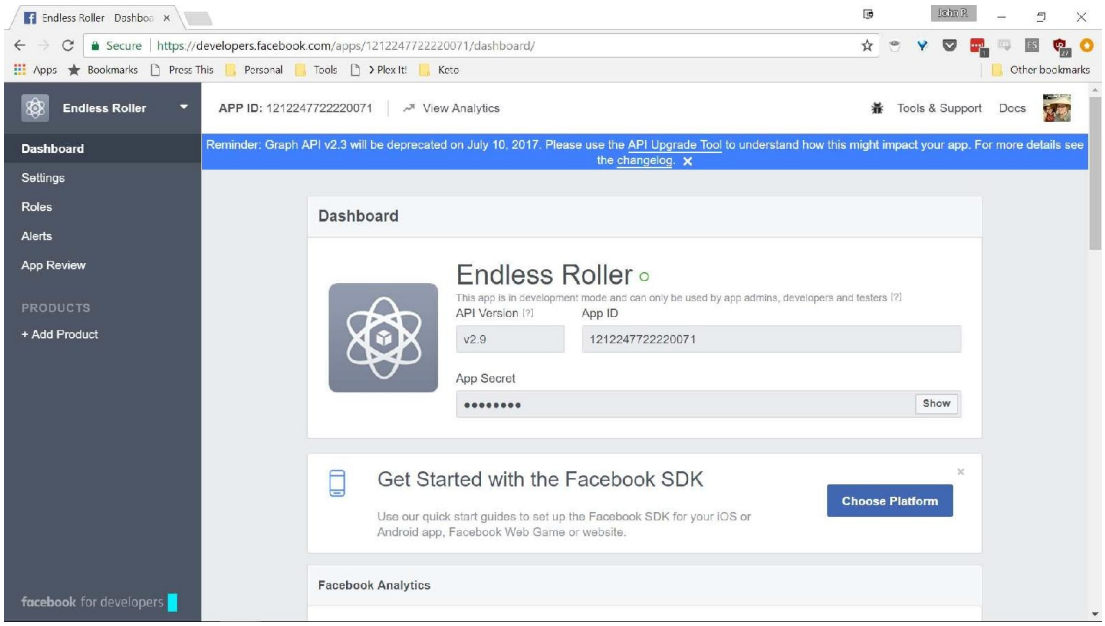
5. Go back to your web browser and go to https://developers.facebook.com/:



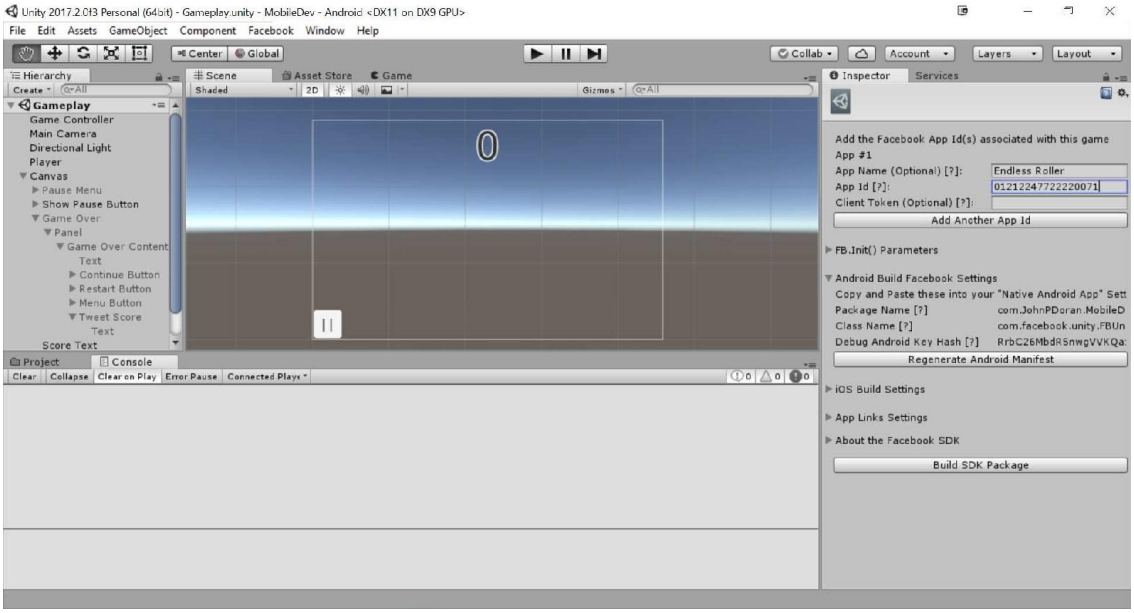
6. From the preceding page, click on the Get Started button on the top-right corner of the screen. From there, you'll be brought to a screen where you need to accept Facebook's Platform Policy and Privacy Policy. Click on Yes and then on the Register button.
7. From there, then you'll see a screen stating that you're all set and then we can click on the Create App ID button.

8. Once loaded, set your Display Name and Contact Email to what you want the game and your email to be called, respectively, and then click on the Create App ID button once again.
9. If you haven't added a credit card or mobile number on your phone, you'll need to add that first. So, do that if you haven't already, and the app should be created after these steps.

10. Once you're brought to your App's page, click on the Dashboard option to be right to the default info for your game. Note the App ID and copy it by highlighting it and then pressing *Ctrl + C*:



11. Return to Unity, and you will note a new option of Facebook on the top bar. Select it and then select Edit Settings. Once there, click on Inspector if you need to and you'll see a number of options here. Set the App Id (?) to our created App ID and then set the name to our game's name:



12. Now that we have set that up, we can start adding onto this.

There are a few other properties that need to be set for building the Facebook SDK, depending on the mobile platforms you are trying

to target.

For Android, check out https://developers.facebook.com/docs/unity/getting-started/android.

For iOS, check out https://developers.facebook.com/docs/unity/getting-started/ios.

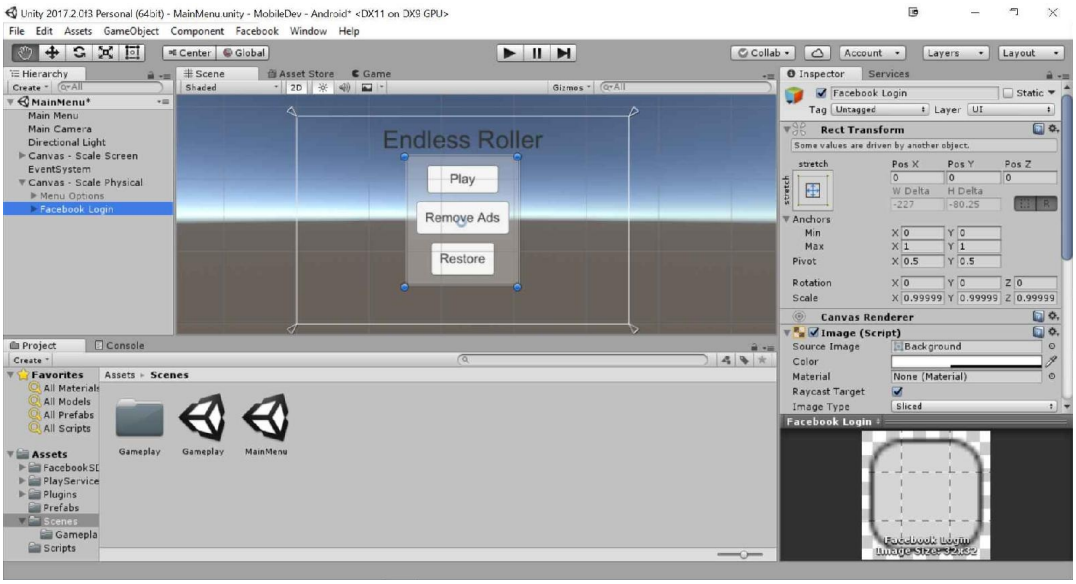
Logging in to our game via Facebook

One of the things we can do using the Facebook API is allow our users to log in to the game using their Facebook account. Then, we can use their name and image automatically within our project:

1. Let's first open up our Main Menu level by going to the Project window and open the `Assets/Scenes` folder and then double-click on the `MainMenu` file.
2. From there, let's click on the 2D button to go into 2D mode if you haven't done so previously.

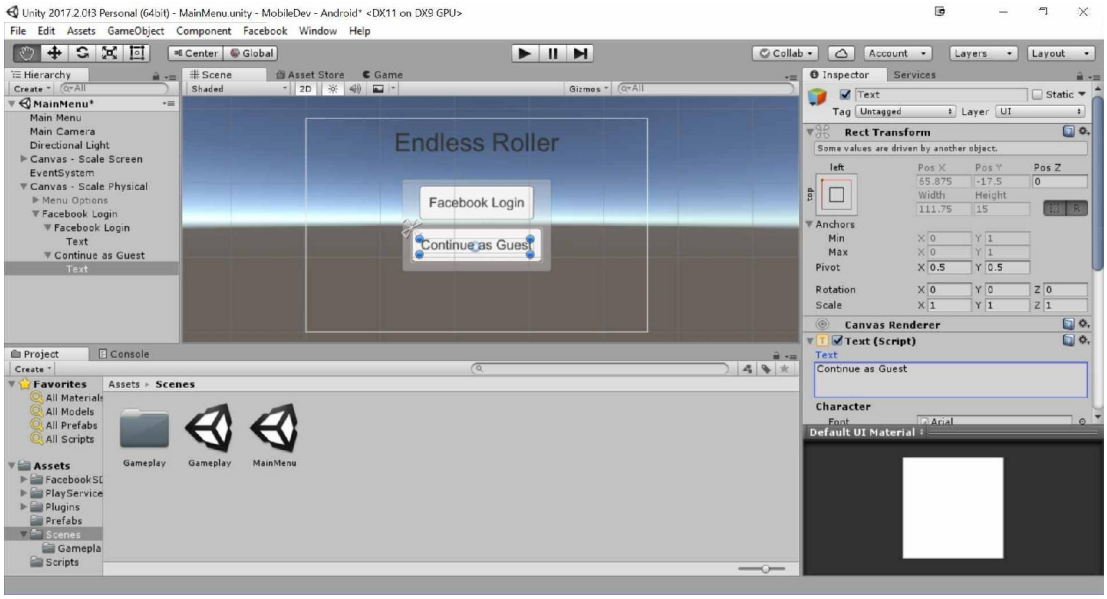
   What we will do is remove the original menu and instead have a button for players to log in via Facebook or play as a guest when the game starts.

3. Go to the Hierarchy window and select the Canvas - Scale Physical object and expand it. Select the Panel child and rename it `Menu Options`.
4. Then, duplicate it by pressing *Ctrl + D*. Then, rename the newly created object `Facebook Login`. Select the Menu Options again and then disable it by going to the Inspector tab and then clicking on the check beside its name:



We will have the `Facebook Login` object turn the menu on when needed.

5. Next, open the Facebook Login options and remove the Restore and Button objects. Click on the Remove Ads button, right-click on the IAP Button component, and then select Remove component. Duplicate the Remove Ads button by pressing the *Ctrl + D* keys. Then, name those two buttons Facebook Login and Continue as Guest, changing the text as well:



6. Now that we have the buttons working correctly, we need to write the script that will allow us to log in. Go to the `Scripts` folder and open our `Main Menu Behaviour` script.
7. We will use the `List` class to hold the permissions we want to access on Facebook and the content of the `FB` class in the Facebook SDK. So, to do that, we'll first add the following to the top of the script:

```
using System.Collections.Generic; // List
using Facebook.Unity; // FB
```

8. Now, add the following code within the class:

```
#region Facebook

        public void Awake()
    {
        // We only call FB Init once, so check if it has been called
        // already
        if (!FB.IsInitialized)
        {
            FB.Init(OnInitComplete, OnHideUnity);
        }
    }
    /// <summary>
```

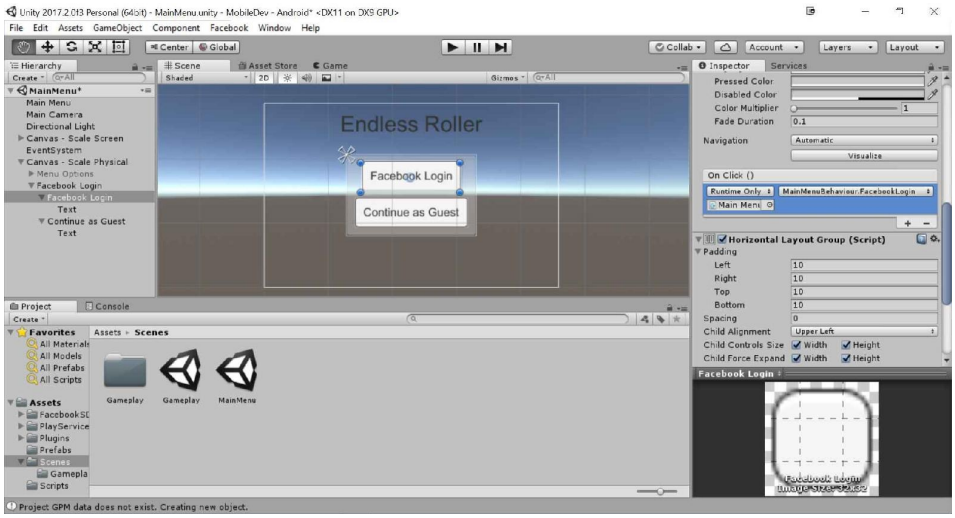
```
    /// Once initialized, will inform if logged in on Facebook
    /// </summary>
    void OnInitComplete()
    {
        if (FB.IsLoggedIn)
        {
            print("Logged into Facebook");

            // Close Login and open Main Menu
            ShowMainMenu();
        }
    }

    /// <summary>
    /// Called whenever Unity loses focus
    /// </summary>
    /// <param name="active">If the gmae is currently active</param>
    void OnHideUnity(bool active)
    {
        // Set TimeScale based on if the game is paused
        Time.timeScale = (active) ? 1 : 0;
    }

    /// <summary>
    /// Attempts to log in on Facebook
    /// </summary>
    public void FacebookLogin()
    {
        List<string> permissions = new List<string>();

        // Add permissions we want to have here
        permissions.Add("public_profile");

        FB.LogInWithReadPermissions(permissions, FacebookCallback);
    }

    /// <summary>
    /// Called once facebook has logged in, or not
    /// </summary>
    /// <param name="result">The result of our login request</param>
    void FacebookCallback(IResult result)
    {
        if (result.Error == null)
        {
            OnInitComplete();
        }
        else
        {
            print(result.Error);
        }
    }

    [Header("Object References")]
    public GameObject mainMenu;
    public GameObject facebookLogin;

    public void ShowMainMenu()
    {
        if (facebookLogin != null && mainMenu != null)
        {
            facebookLogin.SetActive(false);
            mainMenu.SetActive(true);
        }
    }
```

```
#endregion
```

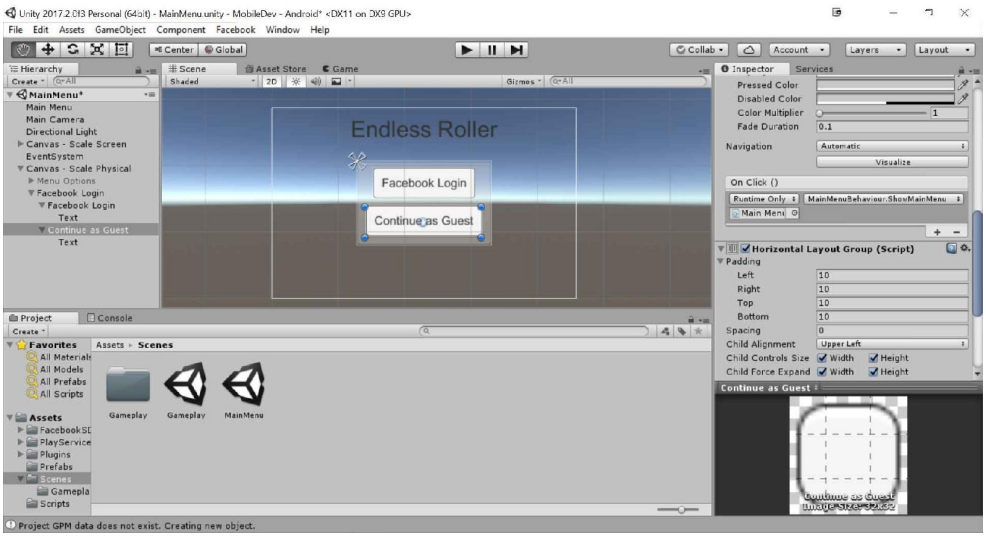
In this case, we are accessing the player's public profile, which contains information such as their name and their profile picture.

For all of the properties that we can get access to, check out https://developers.facebook.com/docs/facebook-login/permissions#reference-public_profile.

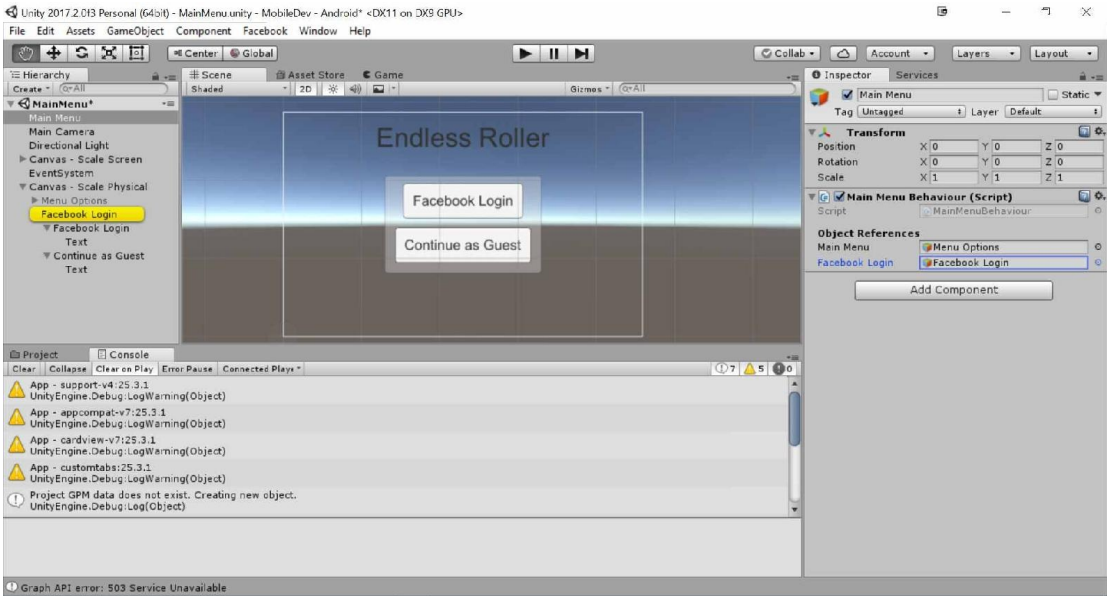
9. Save your script and go to the Facebook Login button and change the button's OnClick() action to now call your function by clicking on the + button and then dragging and dropping the Main Menu object in and then selecting Main Menu Behaviour | Facebook Login instead:



10. Then, on the Continue as Guest button under the Button component, go to the On Click () section and then click on the + button. Drag and drop the Main Menu object into it and select the MainMenuBehaviour | ShowMainMenu function:

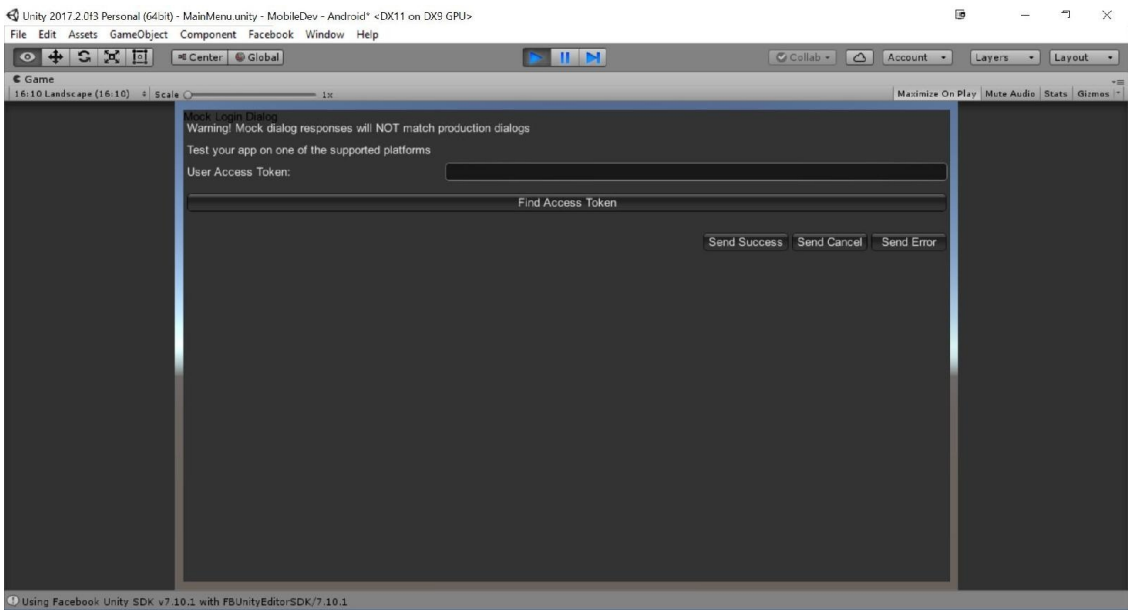


11. Finally, we will need to set the variables we created. Select the Main Menu object in the Hierarchy window and then set the Main Menu and Facebook Login properties:



Ensure that the Facebook Login is set to the panel object holding both buttons.

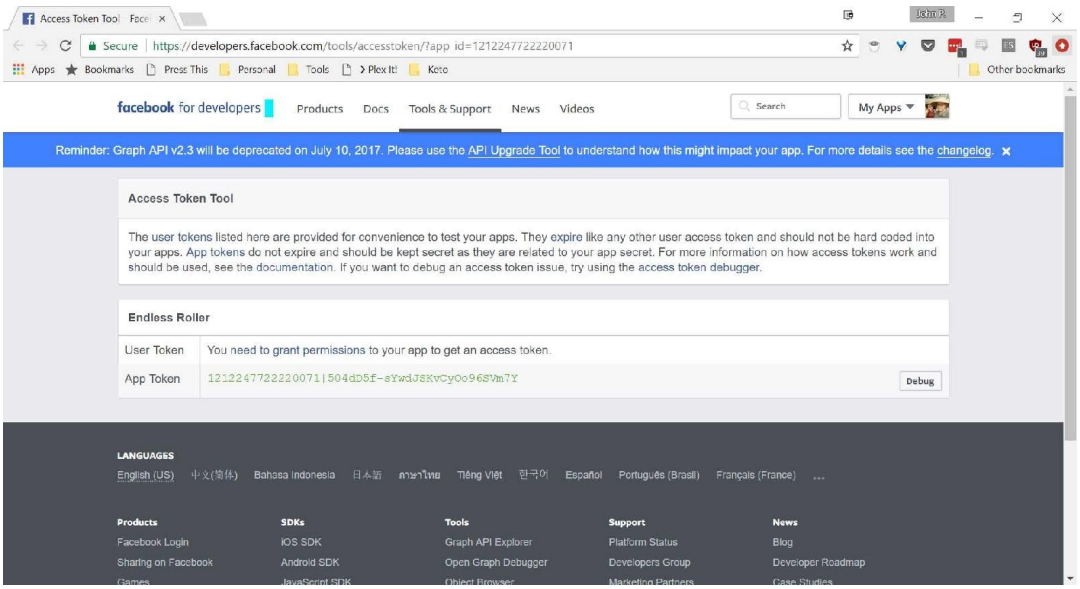
12. Save your scene and start the game and click on the Facebook Login button:



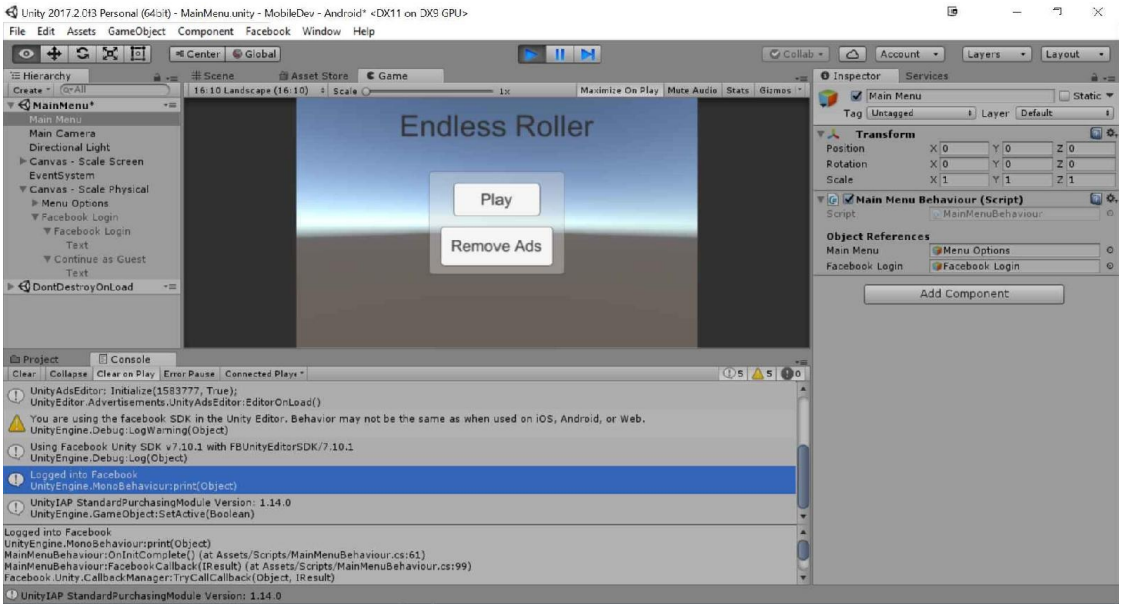
To see everything properly within the editor, it's a good idea to maximize the Game tab, which you can do by right-clicking on the Game tab and selecting Maximize or checking the Maximize On Play option on the toolbar.

Now, you should see a menu, which is asking for a user access token, a value that every profile has that we can associate with. We'll need to go to Facebook to get that, so that's what we'll do next.

13. Click on the Find Access Token page, and a web browser will open with a new page:



14. You'll then need to click on the need to grant permissions link and then on Continue and you'll see a string of characters under user token. Copy that and then paste it into Unity and then click on the Send Success button:



Now, you'll note that the console has printed that we've logged into Facebook and the menu has closed when we've sent the key.

For more information on user access tokens, check out https://developers.facebook.com/docs/facebook-login/access-tokens/#usertokens.

Displaying Facebook name and profile pic

A good thing to do is to personalize our game to fit our player. So, with that, once the player logs in, we will welcome them and display their image on the screen by following these steps:

1. Go to the `MainMenuBehaviour` script once again. From there, we'll need to add a new using statement as to display an image and change text we need to use Unity's UI system:

```
using UnityEngine.UI; // Text / Image
```

2. Afterward, we will update the `ShowMainMenu` function and add some new functions to use:

```
public void ShowMainMenu()
    {
        if (facebookLogin != null && mainMenu != null)
        {
            facebookLogin.SetActive(false);
            mainMenu.SetActive(true);

            if (FB.IsLoggedIn)
            {
                // Get information from Facebook profile
                FB.API("/me?fields=name", HttpMethod.GET, SetName);
                FB.API("/me/picture?width=256&height=256",
                HttpMethod.GET, SetProfilePic);
            }
        }

    }

    public Text greeting;

    void SetName(IResult result)
    {
        if (result.Error != null)
        {
            print(result.Error);
            return;
        }

        string playerName = result.ResultDictionary["name"].ToString();
        greeting.text = "Hello, " + playerName + "!";

        greeting.gameObject.SetActive(true);
    }
```

```
public Image profilePic;

void SetProfilePic(IGraphResult result)
{
    if (result.Error != null)
    {
        print(result.Error);
        return;
    }

    Sprite fbImage = Sprite.Create(result.Texture,
                                   new Rect(0,0, 256, 256),
                                   Vector2.zero );
    profilePic.sprite = fbImage;

    profilePic.gameObject.SetActive(true);
}
```

The `FB.API` function makes a call to Facebook's Graph API to get data or take an action on the user's behalf and allows us to get the information that we have permission to as defined earlier.

In our case, we are looking for the name and the profile picture of the user and calling the `SetName` and `SetProfilePic` functions, respectively, once we have obtained that data.

After getting the data, we will modify the image or string to display this new data that we retrieved.

For more information on the `FB.API` function, check out https://developers.facebook.com/docs/unity/reference/current/FB.API.

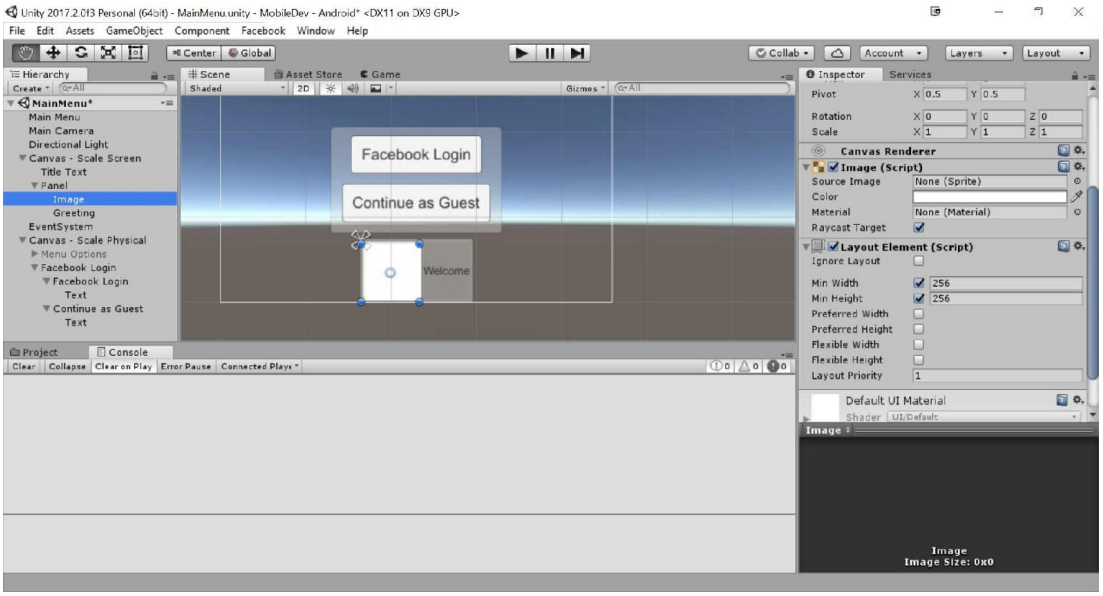
3. Now, we will need to actually create the text and image we want to display. Open up the Canvas - Scale Screen object in the Hierarchy tab and right-click on it and select UI | Panel.
4. This will act as a container for all of our information for the player. Add a Horizontal Layout Group with the Padding and Spacing both set to `10`. From there, change the Child Alignment to Lower Center and then check Width and Height under the Child Control Size property. Then, add a Content Size Fitter component and change the sizes to Preferred Size. Finally, in the Anchor Presets menu, hold down Alt+Shift and select bottom-center.
5. Now, select the Canvas - Scale Screen object in the Hierarchy tab and right-click on it and select UI | Text.

6. Rename the next Text object `Greeting`. From the Anchors Preset menu that we learned about earlier, change the setting to bottom-stretch, holding the *Shift + Alt* to adjust the anchors and presets accordingly. Then, change the Height to `75` so that it's large enough to increase our size.
7. Then, adjust the Text to `Welcome` and the size to something larger like `50`, and then adjust the Alignment to be centered vertically and horizontally.
8. Likewise, let's next right-click on Canvas - Scale Size again, and this time select Image. From there, change the Width and Height to a larger value, such as `256`.

    > You may have noted that when the image was at the center, the image was drawn on top of our menu. This is possible due to both Canvas' being told that they have the same priority in being drawn, similar to how z-fighting works for 2D games. To fix potential problems in the future, we will instead put the scaling canvas as the background element.

9. To do this, we will select the Canvas - Scale Physical, and under the Canvas component, change the Sort Order to `1`.

10. Now, let's go ahead and drag and drop the Greeting and Image objects into the Panel. You'll note that the image has reverted back to becoming super small. To fix this, select the Image object and add a Layout Element component to it. From there, set the Min Width and Min Height to `256`:

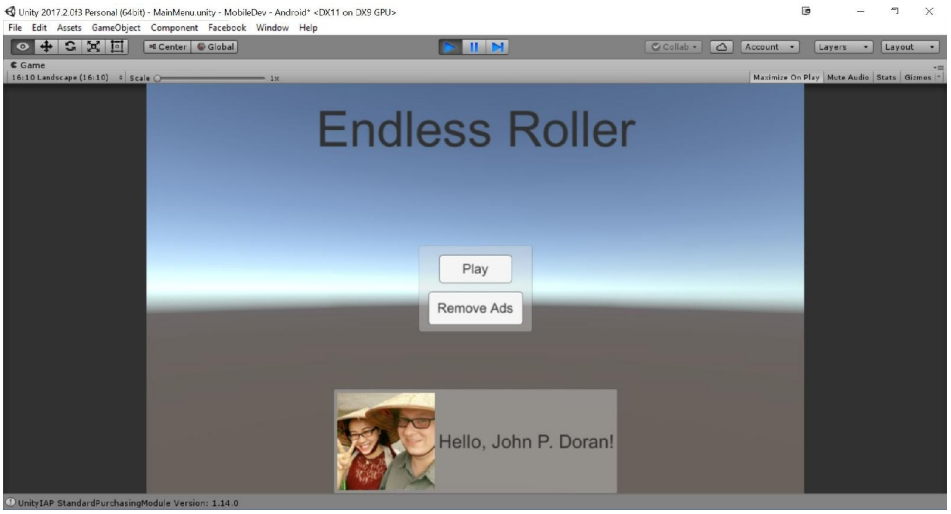


The Layout Element (Script) component is great for allowing you to override things that Layout Groups will do by default and can be useful if you're not getting exactly what you want with the default behavior.

For more information on the Layout Element (Script) component, check out https://docs.unity3d.com/Manual/script-LayoutElement.html.

11. Now, dive back into the Main Menu object and set the Greeting and Image properties in the MainMenuBehaviour component.
12. Finally, since we don't want them visible when the game starts, let's turn off Greeting and our Image as well.

13. Save our game, and start it up again going through the appropriate login information:



As you can see in the preceding screenshot, I retrieve my actual Facebook info once I've logged in.

For more information on the Facebook SDK for Unity and additional examples on what you can do with it, check out ;https://developers.facebook.com/docs/unity.

Summary

In this chapter, we were introduced to some of the potential ways that we can share our game with others, as well as personalize our game experiences, utilizing social media and the functionality that it provides to us. We started off by adding in a simple score system and then allowed others to share their score via Twitter. We then set up the Facebook SDK, making it so we can log into it to play our game and retrieve information about our users, which we can use to customize their gameplay experience.

Now that we have people playing our game, we may want to see what they're doing. Then, we can use that information to improve and/or tweak our game. In the next chapter, we will take a look at how we can do this using tools from Unity analytics.

Using Unity Analytics

Having made a game by itself is a wonderful experience and a lot of hard work, but when designing projects, you have to rely on your experience and gut feelings in order to make it as awesome as possible. Often, in the game industry, we will use playtesting--a method where selected people play the game, watch them, and then we use the feedback we receive to improve the project.

This playtesting is most often done in person; however, by creating games for mobile, a lot of people will be playing your game after release and most of them will have an internet connection. With this, we can send pieces of data about how the game is being played to ourselves. This will still allow us to do "playtesting" with a large variety of people. Being able to look at our data will allow us to check whether the choices that are made to change the game are the right ones, and we will be able to make adjustments to our games on the fly.

This could be something as simple as where the player dies in the game to things such as how often they come back to play, the daily average time they play, the number of users we have at a time, how long people play the game before stopping, and what choices they made.

Chapter overview

In this chapter, we will cover some of the different ways that we can integrate Unity's Analytics tools into our projects.

This chapter already assumes that you have Unity analytics activated. If you haven't done so yet, refer to Chapter 5, Advertising with Unity Ads, to learn how to create an account.

Your objectives

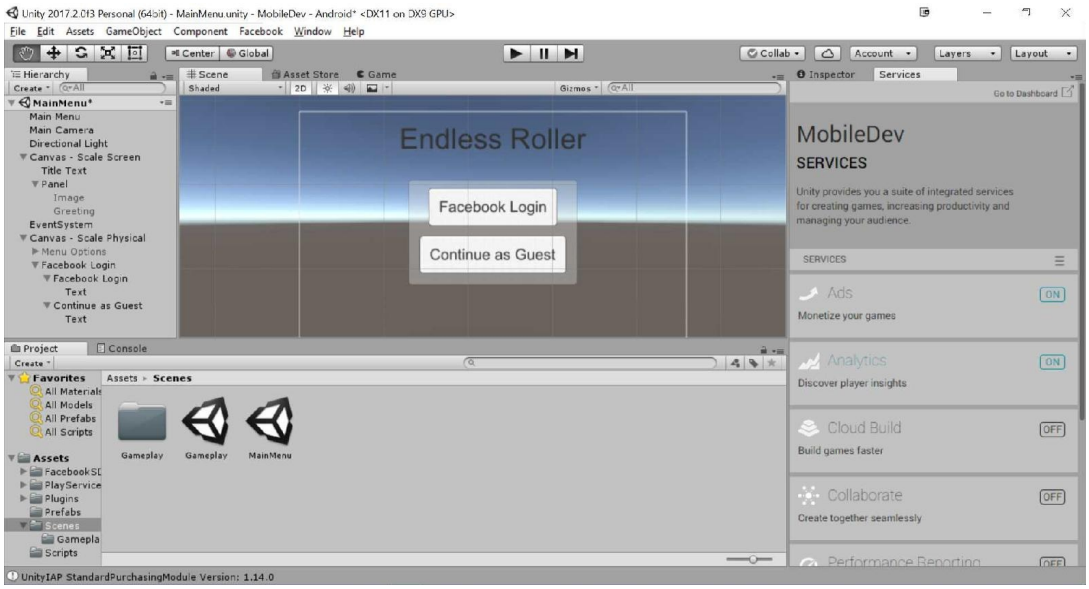
This chapter will be split into a number of topics. It will contain a simple step-by-step process from beginning to end. Here is the outline of our tasks:

- Setting up Unity analytics
- Tracking custom events
- Working with the funnel analyzer
- Tweaking properties with remote settings

Setting up analytics

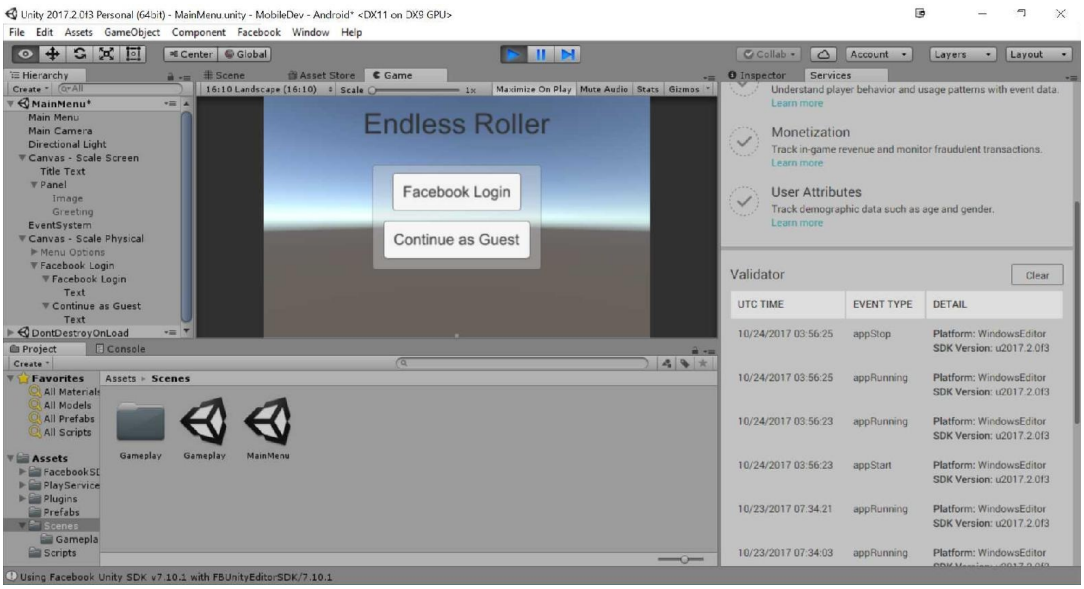
Although we activated Analytics in order to use Unity's Ads system, we didn't really dig into the system itself. Let's finish the setup for that now:

1. From the Unity Editor, open the Services tab (shown in the top-right side of the next screenshot) by either selecting it or going to Window | Services.
2. From there, scroll down and click on the Analytics button:



> As long as Analytics is enabled, the Editor sends an App Start event to the Analytics service when we press the Play button to start the game.
>
> The nice thing about this is we can ensure that this works correctly without having to export our game.

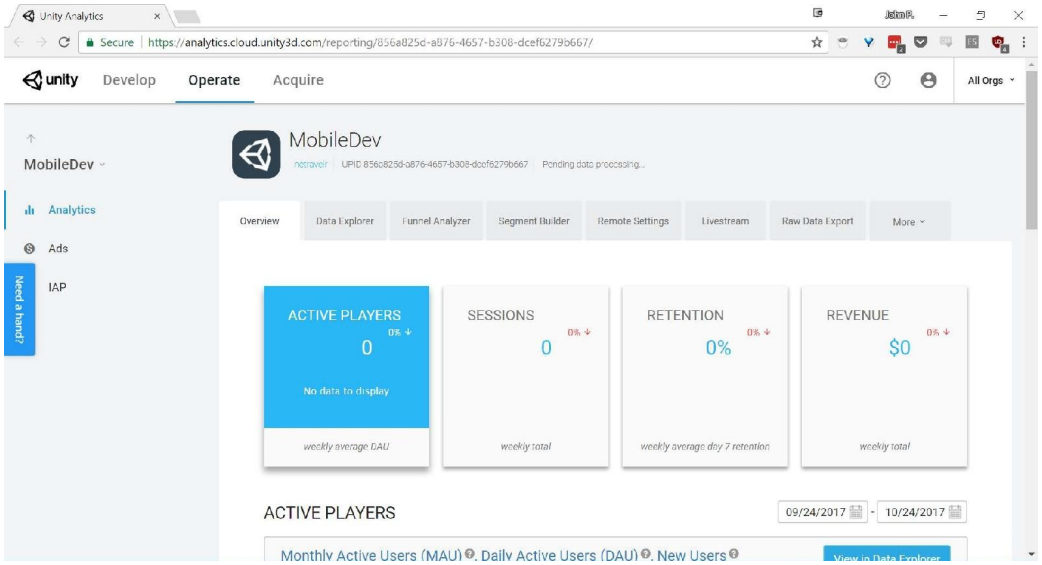
3. Press the Play button on the game, then go ahead and scroll down on the Analytics tab; you'll note a new menu called Validator, which shows the events being sent to Unity's analytics tool:



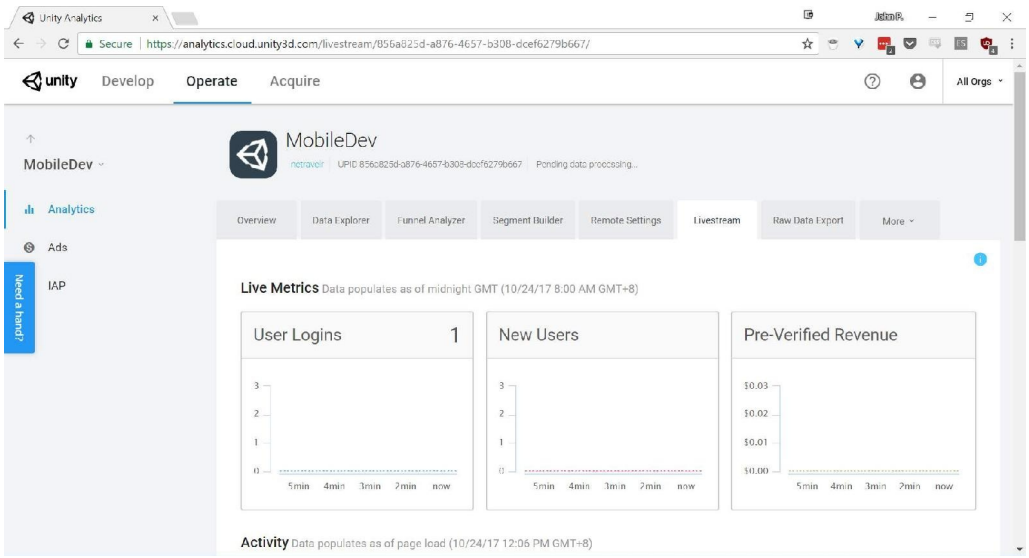
If you are not able to view anything, ensure that you are using a supported platform, such as Android or iOS.

4. You can also see that people are playing the game via the Unity analytics website-- for this, click on the Go to Dashboard button, which can be found on the top-left side of the Services tab.
5. From there, you may be taken to a welcome screen. Go ahead and click on Agree.

6. Afterward, you'll be led to the following menu:



7. From there, let's click on Livestream. This information updates as soon as it receives data from our game from events that have been sent:



You'll note from the preceding screenshot that I currently have a User Logins number of 1 because I've run the game from the Editor. If I run it from my mobile device also, that would increase to 2.

With that, we now know that the analytics information is being sent and received.

Tracking custom events

Unity Analytics does a number of different things automatically to make it easy to work with; however, as a game designer, you may often want to check whether certain aspects of the game are being used or whether players are reaching certain pieces of content. To keep a track on this, we can make use of the **Custom Events** system.

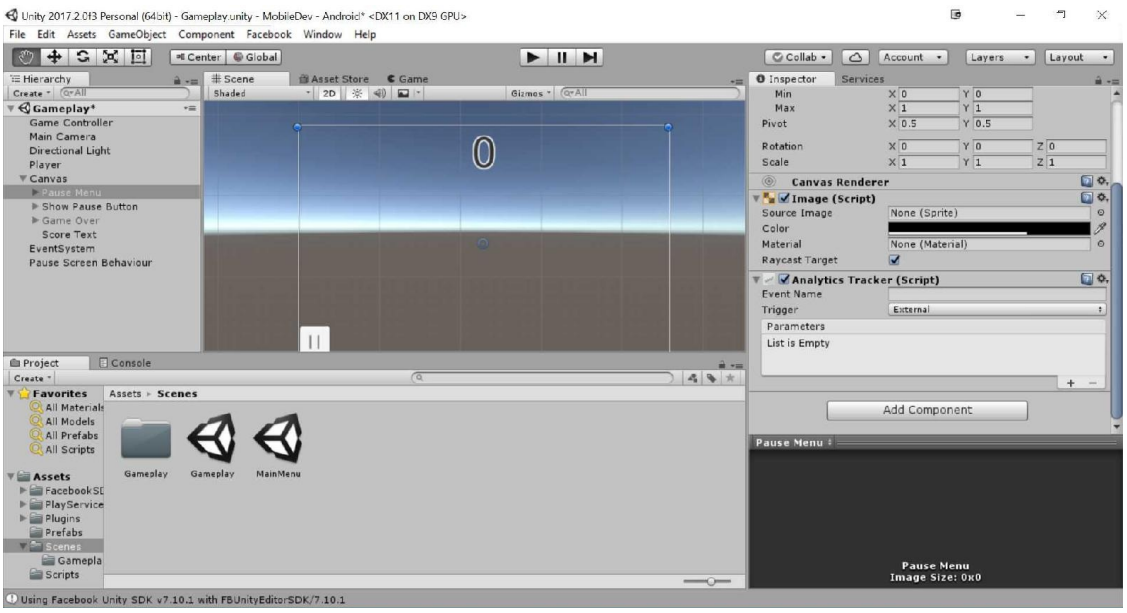
Custom Events are pieces of data that users send to the cloud as they play the game. Each custom event can have its own parameters, which will allow us to filter the data that we send when it occurs.

Using the AnalyticsTracker component

Now, one of the simplest ways to track when something occurs is through the editor itself, making use of the Analytics Tracker (Script) component. For example, perform the following steps when you want to track whether players are using the Pause menu.

1. Open the Gameplay scene by going to the Project window and then to the `Assets\Scenes` folder. From there, double-click on the Gameplay scene.
2. From the Hierarchy window, expand the Canvas object. From there, select Pause Menu.

3. From the top bar, select Component | Analytics | AnalyticsTracker to add the component to our selected game object:

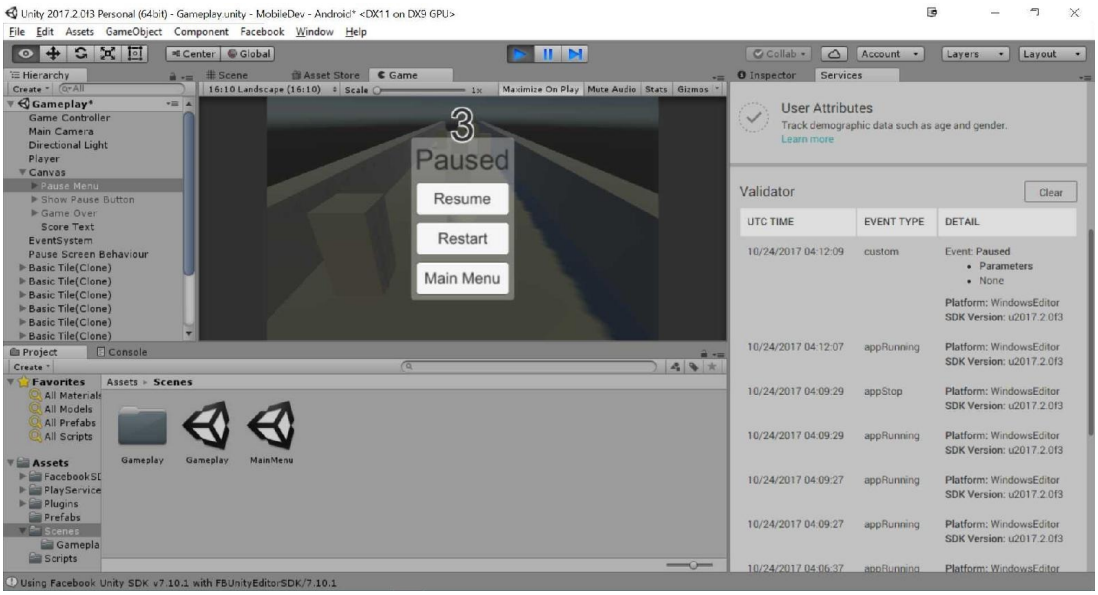


4. Next, we will need to come up with an Event Name to use. In this case, I used `Paused`. Then, under Trigger, select the dropdown and the On Enable option.

This means that if we turn on the pause menu (or rather when it is

enabled), an event will be sent. Note that this is different from the `On Start` event due to that event only happening once when it is initially turned on.

5. Now, start the game and then pause the game. If you go to the Services tab and open up Analytics, you should note from the Validator that a new event has been added:



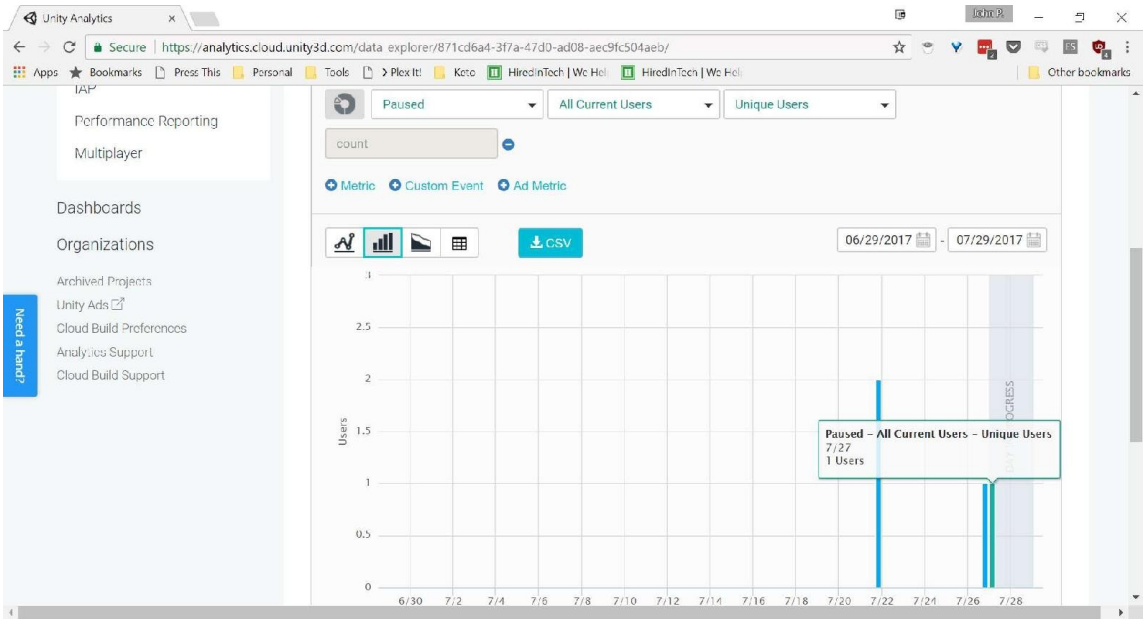
In addition to seeing this data in the Services tab, we can also see it in the dashboard.

6. Go ahead and scroll up in the Services tab and select the Go to Dashboard button on the top-right portion of the tab. From there, click on the Data Explorer option.
7. From the Data Explorer tab, you'll see a section for Metrics and Custom Events. You'll also see a chart that shows the DAU (Daily Active Users) that have been playing your project. We can use this menu to observe whenever the Paused event is called.

If you have just created the event, it may take up to 12 hours for the information to be received. Go ahead and check back later whether that's the case.

8. Click on the + button to the left of Custom Event to add a Custom Event to

this graph. Then, select the Custom Event dropdown and select Paused. Since we have just made the event, we won't see it in previous dates in Analytics, but we can see it a little easier if we click on the Column Chart button to change how the data is displayed:



Notice that when we scroll down we can see that the Data Explorer now shows that the Paused event has been called!

In addition to the calling event, you can also add parameters to the events in a similar manner to how Unity's UI system works. For more information on that and the Analytics Tracker component, check out https://docs.unity3d.com/Manual/UnityAnalyticsAnalyticsTracker.html.

Customizing events through code

It's great to have a way to do the events through the editor, but for those who are more comfortable working with code, it's also just as easy to do so.

One additional thing that we may want to track is how far players get before they lose. Let's take a look at how to do that now:

1. First, we will need to open up the `ObstacleBehaviour` script to modify what happens when the game ends.
2. To utilize Unity Analytics at the top of the file, we will add the following using declarations:

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

> The top option is obvious, but we are also adding `System.Collections.Generic` in order to get access to the `Dictionary` class, which we will use in the next piece of code.

3. Next, we will update the `OnCollisionEnter` function to the following:

```
void OnCollisionEnter(Collision collision)
    {
        var playerBehaviour =
        collision.gameObject.GetComponent<PlayerBehaviour>();

        // First check if we collided with the player
        if (playerBehaviour != null)
        {
            // Destroy the player
            collision.gameObject.SetActive(false);

            player = collision.gameObject;

            var eventData = new Dictionary<string, object>
            {
                { "score", playerBehaviour.Score }
            };

            Analytics.CustomEvent("Game Over", eventData);

            // Call the function ResetGame after waitTime has passed
            Invoke("ResetGame", waitTime);
        }
    }
```

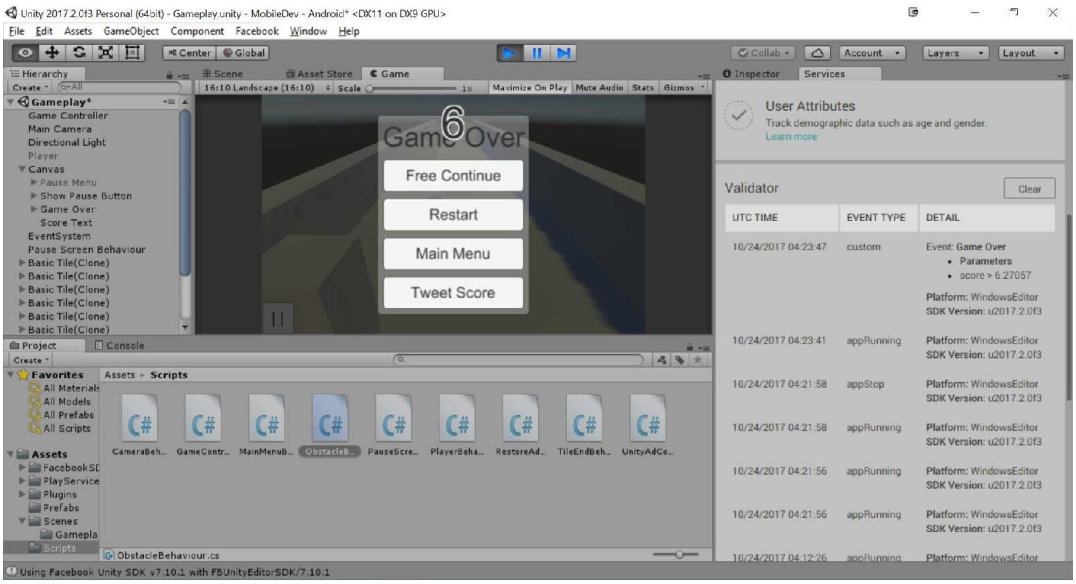
We've done a number of things within this script. To start off with, we have rewritten our check for the player to use the component as a variable now so that we don't have to call `GetComponent` again for the same thing.

Aside from that, the main addition is the calling of the `Analytics.SendEvent` function. This function takes in two parameters. The first is a string, which is the name that you wish for the event to have. The second parameter (which is optional) is a dictionary, which we haven't discussed yet.

A **dictionary** is a class that represents a pair of keys and values. The key is an identifier of some sort, which allows us to have a reference to obtain the value. This is most often used with strings as the key so that you can refer to some other data type.

For more information on dictionaries, check out http://csharp.net-informations.com/collection/dictionary.htm.

4. Save the script and return to the Unity Editor.
5. Play the game and lose it. You will note in the Validator that now you are sending a Game Over event with your score value:



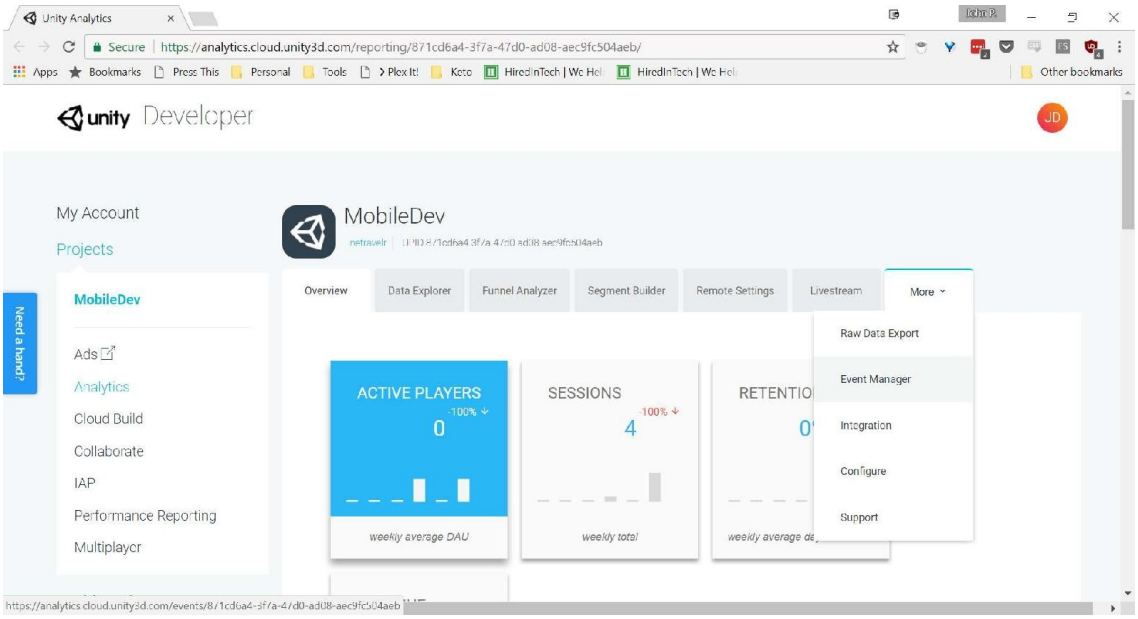
You may also dive into the Dashboard to see the information as well,

but it may take up to 6 hours before it will be visible. Events typically take a few hours to cycle. You will see them instantly on the Validator (to help you confirm that your code is working), but they don't populate into analytics until backend calculations have been processed on Unity's end due to all of the events they recieve.

6. After you've waited, go ahead to the Analytics tab and click on the Go to Dashboard button once again.

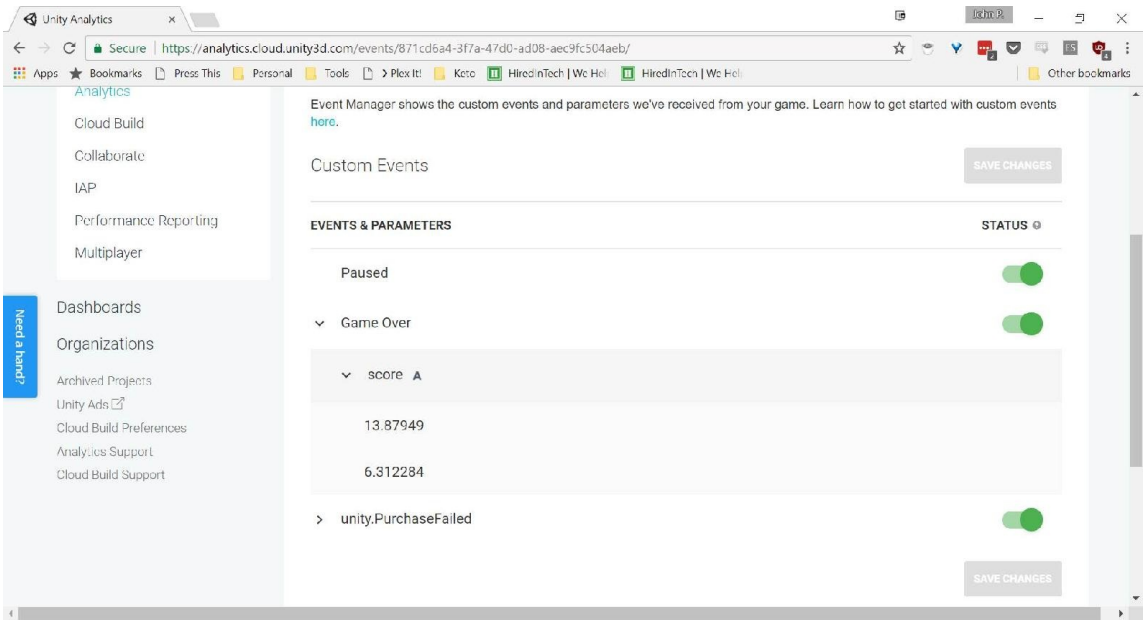
   We can use the Data Explorer like previously, but, in our case, we may just want to see all of the data from times it was called. For that, we can make use of the Event Manager.

7. Once there, go to the More tab and click on it. From the dropdown that pops up, click on Event Manager:



   The Event Manager is the location where users can see the custom events that were received from users playing the game as well as the parameters that were passed to them.

8. Scroll down to the Custom Events header--you'll see a list of all of them under it, with the Paused option and the Game Over event created in this section:



If you click on the dropdown, you should be able to see the score property, and clicking on that will show all the values received whenever it was called.

Instead of a single value, we can pass up to 10 parameters into the dictionary. However, the value must be one of the following types:

- `bool`
- `string`
- `int`
- `float`

Remember that you can always convert an object to a string in C# using the `ToString` function.

You can only send 100 custom events per hour per user, so you should not be doing too many custom events in the game. I suggest that you create custom events for whenever a user reaches an important milestone, for example, when they level up or if they make an **In-App Purchase** (**IAP**).

For more information on `Analytics.CustomEvent` and other ways it can be called, check out https://docs.unity3d.com/ScriptReference/Analytics.Analytics.CustomEvent.html.

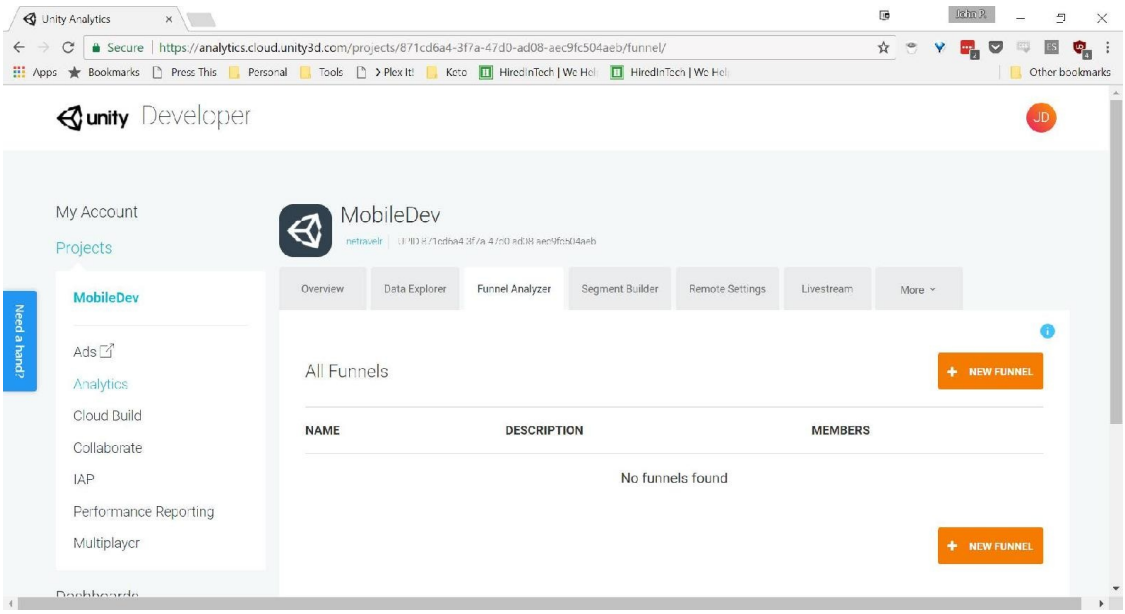
Working with the funnel analyzer

One of the many things we'd like to know about our players is how they are actually playing the game. Are users skipping our tutorial? To keep track of how players go through a series of events, we have funnels. Funnels help us to identify where drop-offs happen in your game.

If you happen to see a large number of people not getting to a certain step, you can assume that something that happened in the preceding step is causing people to stop playing your game.

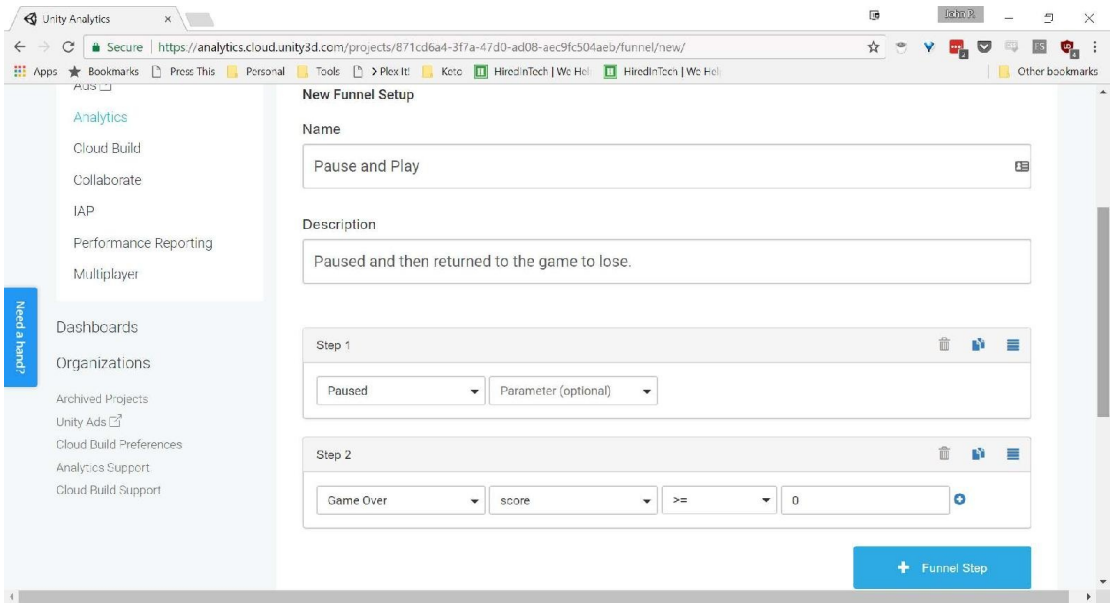
Funnels are based on the concept of custom events that we used in the previous two sections of this chapter:

1. From the Dashboard, select the Funnel Analyzer section:

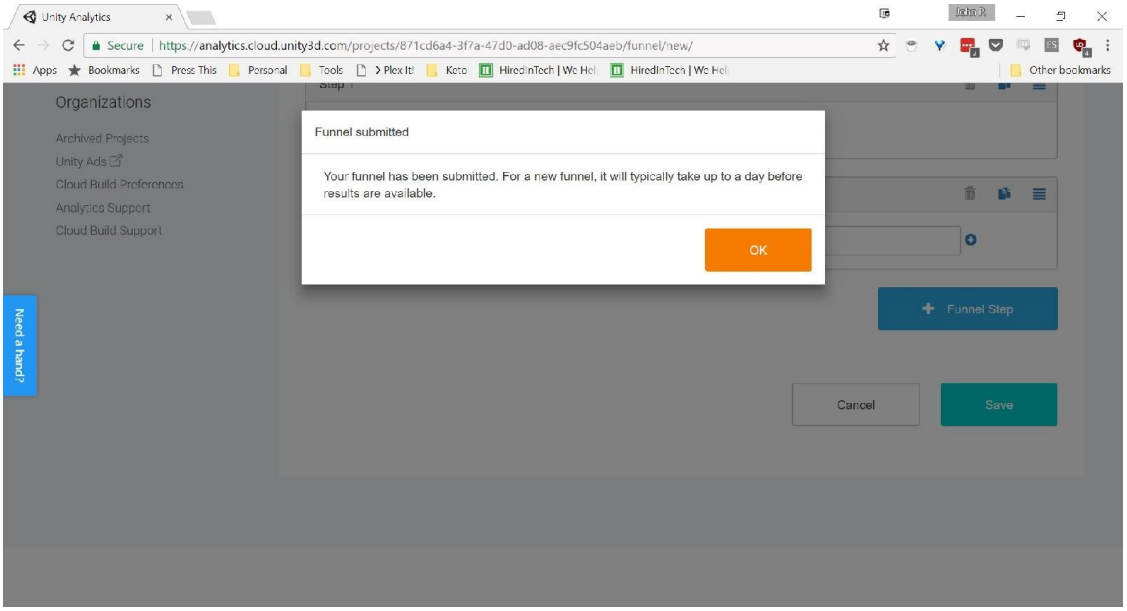


Right now, there are no funnels set up, so we should create one.

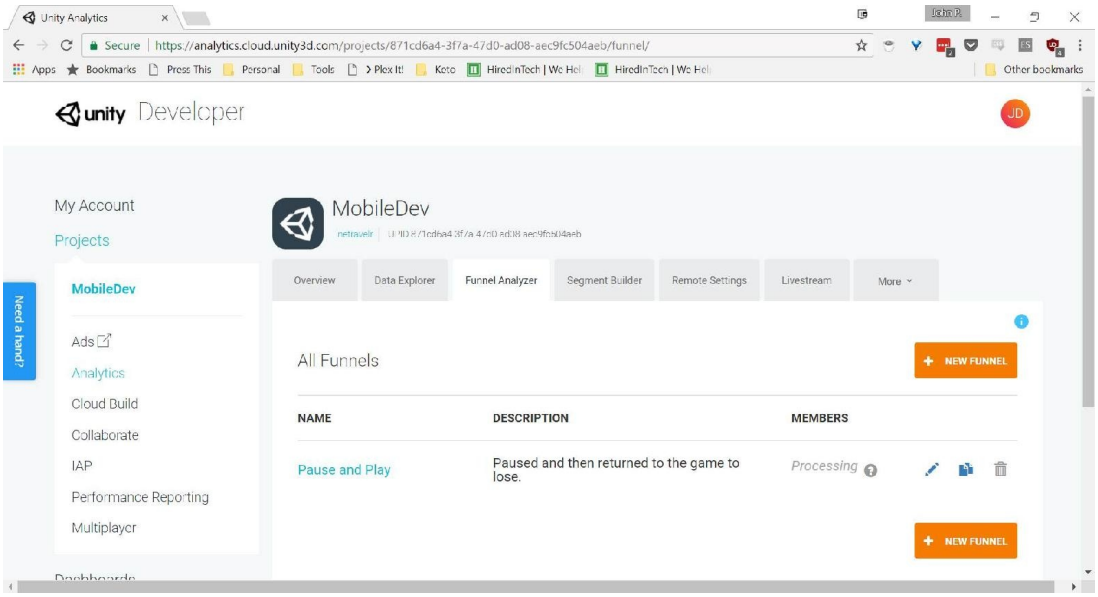
2. Click on the New Funnel option:



3. Next, scroll all the way down and then click on the Save button:



4. You'll get a notice saying that the funnel has been submitted, but it may take up to a day before we can see the results (at least 10-12 hours). Go ahead and click on OK.
5. Now, go back into the Unity Editor and play the game a couple more times and ensure that you pause the game before failing to fill up some test data for us to check the next day:



6. You should be able to select that funnel, and it will provide information of all the times it has been called.

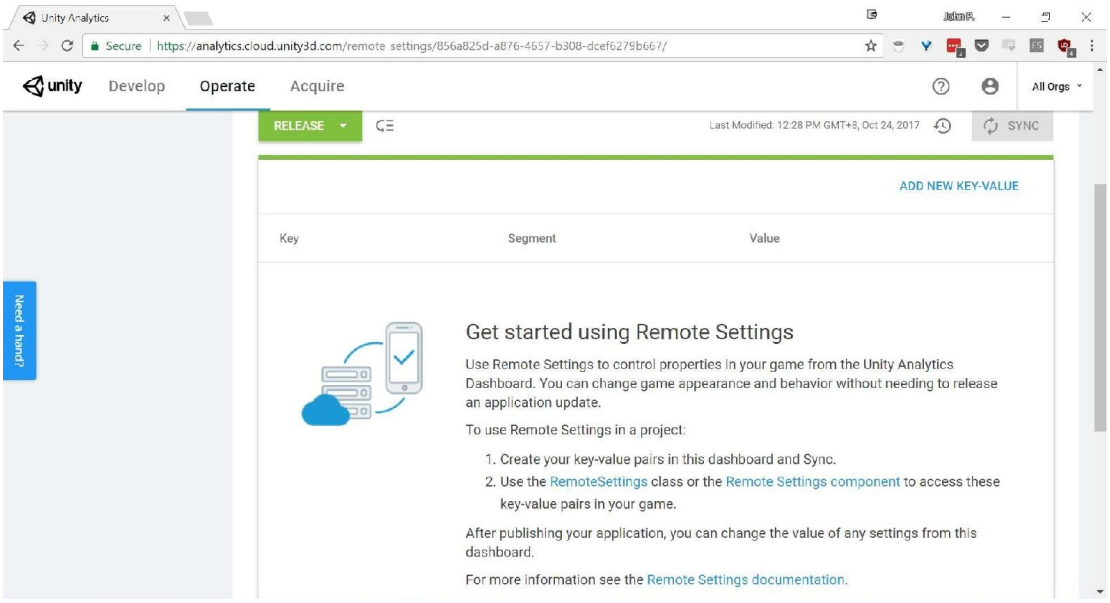
Tweaking properties with remote settings

Getting a new build of your game exported can take quite a bit of time. It takes time to actually make the changes in the editor, then you have to export the game and upload a new version on each of the App Stores you are targeting. Then, you have to spend time waiting on them to approve the app and for everyone to actually download it.

One of the things I talk to my students about is creating projects that can be easily changed without having to open up the Unity Editor. This could be done using data-driven development practices--such as building levels or encounters using text files, Asset Bundles, or Unity's Remote Settings menu--to allow us to instantly modify variables in copies of the game that are already out.

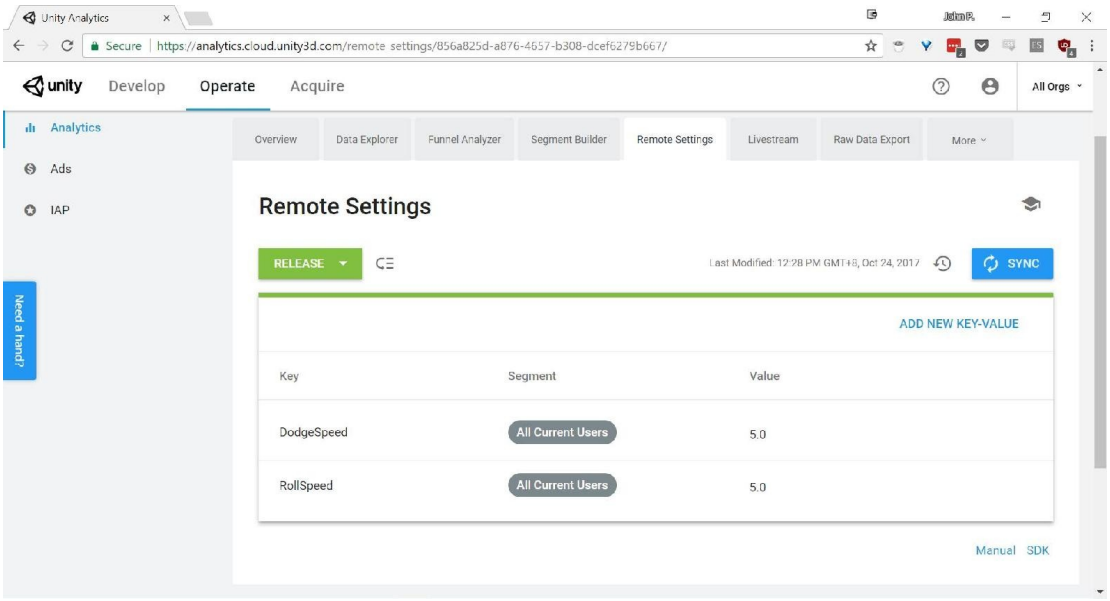
One of the things we may want to be able to update is tweaking the difficulty of our game by changing the speed at which the player moves. So, let's take a look at how we can do that now:

1. The first thing we will need to do is create the variables that we would like to change. Open up the Analytics dashboard and go to the Remote Settings tab:

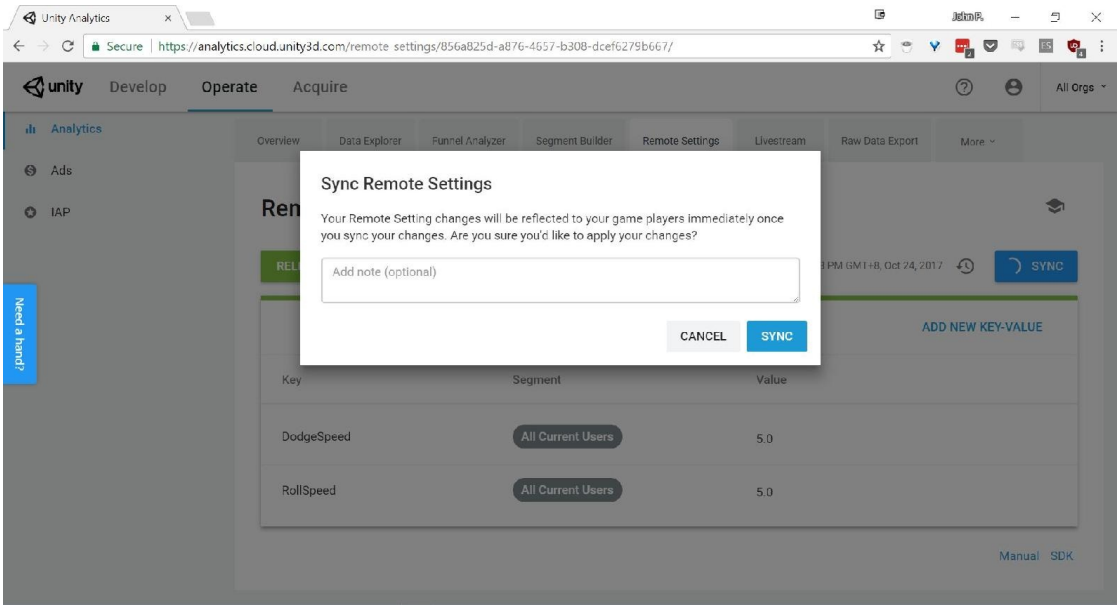


This tab is the location where we can set and modify the values. Just like working with dictionaries, remote settings are key-value pairs, and there are two configurations that can be used: Release or Development. Release is used by computers and devices running regular builds of your game. Development is the mode used by playing the game in the Unity Editor as well as any builds created with the Development Build property set to true from the Build Settings window.

2. Click on the Add New Key-Value button at the top-right corner. Under the Enter Remote Setting Key property, type `RollSpeed`. Select Float under the Select Type dropdown and put `5` in the Value field. Finally, click on the Save button.
3. Then, let's do the same thing for the `DodgeSpeed` variable with a value of `5`:



4. It's important to note that this doesn't actually make the change. Note how there is a big blue button that says Sync. Click on that and then the changes will be deployed. It'll pop up a window asking whether you want to confirm the changes:

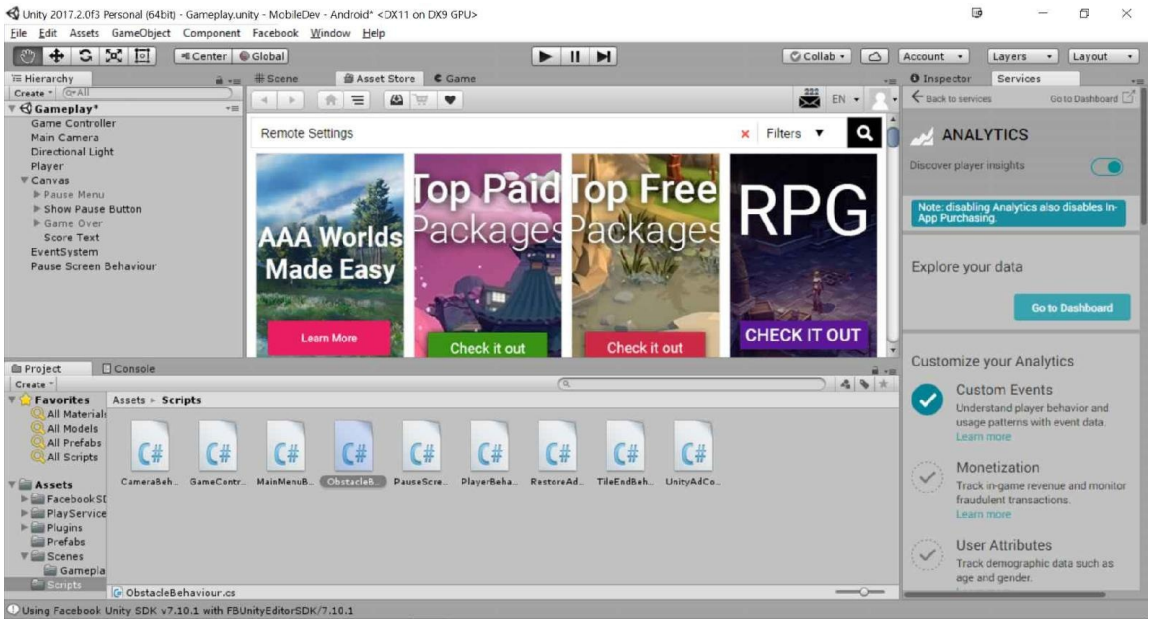


5. Go ahead and then click on Sync.
6. Now that we have some values to grab, let's take a look at how we can actually do that. Head back into Unity Editor.

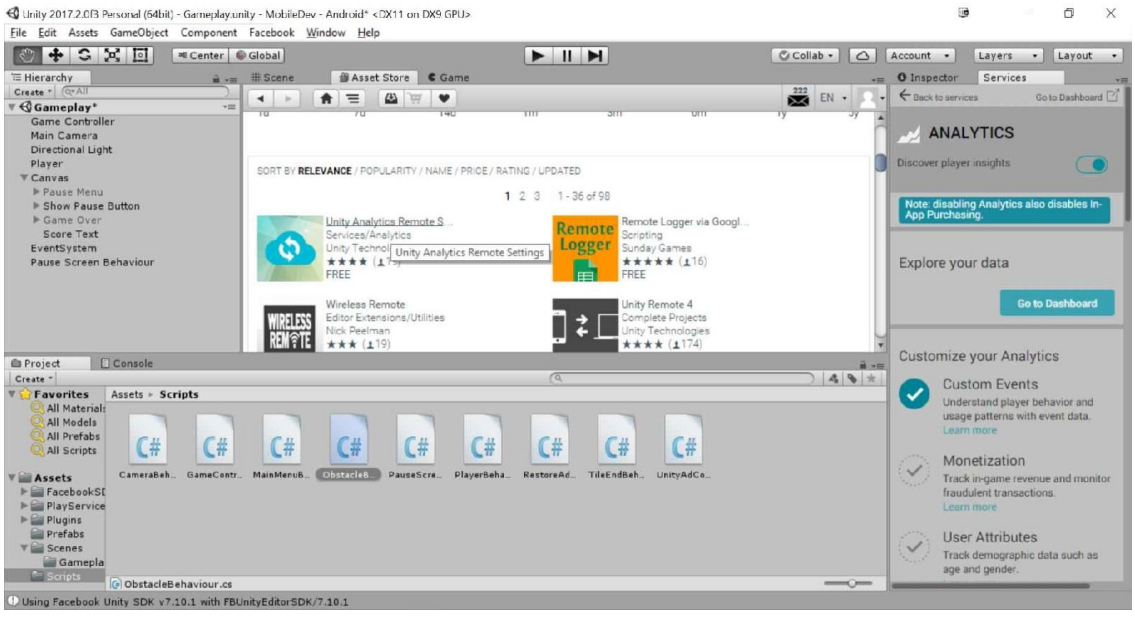
7. In order to use the Remote Settings, we will need to download and import

the Remove Settings package. To do this, go to Window | Asset Store (or press *Ctrl+9*).

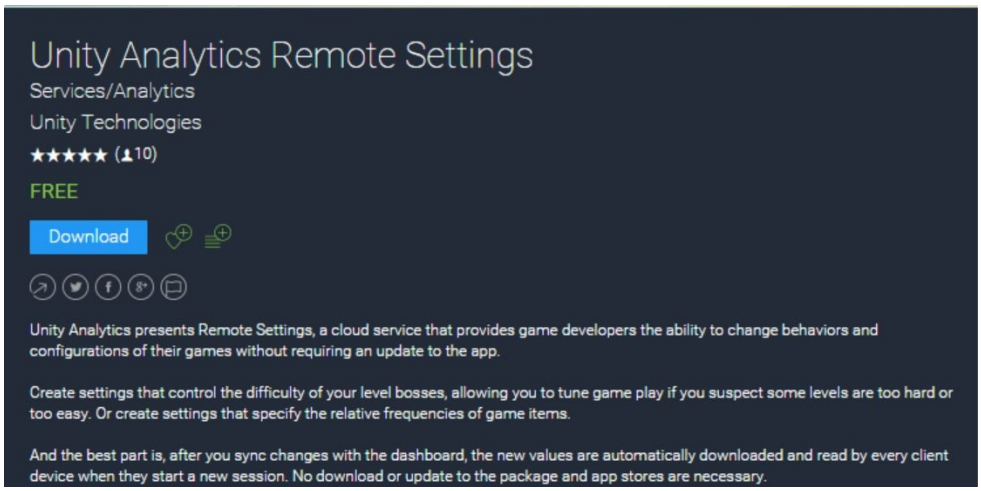
8. From the Asset Store, click on the Search bar and type in `Remote Settings` and press *Enter*:



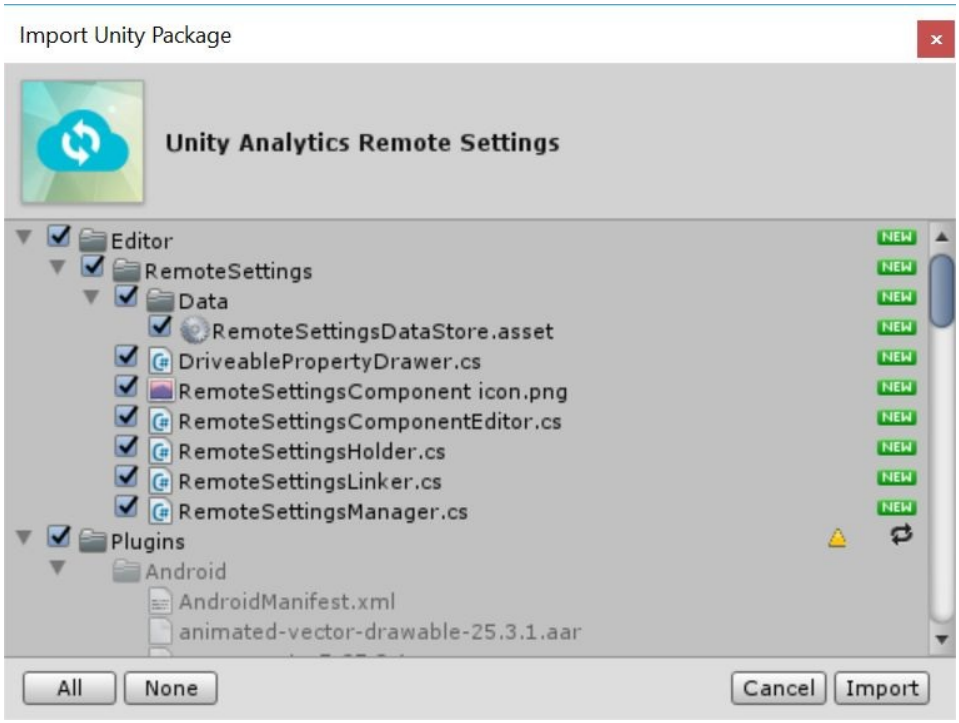
9. From there, click on the first selection of the Unity analytics Remote Settings option:



10. Next, click on the Download button to add it to our project:

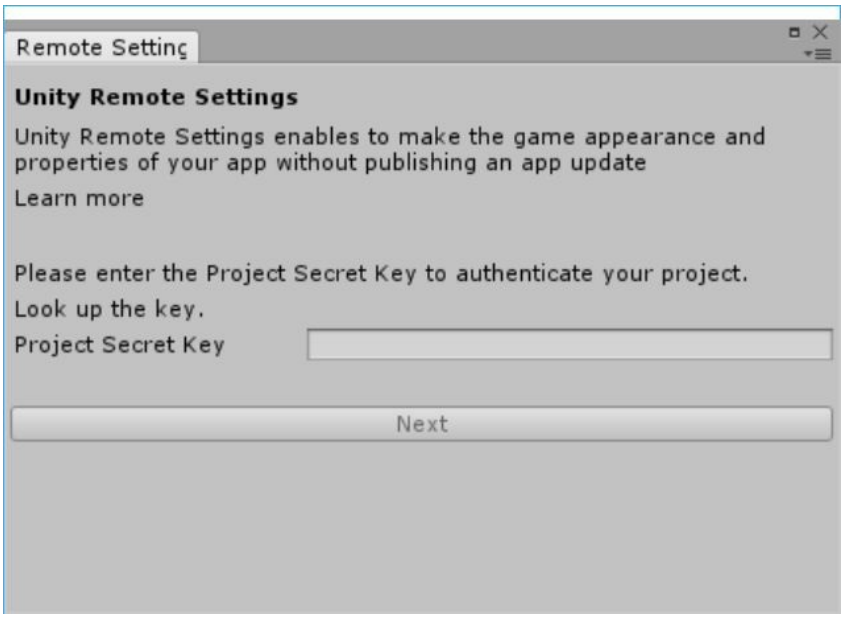


11. Upon reaching the Import Unity Package dialog, go ahead and click on Import with everything selected:



I deleted the `MiniJSON.cs` file that was included with the Remote Settings due to it already being defined within Unity's IAP scripts. This should be fixed by the time the book is published, but I am including it just in case it shows up.

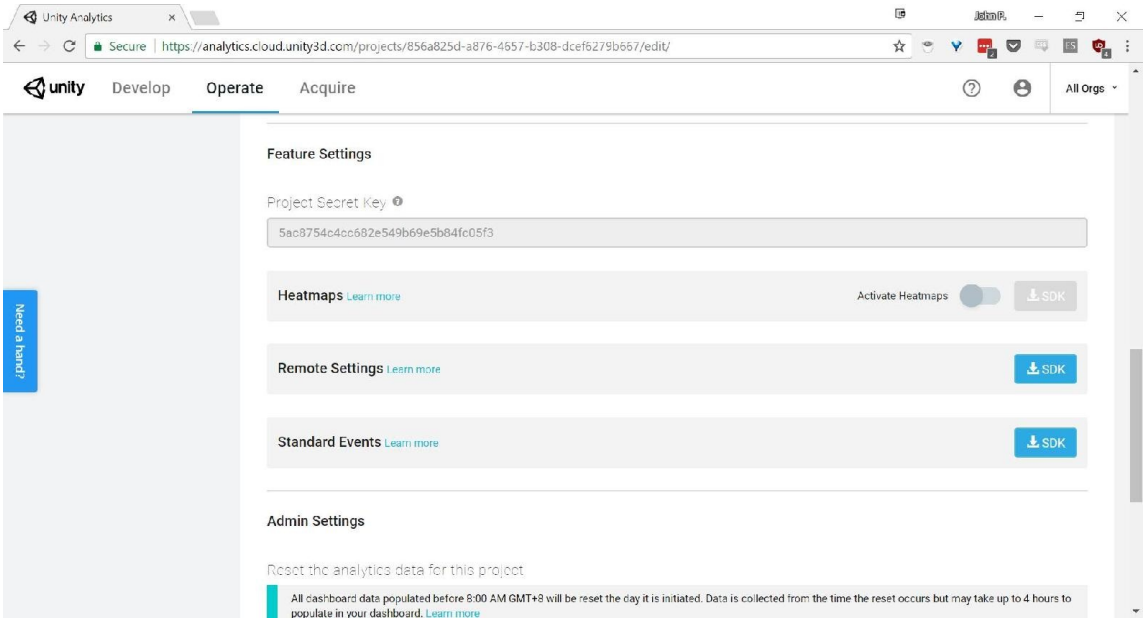
12. Now that we have it imported, we need to enable it. To do that, we will need to go to Window | Unity Analytics | Remote Settings:



As you can see in the preceding screenshot, we need to plug in the project's secret key in order to make use of these features. To do this, we will need to once again go to the Dashboard.

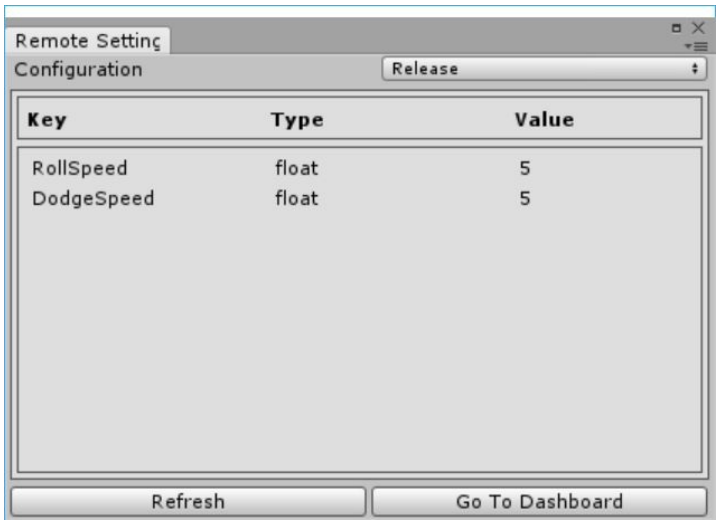
13. From the dashboard, go to the More dropdown and select Configure.

14. Scroll down, and you should see the Project Secret Key property under Feature Settings:

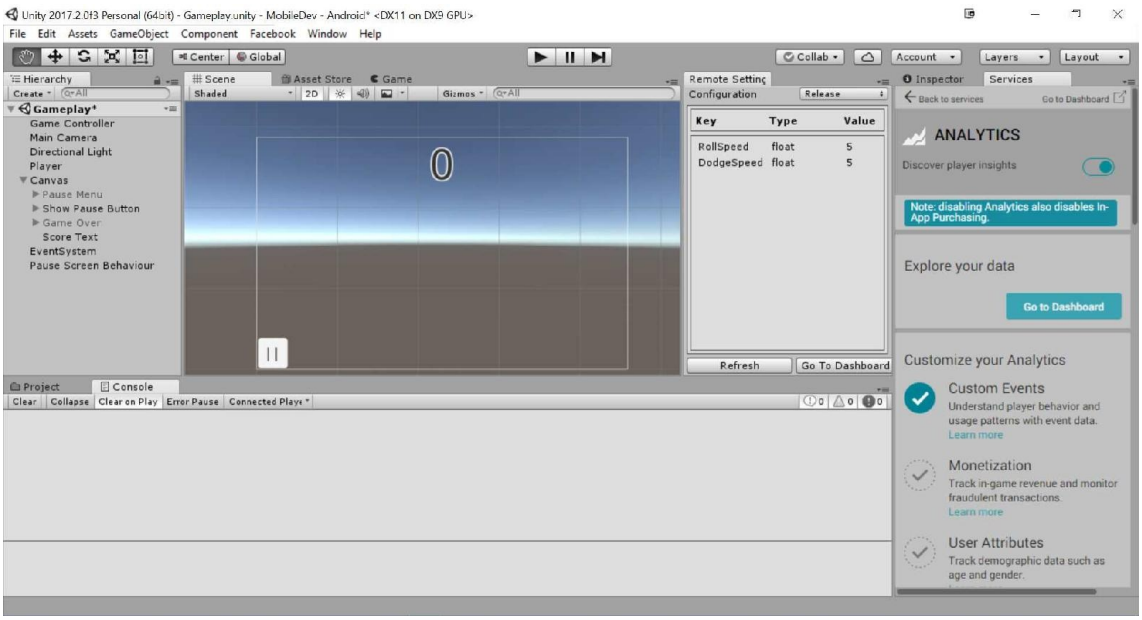


You should keep the value of the Project Secret Key a secret, as it allows others to have access to your project.

15. Copy this value and then go back into Unity Editor, paste it into the slot and then click on the Next button. If all goes well, you should see the menu change:

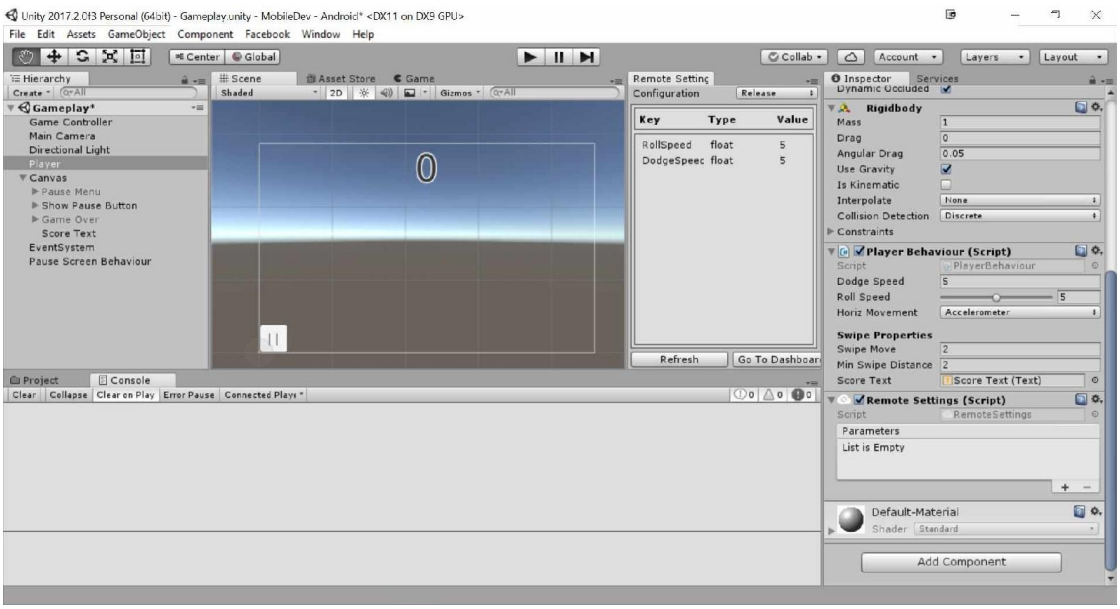


16. To make it easier to see, I'm going to drag the Remote Settings tab and drop it to the right of my Scene tab:



Now it's a lot easier to look at and work with.

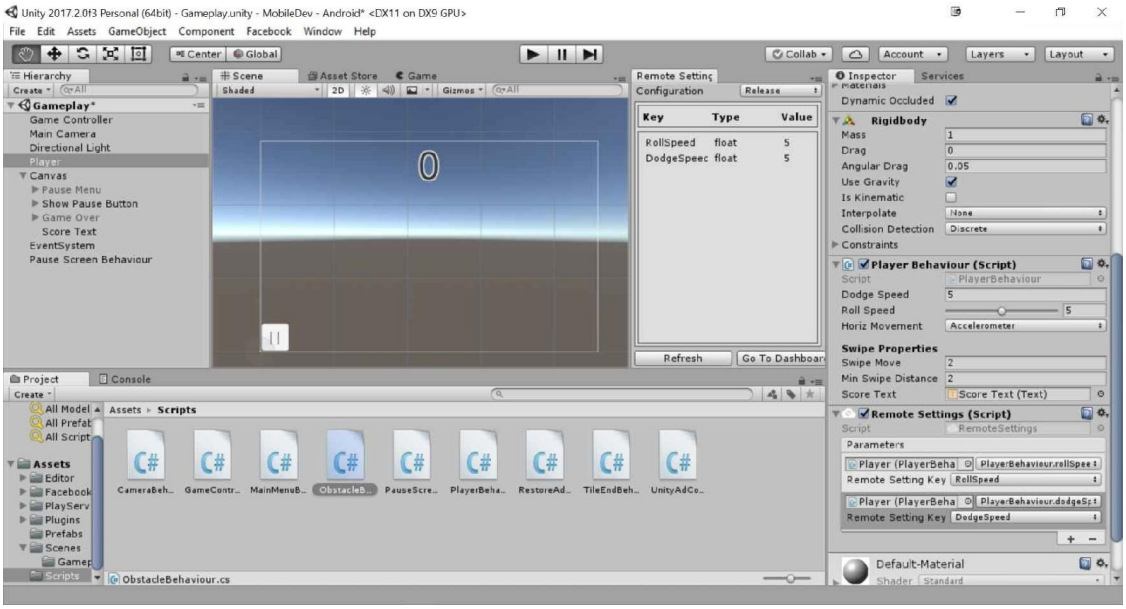
17. Again, this doesn't do anything yet, so let's fix that next.
18. Open up the Gameplay scene if it isn't opened up already and select the Player object and the Inspector tab.
19. Next, go to Component | Analytics | Remote Settings to add the Remote Settings component to the object:



20. From there, click on the + button to add a new parameter for the component.
21. Drag and drop the Player game object from the Hierarchy tab into the Object section. Then, for the No field property, select Player Behaviour | rollSpeed. Then, in the Remote Setting Key, select Roll Speed from the dropdown.

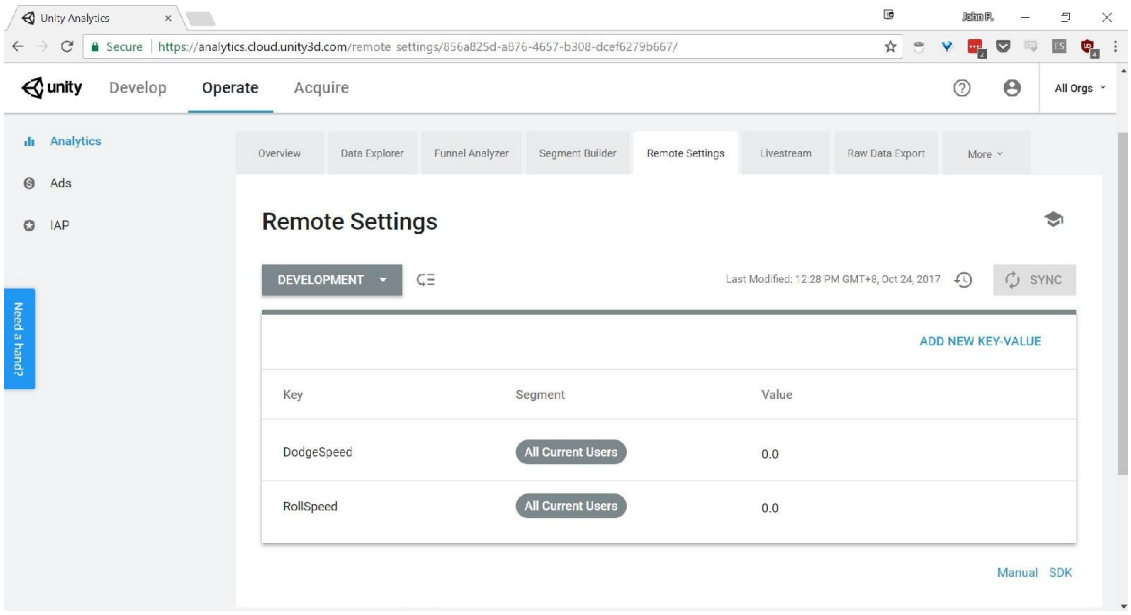
    This makes the `rollSpeed` variable in the `PlayerBehaviour` class set to the Roll Speed value in the Remote Settings menu.

22. Then, do the same thing for the `DodgeSpeed`:

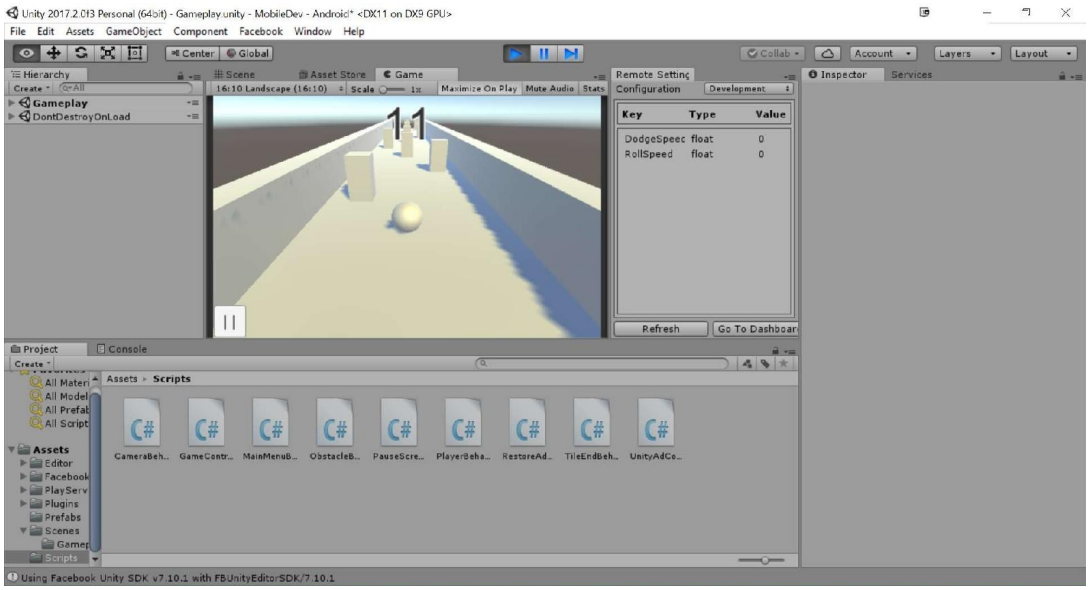


This will work correctly if we were to export our game, but currently we do not have variables set up for the Development configuration. This means that when playing the game in the editor, nothing will change.

23. Dive back into the dashboard, and go to the Remote Settings tab. Under the dropdown, change the Release value to Development and add in `DodgeSpeed` and `RollSpeed` again, but give them both a value of `0`. Finally, click on the Sync button to update the values:



24. Next, dive back into the Editor, and in the Remote Settings property, click on the Refresh button. You will note that the Configuration property now has a dropdown where you can select something. Go ahead and select Development and play the game:



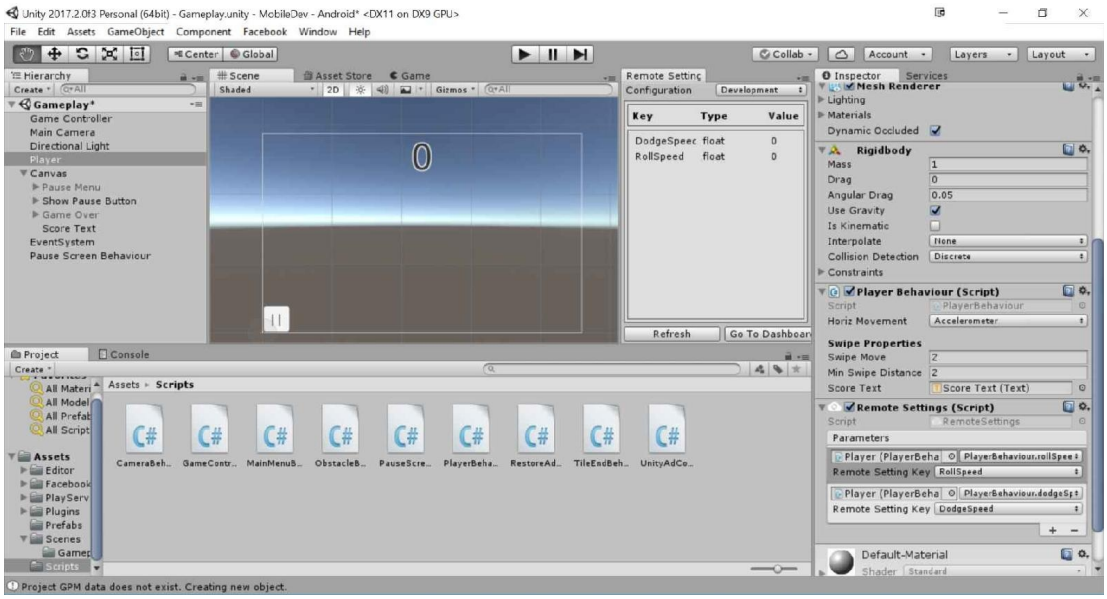
As you can tell, in this version of the game, the player cannot move at all due to how we set the properties. Now, you can test out the values and sync them in the Editor and modify them for release when they're ready.

25. Since the `dodgeSpeed` and `rollSpeed` variables are now being set via the Remote Settings component, we can now hide them from the Inspector. Replace their declarations so that the class looks as follows:

```
/// <summary>
/// Responsible for moving the player automatically and
/// receiving input.
/// </summary>
[RequireComponent(typeof(Rigidbody))]
public class PlayerBehaviour : MonoBehaviour
{
     /// <summary>
     /// A reference to the Rigidbody component
     /// </summary>
     private Rigidbody rb;

     /// <summary>
     /// How fast the ball moves left/right
     /// </summary>
     [HideInInspector]
     public float dodgeSpeed = 5;
```

```
    /// <summary>
       /// How fast the ball moves forwards automatically
    /// </summary>
    [HideInInspector]
    public float rollSpeed = 5;

public enum MobileHorizMovement
{
    Accelerometer,
    ScreenTouch
}

    /// Rest of the file here
```

26. Save the script and go to Inspector:



Now, the values will be set through the Remote Settings component, and users won't be confused about why their values are being replaced from what's in the Player Behaviour.

There's a lot more you can do with Remote Settings. You can learn more about the Remote Settings component and how to use it for working with non-default parameters at https://docs.unity3d.com/Manual/UnityAnalyticsRemoteSettingsComponent.html.

Summary

In this chapter, we explored a number of ways that we can make use of Unity's Analytics tool to make our games better: from how to tell what our players are doing to learning how we can adjust our game based on that feedback without users having to download an entire new copy of our game.

With this, we have all of the implementation details of our game completed, but our game right now is pretty barebones. In the next chapter, we will look into ways to make our game more polished using features such as particle systems and screen shake.

Making Your Title Juicy

We now have a basic game, but it's just that...basic. In this chapter, you will learn some of the secrets that game developers use to take the basic prototype of their game and turn it into something with a lot of polish that feels satisfying to play, which is known as making our games juicy.

Also known as *game feel*, juicyness is a kind of a catch-all term for all the things that we do in a game to make it pleasing for its users to interact with. This is something that is done with most mobile games that are out there today, and lacking this kind of interactivity will make others believe our project is lacking in polish.

Chapter overview

In this chapter, you will learn some of the different ways that you can integrate features of juiciness into our projects.

Your objectives

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

- Working with Tweens
- Working with materials
- Using post-processing effects
- Adding particle effects

Animation using iTween

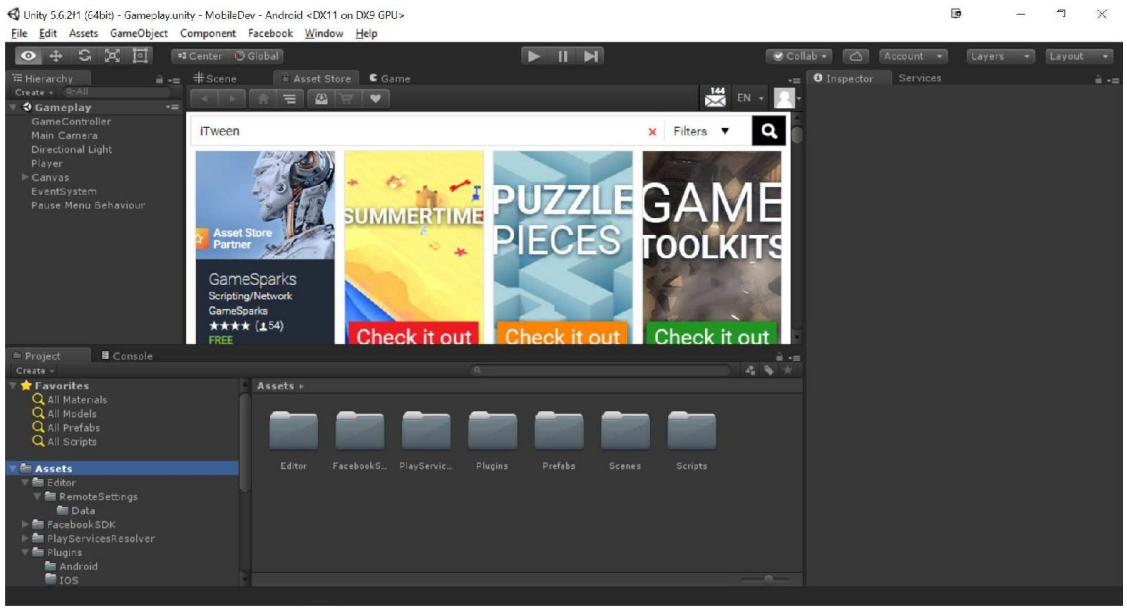
Currently, our game's menus are completely static. This is functional, but does not make players excited about playing our game. To make the game seem more alive, we should animate our menus. Being able to use Unity's built-in animation system is great, and it can be quite useful if you want to modify many different properties at once. If you don't need precise control, if you're only modifying a single property, or if you want to animate something purely via code you can also make use of a tweening library. If it is given a start and an end, the library will take care of all the work in the middle to get that property to that end within the time and speed you specify.

One of my favorite tweening libraries is PixelPlacement's iTween, which is open source and usable for free in commercial and noncommercial projects.

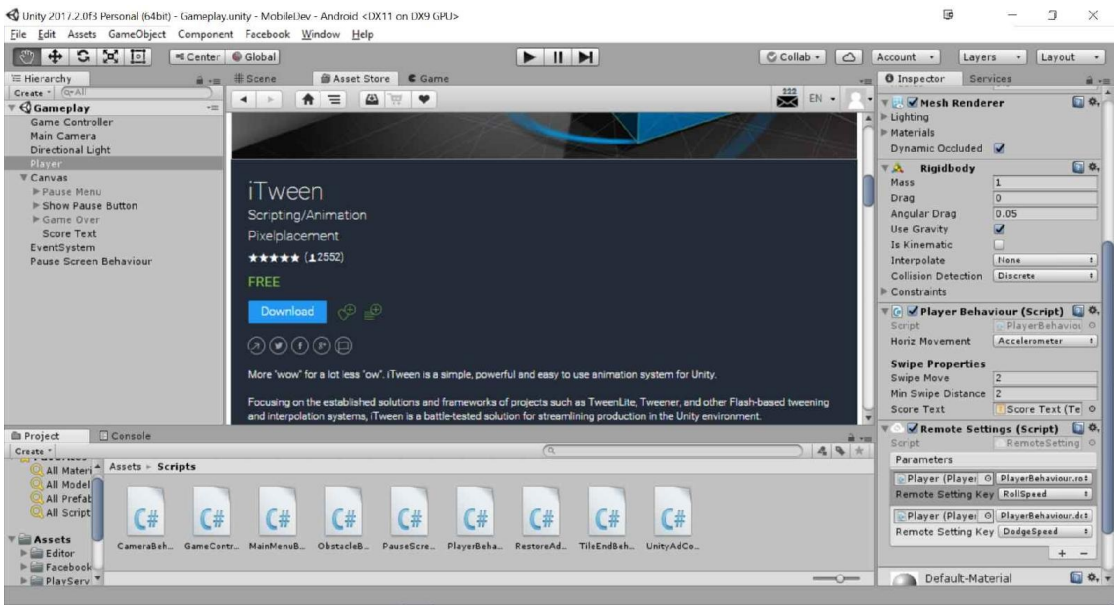
iTween setup

Now that we know we want to add tweens to our project, let's start off by actually adding it to our project:

1. Open up the Asset Store tab by going to Window | Asset Store. Type in `iTween` in the search bar at the top and then press Enter:

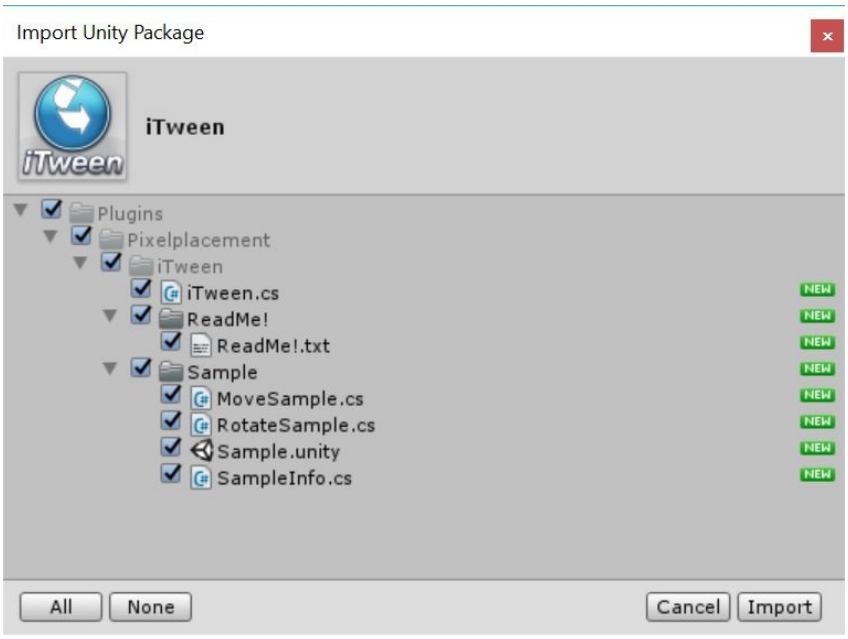


2. If the Asset Store tab is too small for you, like it is here, feel free to drag it out of the middle section and resize until it looks nice for you.
3. From there, you'll be brought to a list of items with the first one being iTween; select it, and you will be brought to iTween's product page. Click on the Download button:

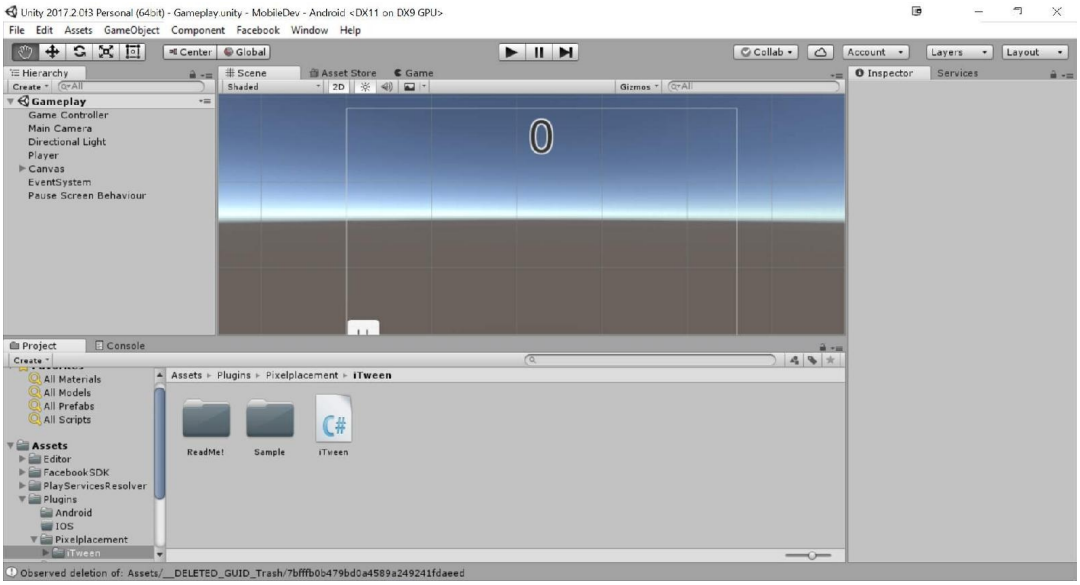


4. From there, you'll be asked to log in to your Unity account, which you created when you installed Unity. If you don't have one, feel free to click on Create Account and do so. Once logged in, click on Download once again, and if it doesn't happen automatically, click on the Import button.

5. You should see an Import Unity Package window pop up; from there, you can check or uncheck whatever files you want to keep. We will just use `iTween.cs` here; however, the others may be useful to you, should you wish to use them on your own. Once you're finished selecting what you want, click on the Import button:



6. We don't need the Asset Store tab anymore, so go ahead and close it. You'll notice that now we have the files we have selected inside our Project tab in the `Assets/Plugins/Pixelplacement/iTween` folder:



With that, we now have iTween set up.

Creating a Simple Tween

Now that we have iTween included in our project, we can use it inside of our code; to do that, perform the following steps:

1. From the Unity Editor, open the MainMenu level by going to the Project window and double-clicking on the MainMenu scene.
2. Now, move to the `Scripts` folder and open the `MainMenuBehaviour` by double-clicking on it.

3. Firstly, we will add the following new function, which we will use:

```
/// <summary>
/// Will move an object from the left side of the screen
/// to the center
/// </summary>
/// <param name="obj">The UI element we would like to
/// move</param>

public void SlideMenuIn(GameObject obj)
{
   obj.SetActive(true);
   var rt = obj.GetComponent<RectTransform>();

   // Set the object's position offscreen
   var pos = rt.position; pos.x = -rt.rect.width; rt.position = pos;

   // Move the object to the center of the screen
   iTween.MoveTo(obj, iTween.Hash("x", Screen.width / 2,
                                  "easeType", "easeInOutExpo",
                                  "time", 1.5f ));
}
```

Before we move anything using iTween, we will first set the position of our object (the `obj` parameter) off-screen by setting the `x` position. It's important to note that when dealing with UI elements in Unity, by default, we are dealing with Screen space, which as you can recall from Chapter 3, *Mobile Input/Touch Controls*, means means that the position (0,0) is the bottom left of the screen and (`Screen.width`, `Screen.height`) is the top right.
From here, we'll see that we are calling the `MoveTo` function from iTween. This takes in two parameters, the first being the object we wish to move and the second being a `Hash` or hash table, which is a data structure that creates an associative array such that certain keys

will be mapped to certain values. In iTween, their implementation takes in sets of twos with the first being a property and the second being the value we want it to be.

For more information on hash tables, check out *http://en.wikipedia.org/wiki/Hash_table*.
For more information on getting started with iTween, check out *http://itween.pixelplacement.com/gettingstarted.php*.

4. Now that we have this function, let's actually call it. In the `Start` function of the `MainMenuBehaviour` script, change it so that it now looks as follows:

```
virtual protected void Start()
{
    // Initialize the showAds variable
    UnityAdController.showAds = (PlayerPrefs.GetInt("Show Ads", 1)
                                 == 1);

    if (facebookLogin != null)
    {
       SlideMenuIn(facebookLogin);
    }

    // Unpause the game if needed
    Time.timeScale = 1;
 }
```

The first thing we do is bring in the Facebook login menu to the screen by calling the `SlideMenuIn` function, which in turn will tween the menu to the center of the screen. iTween, by default, makes use of the game's `Time.timeScale` property to scale movement. When we leave the game from the pause menu and go back to the main menu, the game will still be paused. This ensures that the game will be unpaused by the time we want to slide this menu in. When we start building the pause menu, we'll see how we can make our Tweens work even when the game is paused.

5. While the code should work fine without this, iTween suggests that you call the `Init` function on all of the objects you'd like to move in the `Awake` function before it is used, so we can do that as well. Add the following to the end of the `Awake` function:

```
if(facebookLogin != null)
 {
      iTween.Init(facebookLogin);
 }
```

```
 if(mainMenu != null)
 {
      iTween.Init(mainMenu);
 }
```

6. Finally, we will next add the ability so that when we select a button to go to another menu we will have the current menu slide out:

```
/// <summary>
/// Will move an object from the center of the screen
/// to the right
/// </summary>
/// <param name="obj">The UI element we would like to
/// move </param>

public void SlideMenuOut(GameObject obj)
{
   var rt = obj.GetComponent<RectTransform>();

   // Set the object's position offscreen
   var pos = rt.position; pos.x = Screen.width / 2; rt.position = pos;

   // Move the object to the center of the screen
   iTween.MoveTo(obj, iTween.Hash("x", Screen.width + (rt.rect.width),
                                  "easeType", "easeOutQuad",
                                  "time", 1.5,
                                  "oncomplete", "OnHidden",
                                  "oncompletetarget", gameObject,
                                  "oncompleteparams", obj ));
}
```

Note that this is similar to the previously written function, except now we are also using some new parameters; let's also add in the `OnHidden` function that we are using in the preceding script to be called when the tween finishes:

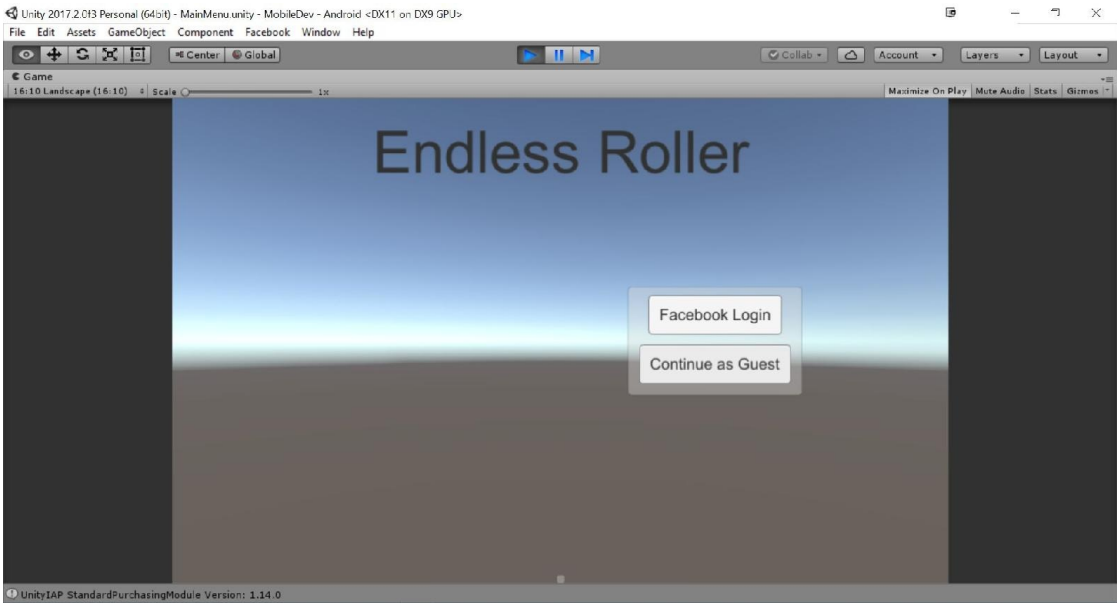
```
void OnHidden(object obj)
{
   GameObject go = obj as GameObject;
   if(go != null)
   {
      go.SetActive(false);
   }
}
```

7. Then, we will need to update the `ShowMainMenu` function to actually display the menus:

```
public void ShowMainMenu()
{
  if (facebookLogin != null && mainMenu != null)
  {
     mainMenu.SetActive(true);
     SlideMenuIn(mainMenu);
     SlideMenuOut(facebookLogin);
```

```
     // No longer needed as menus will be animating
     //facebookLogin.SetActive(false);
     //mainMenu.SetActive(true);

    if (FB.IsLoggedIn)
    {
       // Get information from Facebook profile
       FB.API("/me?fields=name", HttpMethod.GET, SetName);
       FB.API("/me/picture?width=256&height=256",
              HttpMethod.GET,   SetProfilePic);
    }
  }
}
```

8. Save the script and dive back into the game:



As you can see, the menus will not fly in and out when in the main menu.

You can find a list of all of the possible parameters you can pass in to t
functions at http://www.pixelplacement.com/itween/documentation.php.

Adding Tweens to the pause menu

Now that we have the main menu finished, let's continue doing this with the pause menu:

1. Go ahead and open up our Gameplay scene. Update the `PauseScreenBehaviour` script to have the following implementation of `SetPauseMenu`:

```
/// <summary>
/// Will turn our pause menu on or off
/// </summary>
/// <param name="isPaused"></param>
public void SetPauseMenu(bool isPaused)
{
   paused = isPaused;

   // If the game is paused, timeScale is 0, otherwise 1
   Time.timeScale = (paused) ? 0 : 1;
   if(paused)
   {
       SlideMenuIn(pauseMenu);
   }
   else
   {
      SlideMenuOut(pauseMenu);
   }
}
```

   Note that because `PauseMenuBehaviour` inherits from `MainMenuBehaviour`, it also can call the `SlideMenuIn` and `SlideMenuOut` functions, respectively, as long as they are marked as `protected` or `public`.

2. Now if we run the game, nothing will appear to happen when we hit the pause menu. This is because—as I mentioned previously—Tweens are scaled by `Time.timeScale`, which we just changed. To fix this, we can make use of another iTween property called `ignoretimescale`, which we will set to `true` in both functions we wrote previously in the `MainMenuBehaviour`:

```
/// <summary>
/// Will move an object from the left side of the screen
/// to the center
/// </summary>
/// <param name="obj">The UI element we would like to
/// move</param>
```

```
public void SlideMenuIn(GameObject obj)
{
   obj.SetActive(true);
   var rt = obj.GetComponent<RectTransform>();

   // Set the object's position offscreen
   var pos = rt.position; pos.x = -rt.rect.width; rt.position = pos;

   // Move the object to the center of the screen
   iTween.MoveTo(obj, iTween.Hash("x", Screen.width / 2, "easeType",
                                  "easeInOutExpo", "time", 1.5f,
                                  "ignoretimescale", true ));
}

/// <summary>
/// Will move an object from the center of the screen
/// to the right
/// </summary>
/// <param name="obj">The UI element we would like to
/// move</param>
public void SlideMenuOut(GameObject obj)
{
   var rt = obj.GetComponent<RectTransform>();

   // Set the object's position offscreen
   var pos = rt.position;
   pos.x = Screen.width / 2;
   rt.position = pos;

   // Move the object to the center of the screen
   iTween.MoveTo(obj, iTween.Hash("x", Screen.width + (rt.rect.width),
                                  "easeType", "easeOutQuad",
                                  "time", 1.5,
                                  "oncomplete", "OnHidden",
                                  "oncompletetarget", gameObject,
                                  "oncompleteparams", obj,
                                  "ignoretimescale", true ));
}
```

3. Save both scripts and dive into the editor and try it out:

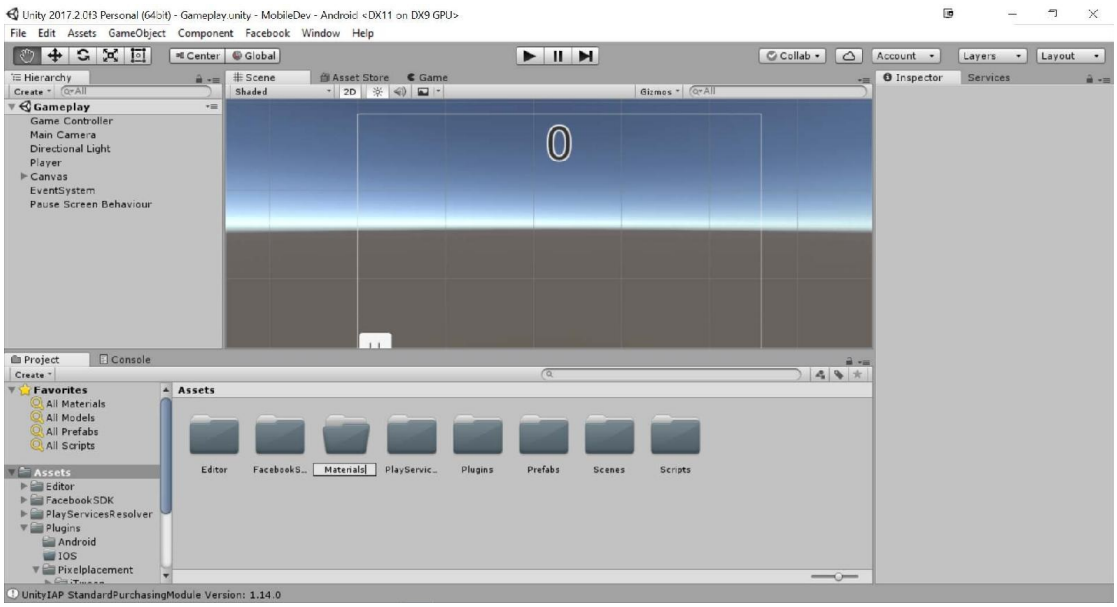
Perfect! We now how the screen flying in just like we wanted it to.

Working with materials

Earlier, we always used the default material for everything in our project. This has worked out well for us, but it may be a good idea for us to talk a little bit about creating custom ones to improve the visuals of our player. Materials are instructions on how to draw 3D objects within Unity. They consist of a shader and properties that the shader uses. A **shader** is a script that instructs the material on how to draw things on the object.

Shaders are a huge subject that entire books have been written on, so we can't dive too much into them here, but we can talk about working with one that is included in Unity, the Standard Shader.

1. First, open the Gameplay scene. Then, let's create a new folder in the Project window called `Materials`:



2. Open up the `Materials` folder we just created, and then once inside, create a new Material by right-clicking within the folder and then selecting Create | Material. Name this new material to `Ball`.
3. In the Inspector, you'll be brought to the Shader menu with the properties for the Standard shader. Set the Metallic property to `0.8` and the Smoothness

property to `0.6`.

4. Now, drag and drop the Ball material onto our player object:

The metallic parameter of a material determines how *metal-like* the surface is. The more metallic a surface is, the more it reflects its environment. The smoothness property determines how smooth the property is; a higher smoothness will have light bounce off it uniformly, making the reflections clearer.

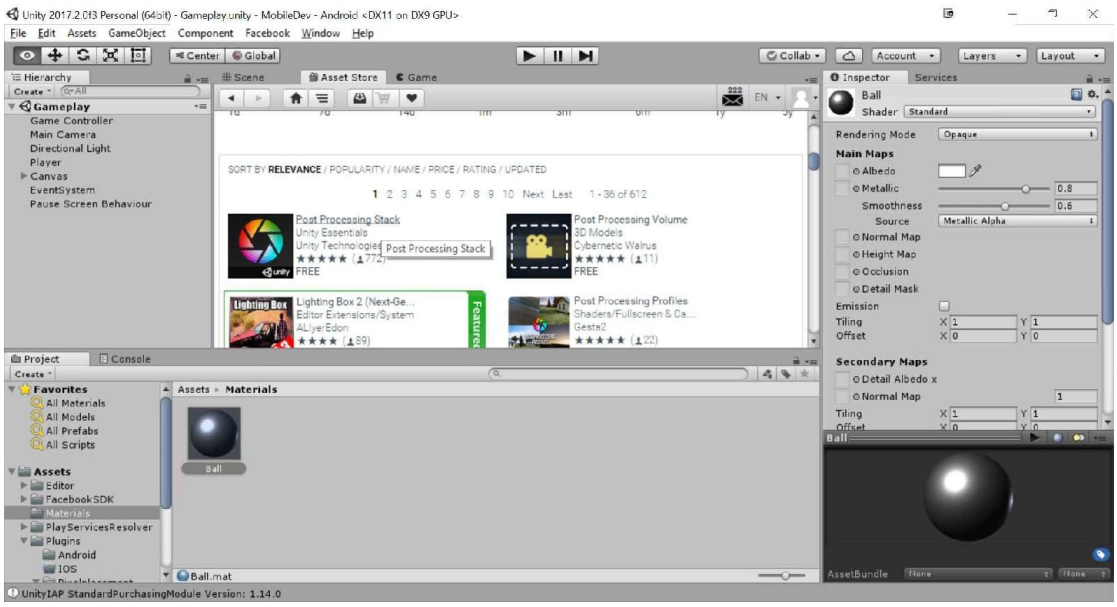
For more information on the standard shader and its parameters, chec
/docs.unity3d.com/Manual/StandardShaderMaterialParameters.html.

Using post-processing effects

One of the ways that we can improve the visual quality of our game with little effort is using post-processing effects (previously called Image Effects). Post-processing is the process of applying filters and other effects to what the camera will draw (the image buffer) before it is displayed on the screen.

Unity includes a number of effects in its freely available post-processing stack, so let's go ahead and add it:

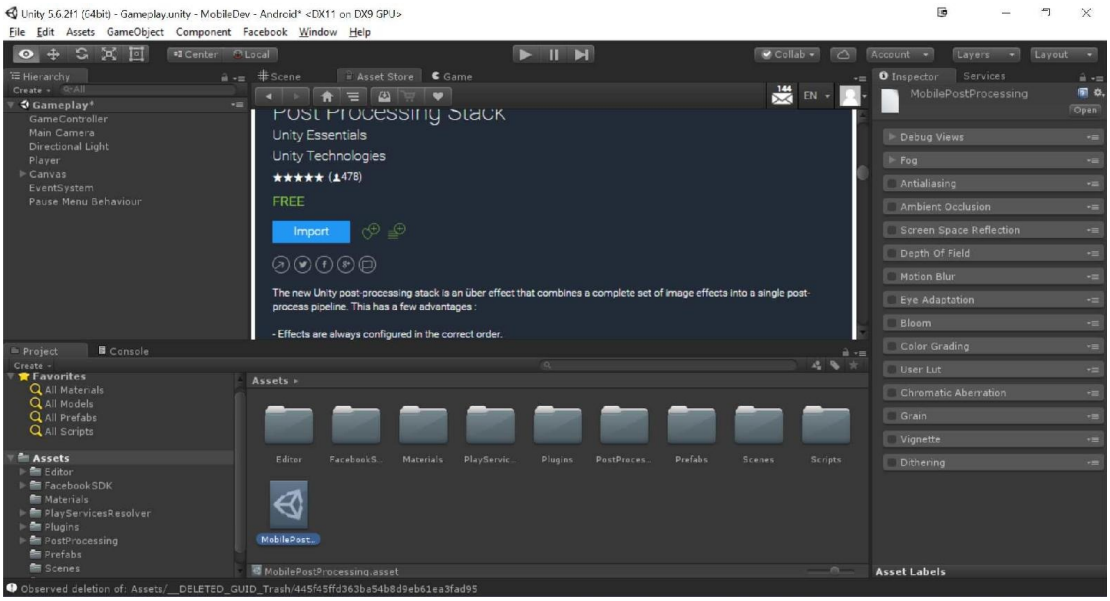
1. Open up the Asset Store again, and search for `Post Processing Stack` this time:



2. Select to Download it, and click on the Accept button when it asks you to. Once it's finished, go ahead and import the contents into our project.
3. Switch to the Scene window, and then from the Hierarchy window, select our Main Camera object and select Add Component in the Inspector and type in `PostProcessing`; then move your mouse over the Post Processing Behaviour selection and then click to add the script to your project.

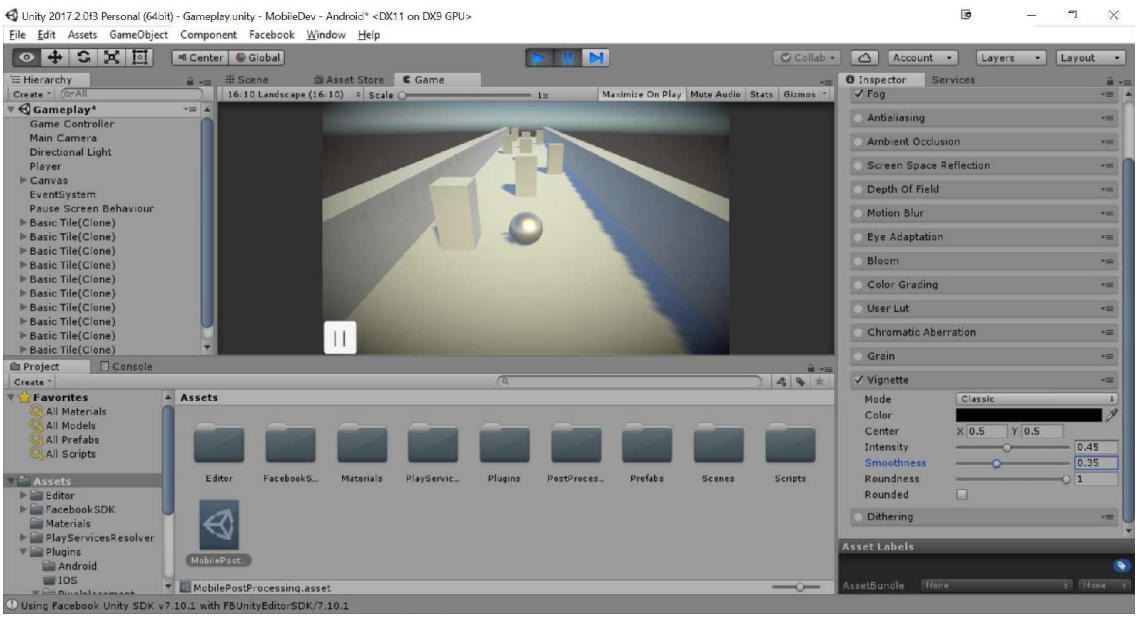
> Note that this component requires a profile. We can go ahead and add that next.

4. We can create a new post-processing Profile by right-clicking on the Project window, selecting Create | Post Processing Profile, and naming it `MobilePostProcessing`:



5. Attach this object to the Profile property of the Post Processing Behaviour component on the camera and then start the game and pause it.
Now, there's a large number of possible effects that can be added to modify how the game looks. Note that for each one you add, the frame rate of the devices we are trying to run our game on will be decreased. Keep testing your device with these options and note how it works.

6. Go to the Project view and then select the `MobilePostProcessing` profile. To start off, toggle the Vignette property. Note how now there seems to be a blackened edge around the game. We will increase the Smoothness to `0.35` to make it even darker by clicking on the top right of the section to expand it:

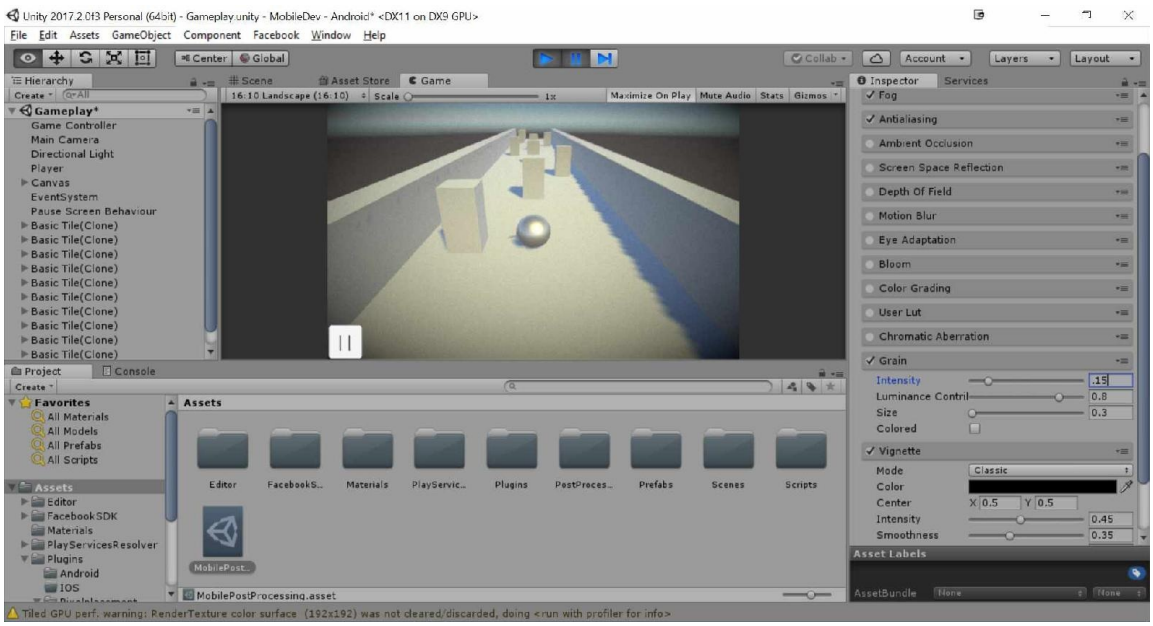


Vignetting is the term used for the darkening and/or desaturating toward the edges of an image compared to the center. I like to use this when I want to have players focus on the center of the screen.

7. Next, check the Antialiasing option.

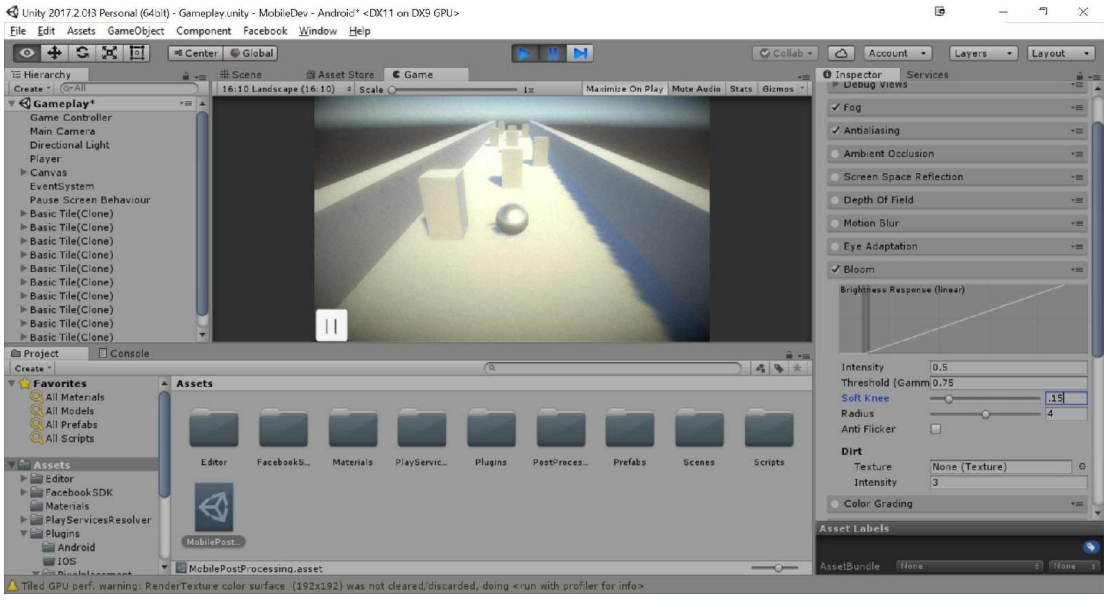
   Aliasing is an effect where lines appear jagged on the screen. This happens if the display we are trying to play our game on doesn't have a high enough resolution to display properly. Anti-aliasing attempts to reduce that appearance by attempting to combine colors nearby these lines to remove the prominence at the cost of it appearing blurrier.

8. Toggle Grain, and you'll note that the screen has become a lot fuzzier. While not a great idea at the default size, decreasing the size to `.1`, unchecking colored, and decreasing the intensity to `.15` will help with the appearance of things:



If you've been to a movie theater that still uses film, you may have noticed how there were little specks of things in the filmstock while playing over time. The grain effect simulates this film-grain, causing the effect to become more pronounced the more the movie is played. This is often used in horror games to obscure the player's vision.

9. Another property to check is Bloom, which makes bright things even brighter. Decreasing the Threshold to `.75` and the Soft Knee to `.1` will help brighten things up:



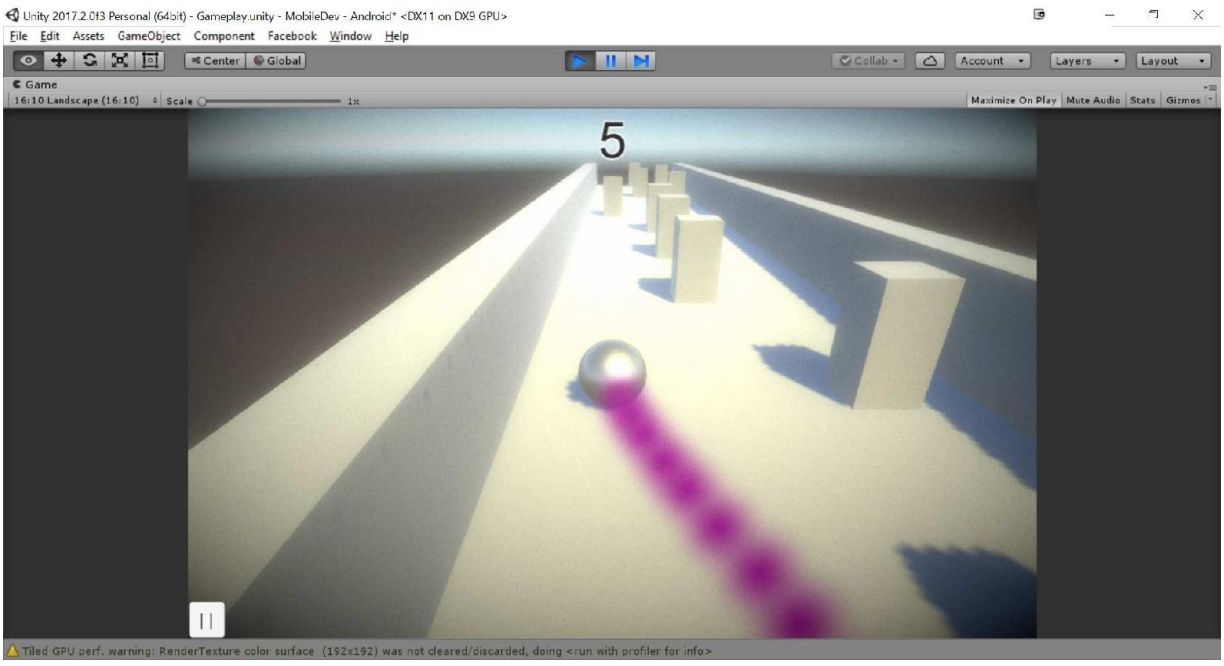
Bloom attempts to mimic the imaging artifacts of real-world cameras, where things in areas with light will glow along the edges, thus overwhelming the camera. There are a number of other properties to look into and adapt your project to look just the way you want it. Check out more information on the post-processing stack at https://docs.unity3d.com/Manual/PostProcessing-Stack.html.

Adding particle effects

The game itself currently works, but it could use some more polish. One of the things we do to increase the polish of game is make use of particle systems; let's get started:

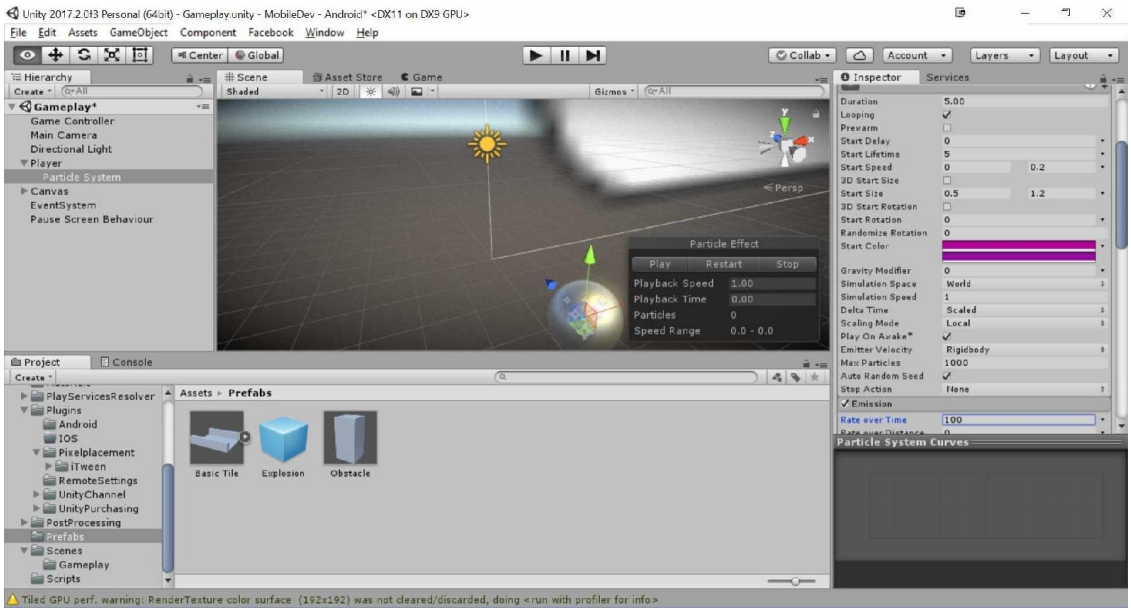
1. First, select the Player in the Hierarchy window and right-click and select Effects | Particle System.
   This will make this system a child of the player, which will be good for what we are going to do.
2. Change the Start Speed to `0` and the Simulation Space to World.

3. Open up the Shape section by clicking on it. Change the Shape to Sphere and set the Radius to `0` (it will automatically change to `0.01`).
4. Then, change the Start Color to something to make it easy to see, such as purple:

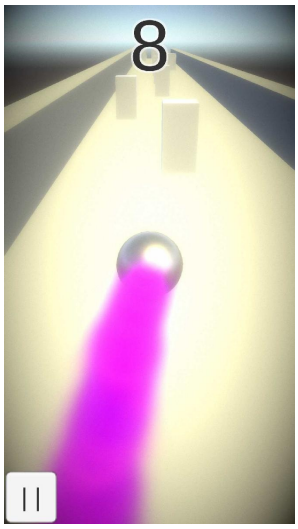


This is a step in the right direction.The particles are now following the player, but there's still a number of things we can do to improve this.

1. Instead of just a single color, we can change it so that we pick randomly between two colors. To do that, go to the right side of the Start Color, and you'll see a little downward-facing arrow. Click on that and then select Random Between Two Colors. Then, change the color to one of two purple colors for some randomness.
2. Then, next to Start Size, click on the right arrow and select Random Between Two Constants and set the values between `0.5` and `1.2`.
3. With that, set the Start Speed property to be a random value from `0` to `0.2`.
4. Then, open up the Emission section and set the Rate Over Time property to `100`:



5. Save the game and play:



As you can see, the particle system looks great on both PC and our mobile device.

If you're interested in exploring more details on things that can be don projects, you can check out my other Unity book, Unity 5.x Game Deve Blueprints, also available from Packt Publishing.

Summary

We now have improved our game by a huge amount by only doing a few simple things to improve the quality of the title. We first animated our menus with a few lines of code using Tweens from iTween and saw how a few lines of code can improve the visual quality of our UI in a number of ways. We then saw how to create materials to improve the visual quality of our ball and then used Post-Processing Effects to polish the contents of our screen. Finally, we discussed how to use particle effects to create a nice trail following our player.

By this point, our game is finally ready for the big leagues. In the next chapter, we will explore getting our game onto the App Store.

Game Build and Submission

Over the course of this book, we have gone over many aspects of building games for mobile devices. The last step in our game development journey is actually releasing the game out into the wild and having people actually play it. All of those long hours of hard work have now come together into something that the masses will be able to enjoy.

When doing this, there are a number of things to keep in mind and this is exactly what we will be discussing next.

Chapter overview

In this chapter, we will go over the process of submitting your game to the Google Play or iOS App Store with tips and tricks to help the process go smoother.

Your objectives

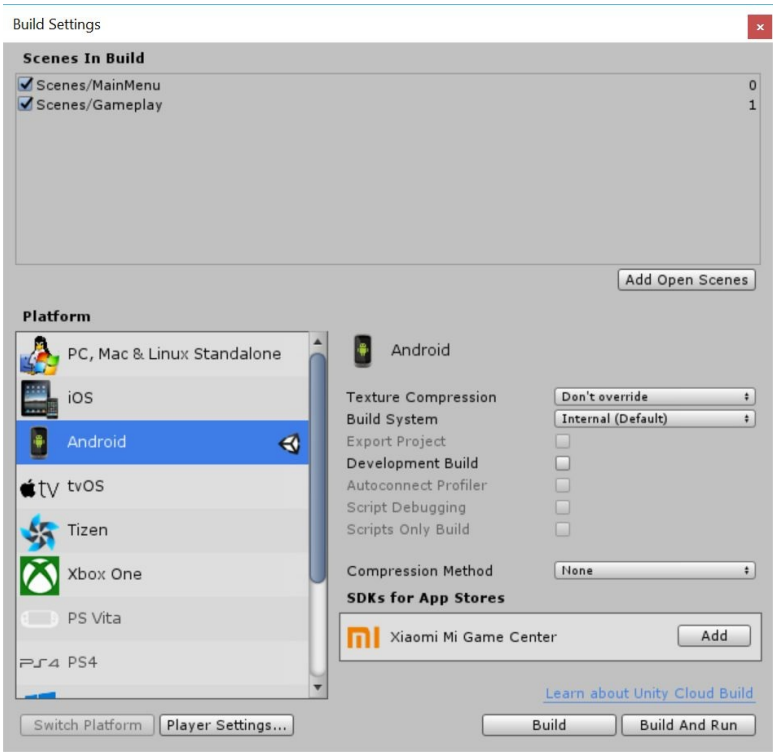
This chapter will be split into a number of topics. It will contain a simple step-by-step process from beginning to end. Here is the outline of our tasks:

- Building a release copy of our game
- Putting your game on the Google Play Store
- Putting your game on the Apple App Store
- What's next?

Building a release copy of our game

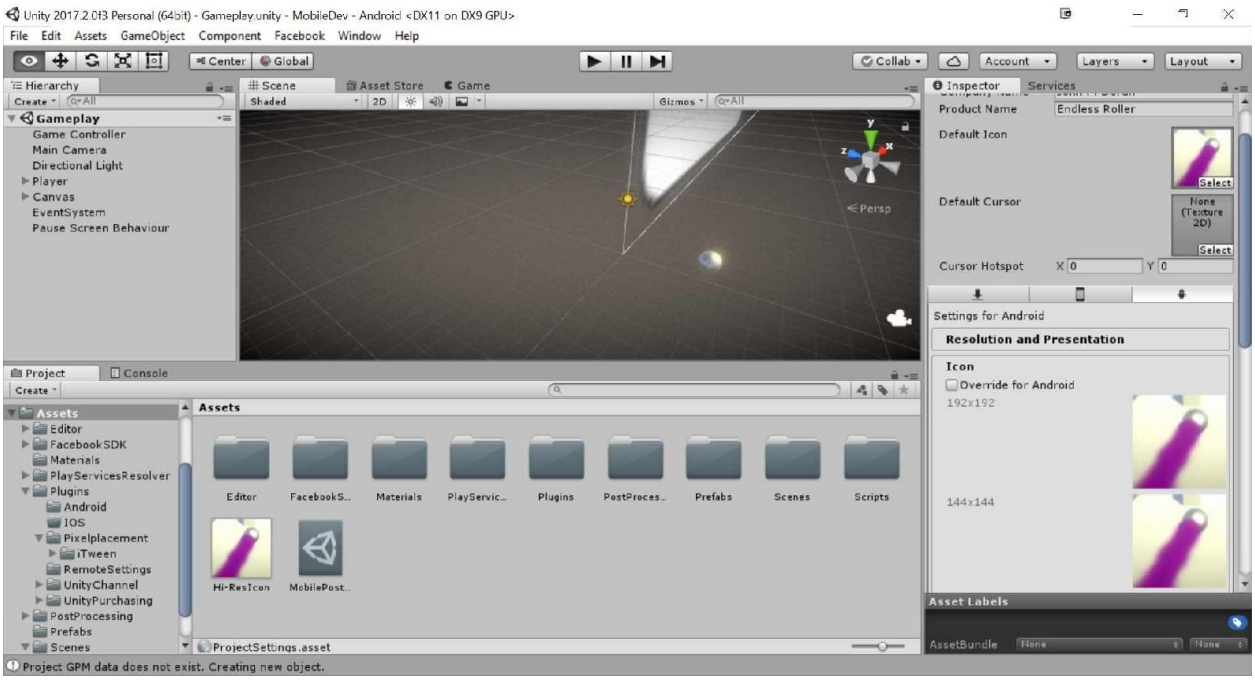
We have exported copies of our game previously, but there are some additional steps that we should do before actually releasing the game on an App Store:

1. The first step will be to confirm you are currently set to deploy your project to our mobile platform of choice. You can check this by going into the Build Settings menu by navigating to File | Build Settings.
2. From there, you should see the Unity logo to the right of the Android or iOS selection. If you do not, select it and then click on the Switch Platform button and wait for it to finish reimporting the assets for the project:



3. After confirming we are building for Android or iOS, open up the Player Settings menu by clicking on the Player Settings button from the menu or by going to Edit | Project Settings | Player.
4. If you haven't done so already, set the Company Name and Product Name values to your own values. In my case, I used `John P. Doran` and `Endless Roller`, respectively.

5. You'll then see a Default Icon item. Drag and drop the `Hi-ResIcon` image into the `Assets` folder and then drag and drop it into the Default Icon slot. This will cause the Icon section of the Android settings to automatically scale the image to fit whatever device you are targeting:



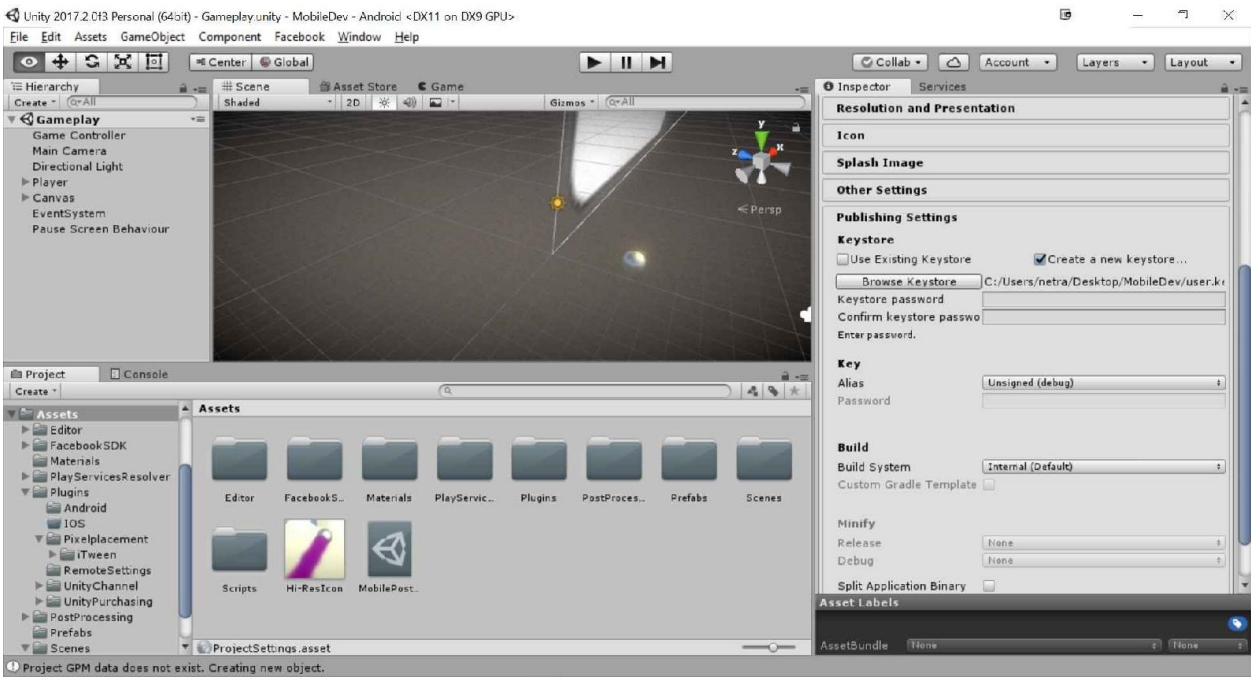
Of course, you can also use your own image, and you can use transparency if you would like to.

6. Under the Resolution and Presentation section, you can enable or disable different rotations and aspect ratios as desired. We adjusted the game to fit these, but this may be useful to know about as you work on your own projects or you wish to restrict users to one experience or another.

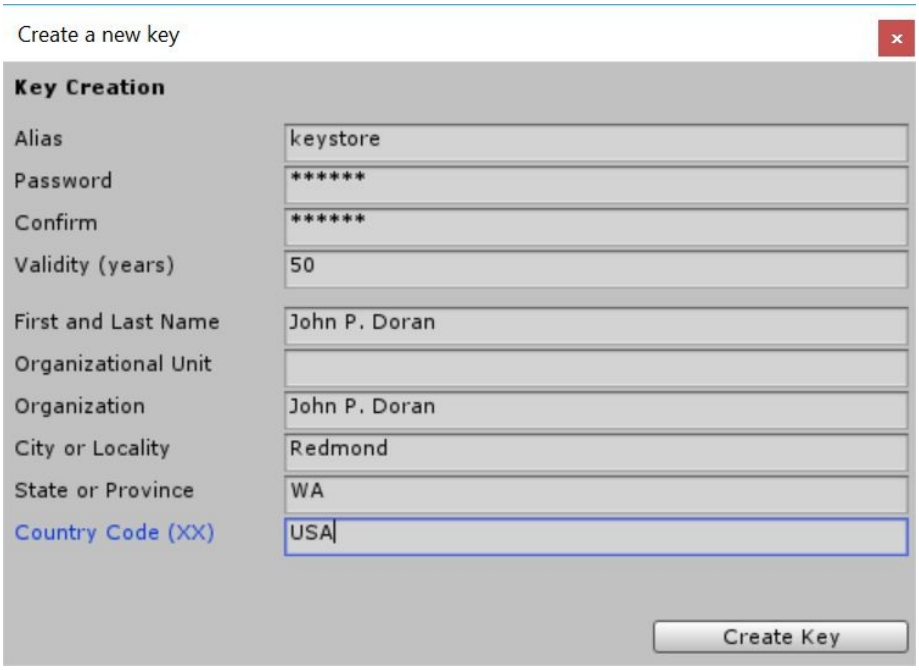
7. The Splash Screen option can be used to display your own logo in addition to Unity's if you have the Personal edition of Unity. If you have pro, you may disable it here.
8. Confirm under Other Settings that the Package Name property is not set to the default values. The general method of naming is `com.CompanyName.GameName`.
9. Next, open up Publishing Settings. This is where we are going to be putting in information about who our game's publisher is (in this case, I'm assuming it's you). Whenever you build a game for Android, you need a Keystore, which allows you to sign off on the game saying that you're allowed to

build. Click on the Create a new keystore icon. Then, select the Browse Keystore button and select a location for this file.

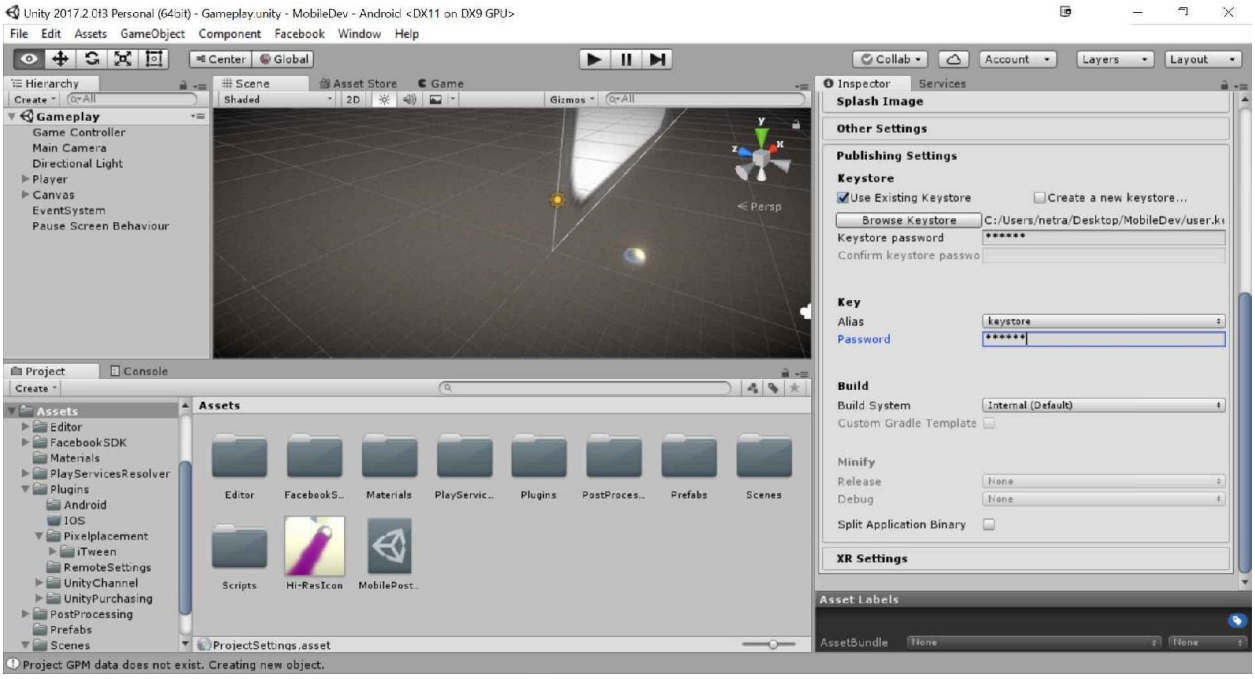
Keep in mind where this is going to be located, as you will be using it in the future to create new versions of your game:



10. Then, you'll need to select a Keystore password textbox that you will need to know as you'll be using it over and over. Afterwards, in the Confirm keystore password text box, you should enter in the same thing as you did before.
11. Next, we will need to click on the dropdown under Key and select Create a new key. From there, you'll need to add in the same information as before: the password with confirmation and then your name and other information. You can see what I put down in the following screenshot. Once finished, click on the Create Key button:



12. Afterwards, from Player Settings, click on the Key dropdown again, and this time, select the keystore we created and then enter the password one more time:



With that, we have everything set up that we need to put the game up on the store.

Putting your game on the Google Play Store

Now that your game is built, you will need to actually put it up on Google's Play Store. To put games up on the Google Play store, you are required to pay a one-time $25 dollar fee. This may or may not seem like a large amount of money, but it is much cheaper than the iOS App Store and is a one-time fee, so for those who are a bit more budget-conscious, you may wish to dive into Google first and make some profit before moving on to Apple's store.

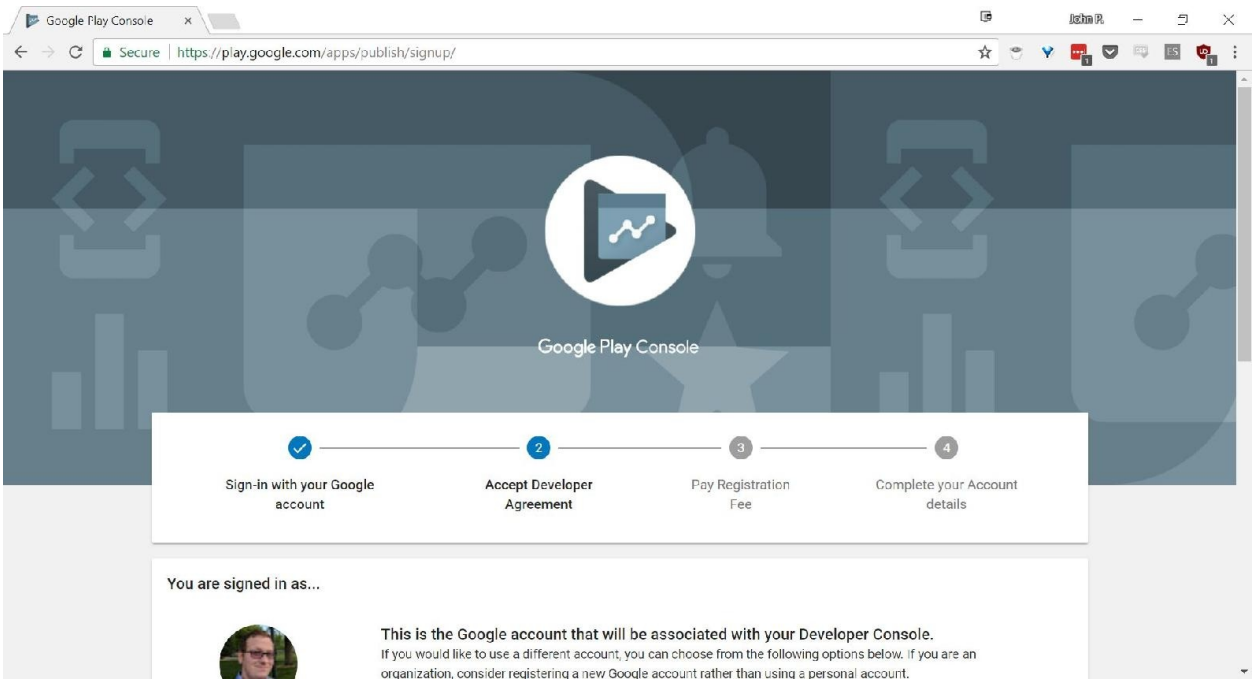
Setting up the Google Play Console

The first step is to gain access to the Google Play console; this is what allows you to publish an Android app on Google Play as well as add Google Play Game Services if you'd like:

1. Open up your web browser and go to https://play.google.com/apps/publish.

   This page is the Google Play Console, which allows you to add apps to the Google Play store.

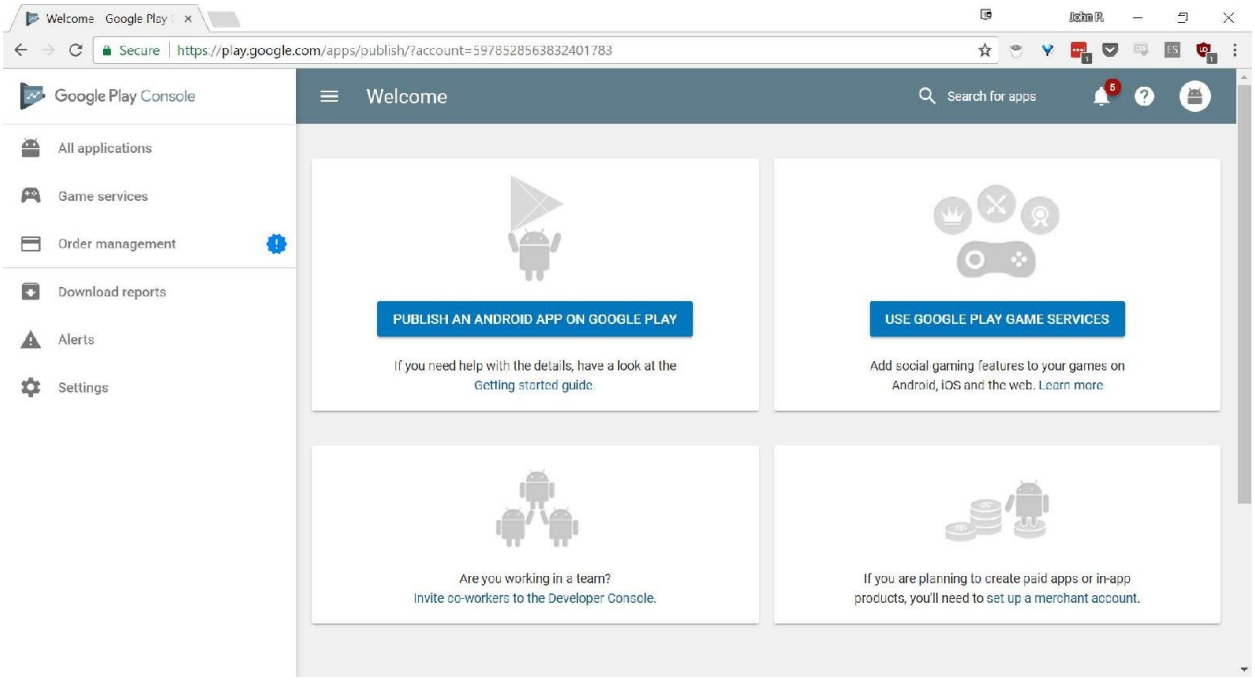
2. If you aren't signed in to your Google account, you'll need to sign in, otherwise you'll be brought to a page that needs you to agree to the developer agreement:



3. Scroll down and you'll see a checkbox saying that I agree and I am willing to associate my account registration with the Google Play Developer distribution agreement. Read the agreement, and if you agree, press the

checkbox.

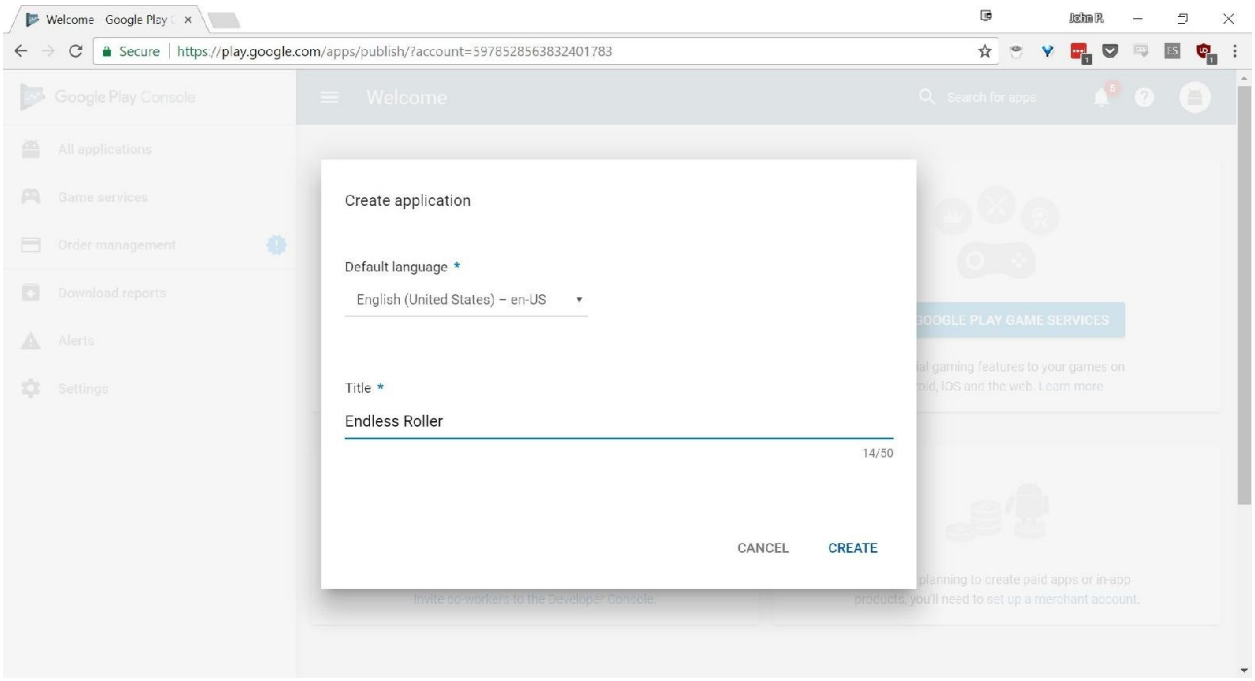
4. Afterwards, click on the Continue to Payment button. You will need to enter in your credit card information and continue until the payment is complete. From there, you'll see a window saying that you'll receive a receipt by mail and then click on the Continue Registration button.
5. You'll then need to enter in the details under the Developer Profile. This will include the developer name, the email you'd like to be contacted by, your website if you have one, and a contact number in case Google needs to contact you about your apps. You'll optionally be offered to receive emails from Google Play, but it's not required for this course.
6. Once you have finished, click on the Complete Registration button. If all goes well, you'll be brought to the Google Play Console:



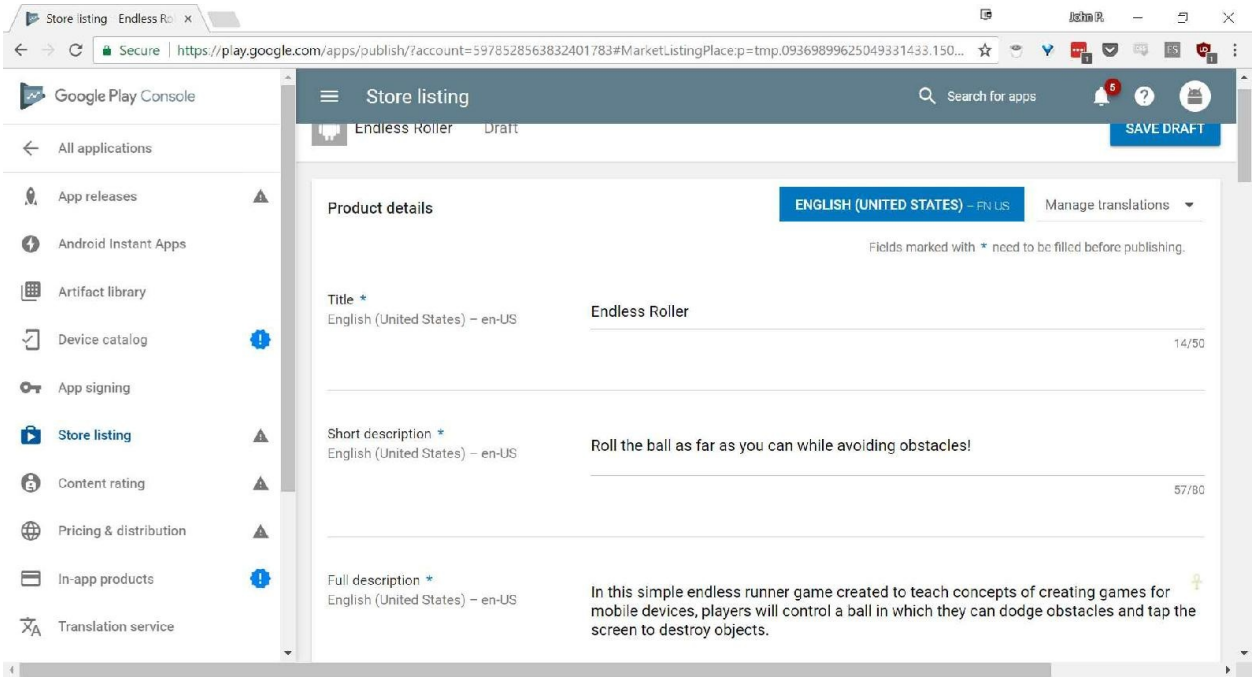
Publishing an app on Google Play

Once you have an account, you can now start the process of actually putting a game up on the Google Play store.

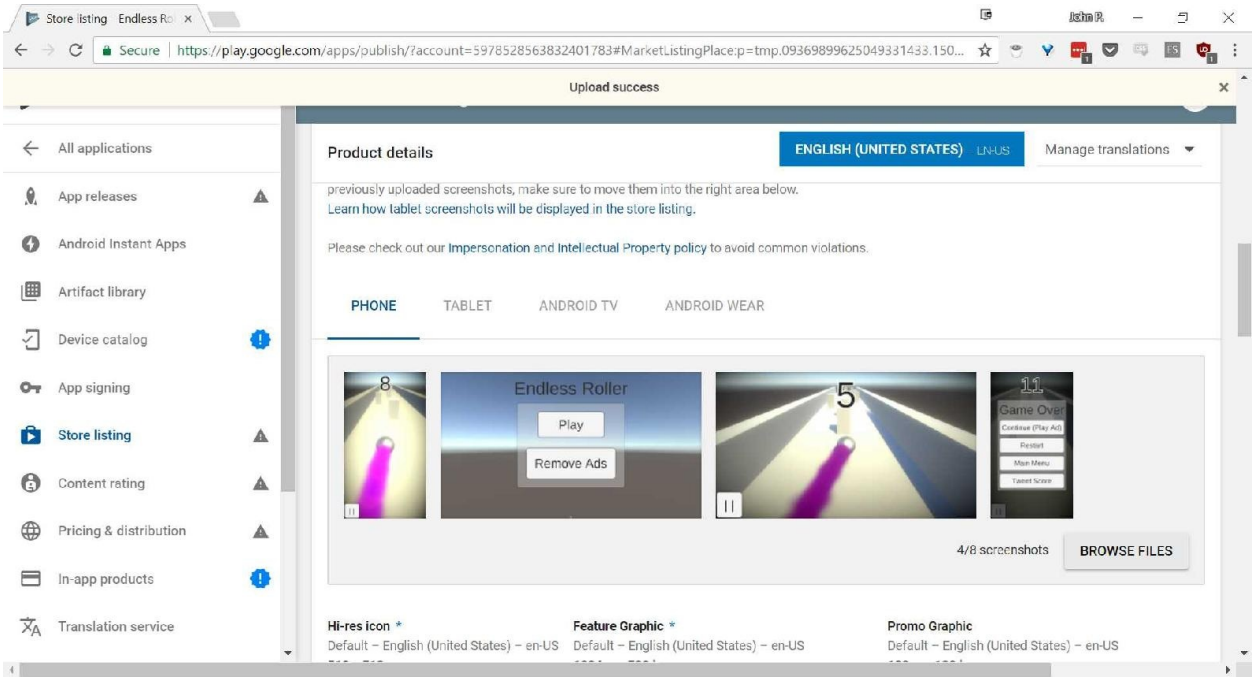
1. Click on the PUBLISH AN ANDROID APP ON GOOGLE PLAY button. You'll be brought to a page where you need to select a Default language and then the Title of your game. Afterwards, click on the Create button:



2. You'll then be brought to a page where you'll need to fill in information about your game, starting with a Short description and then a more detailed Full description:

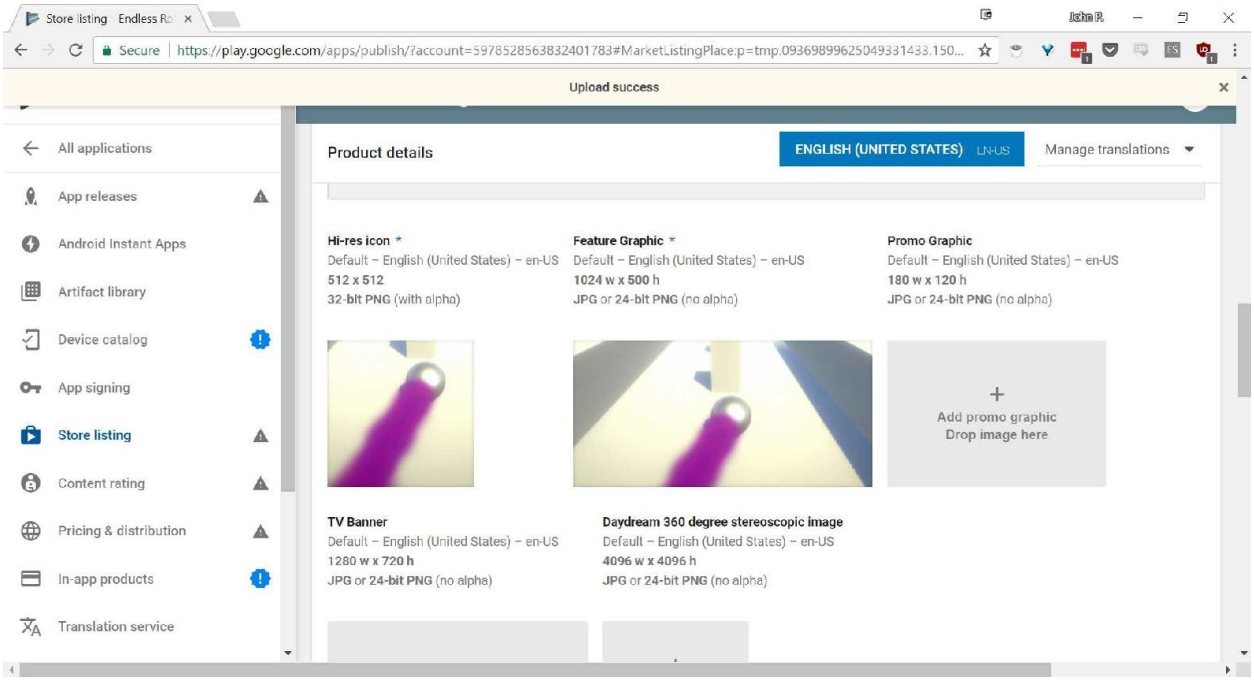


3. You'll then need to provide graphical assets to be used to display your game. You are required to have at least two screenshots and then some additional icons and graphics:

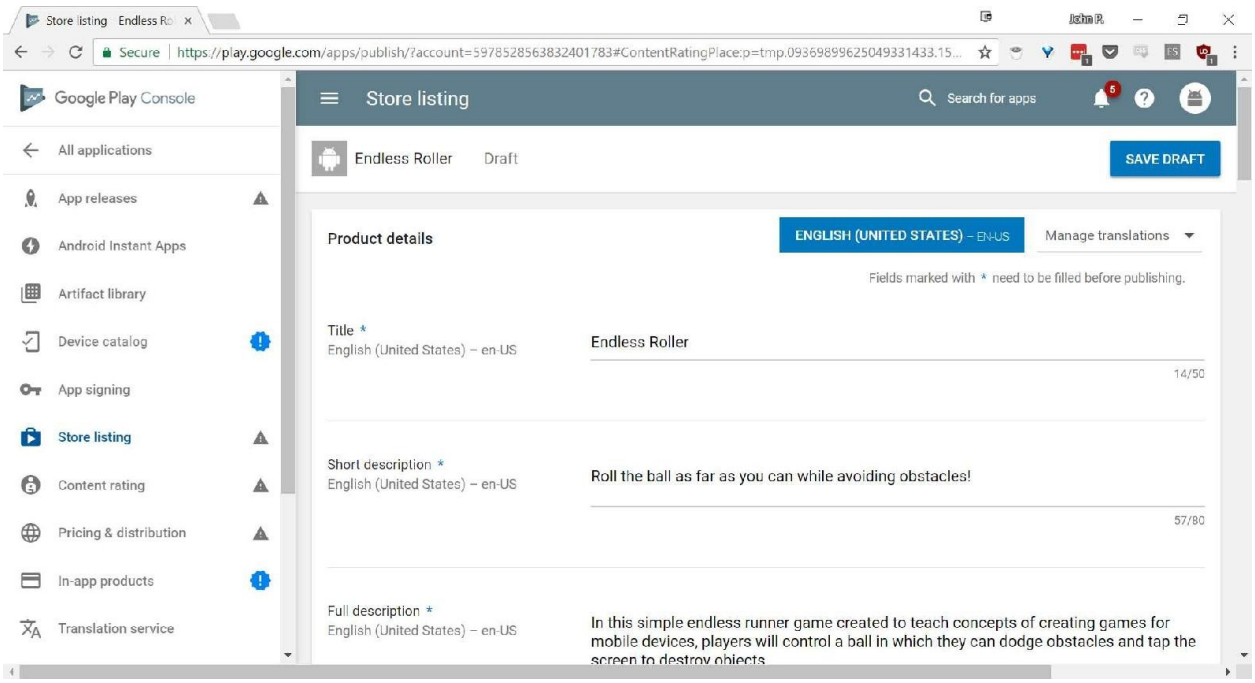


4. You'll then need to include some more images for icons and other featured graphics. The ones with an * are required. You can find some already made

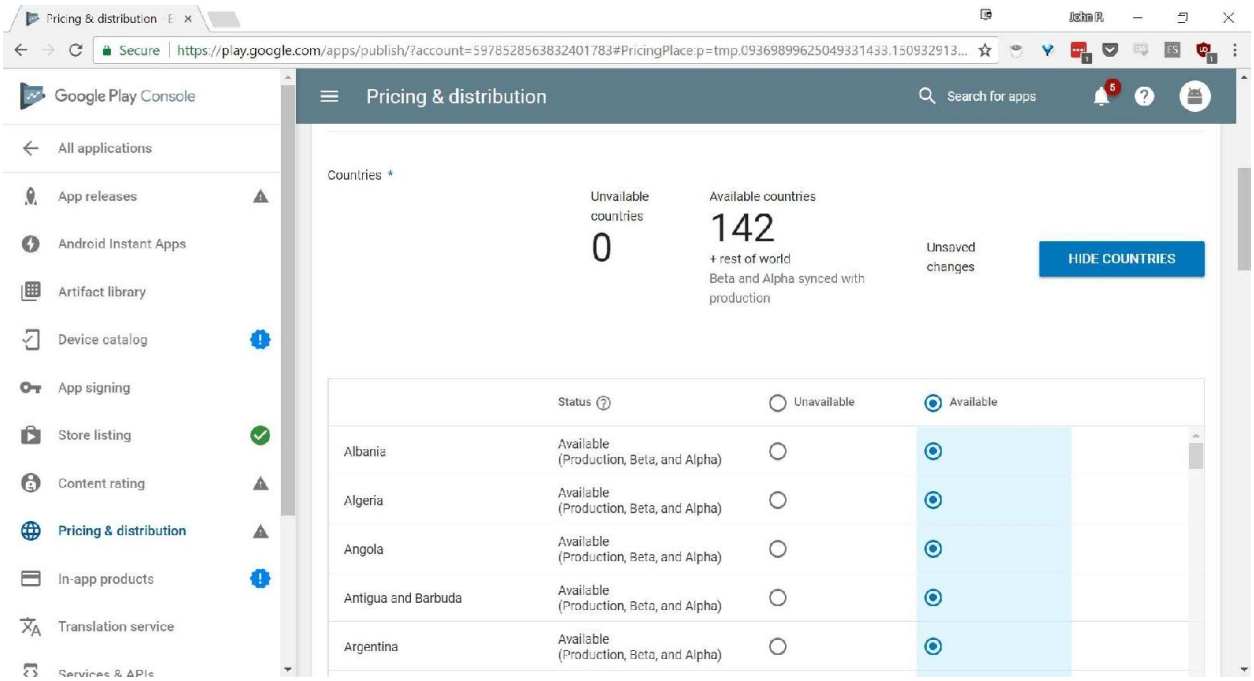
ones in the example code with this book, but I suggest that you create your own once you've customized this game to your liking:



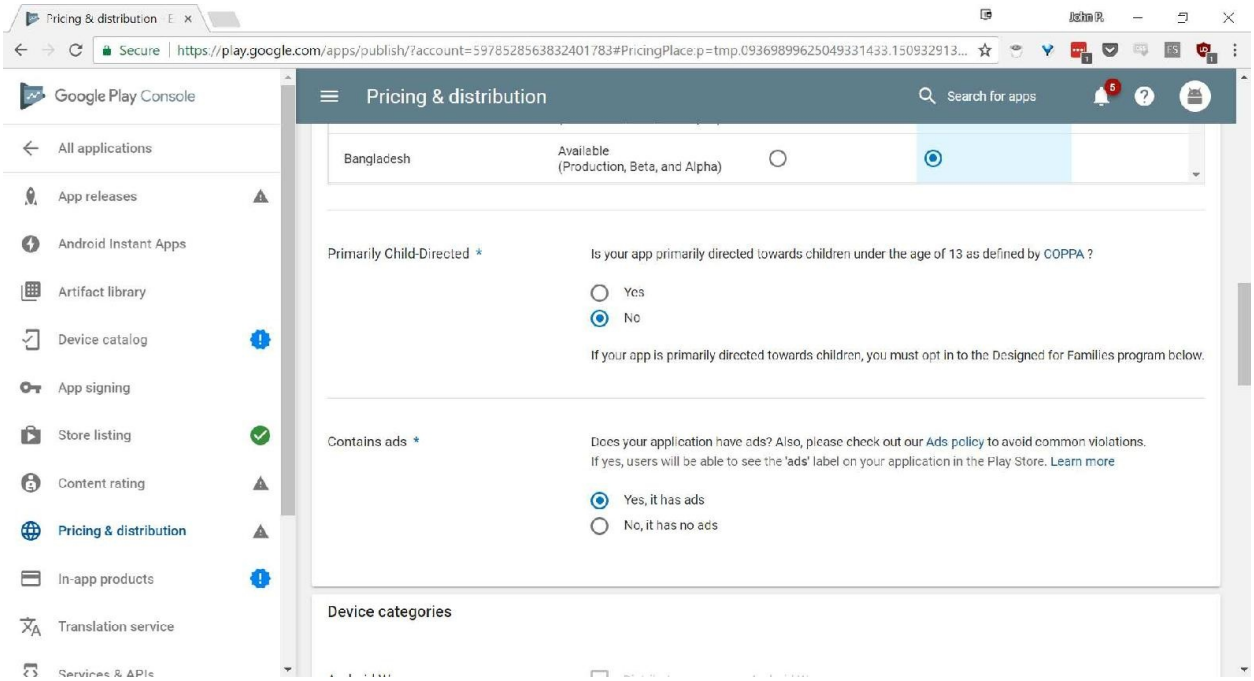
5. Now, scroll down and you'll need to select an Application type (I picked Games) and a Category (I chose Arcade).
6. Finally, confirm your contact info and check whether you have a Privacy Policy or not.
7. Afterwards, scroll all the way to the top and then click on the SAVE DRAFT button:



8. Next, click on the Pricing & distribution option on the left-hand side. By default, you need to decide whether you want your app to be paid or free. I'm going to go with free, but if you click on the set up a merchant account button, you can take payment as well.
9. You'll need to scroll down and select the countries you'd like to have people be able to download your game in. Generally, unless we are doing some kind of testing or beta program, we will generally hit the Available button to allow the entire world to play:

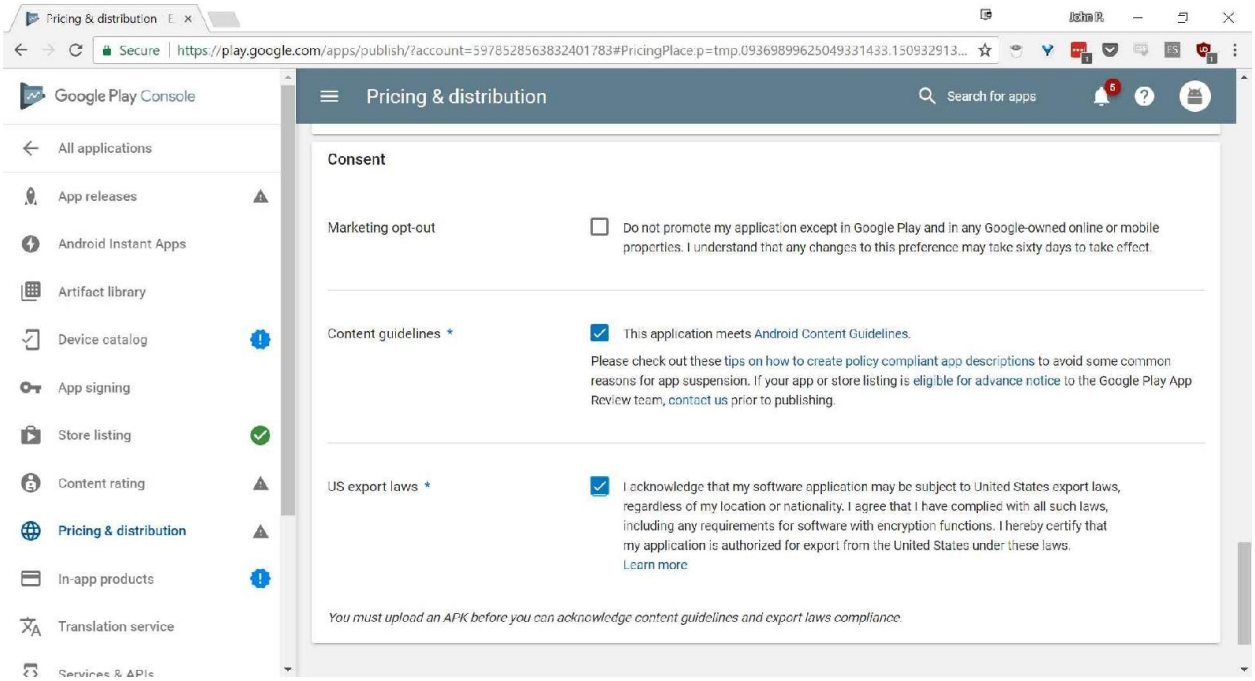


10. Scroll down and you'll need to answer whether your game is directed toward children under 13 and also state whether your game has ads or not. In my case, the app is not directed toward children and the game contains ads:

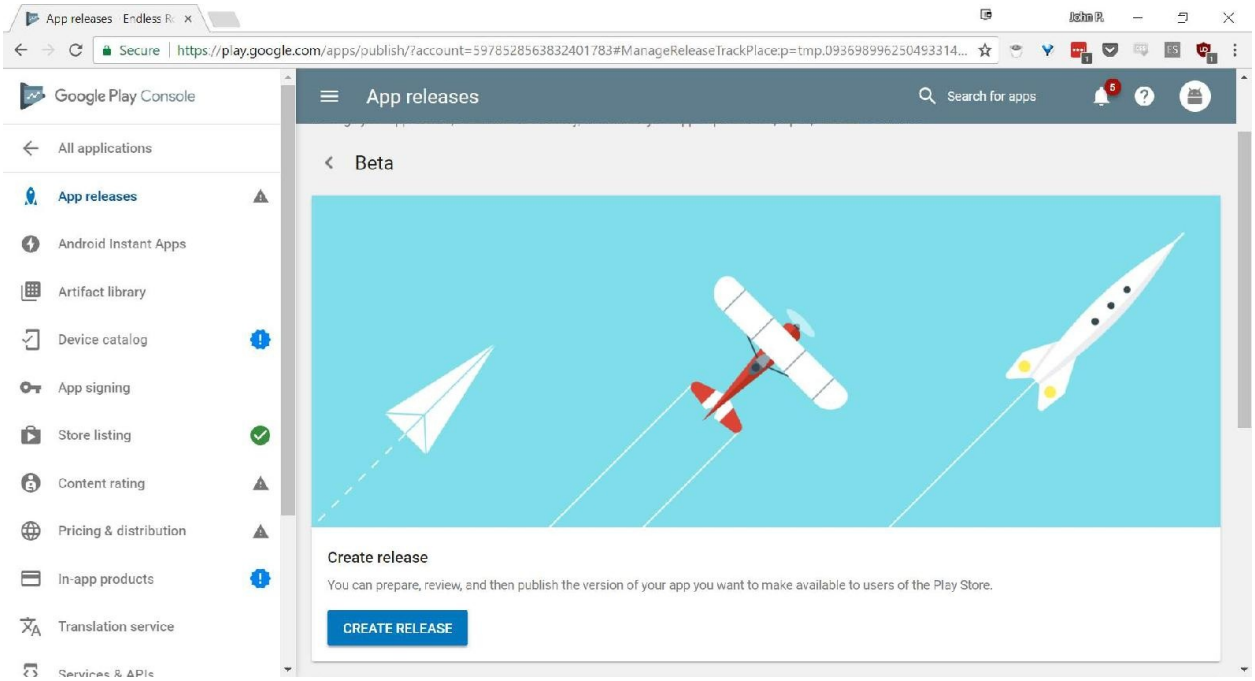


11. Now, scroll down till you get to the Consent section and check the final two

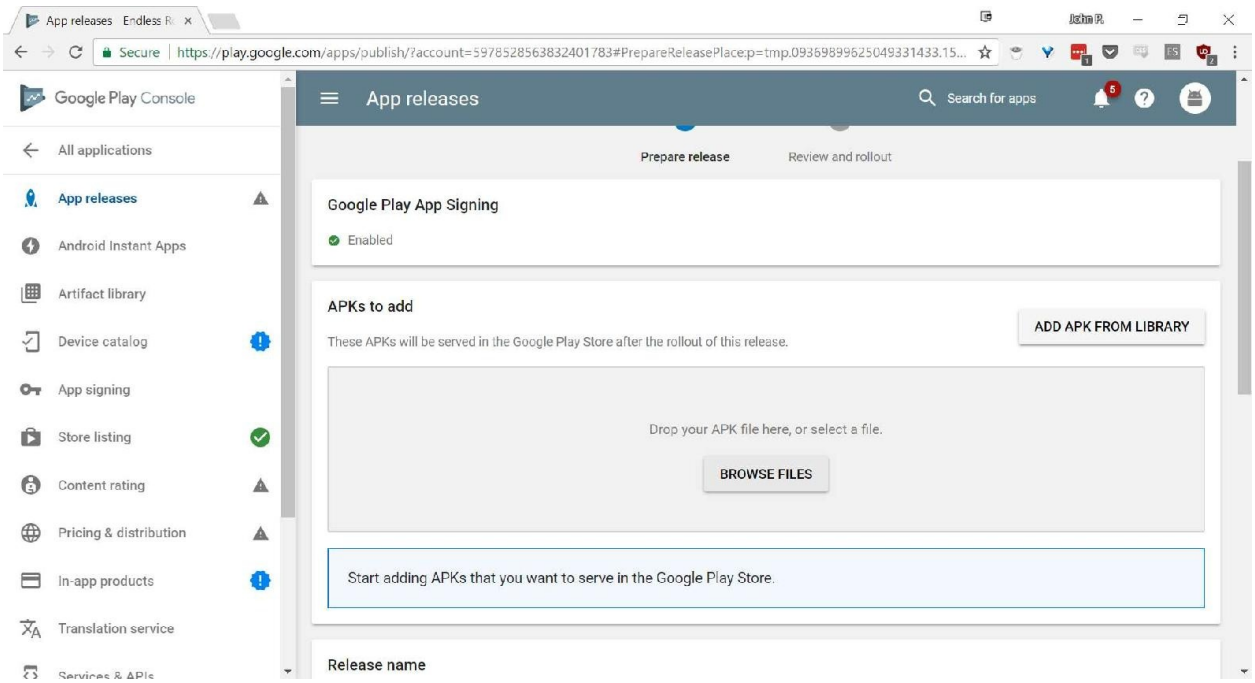
options after reading and agreeing to their stipulations:



12. At the end, scroll all the way up and click on the SAVE DRAFT button again.
13. Next, we will need to bring in our APK file to the store. Click on the App releases section. From there, we need to select what version of the game we want to release. Production means that the game is completely done, but assuming we are looking for feedback and/or wanting to make the project better, we will want to select Beta or Alpha. I'll go ahead and select Beta and click on the Manage Beta option:



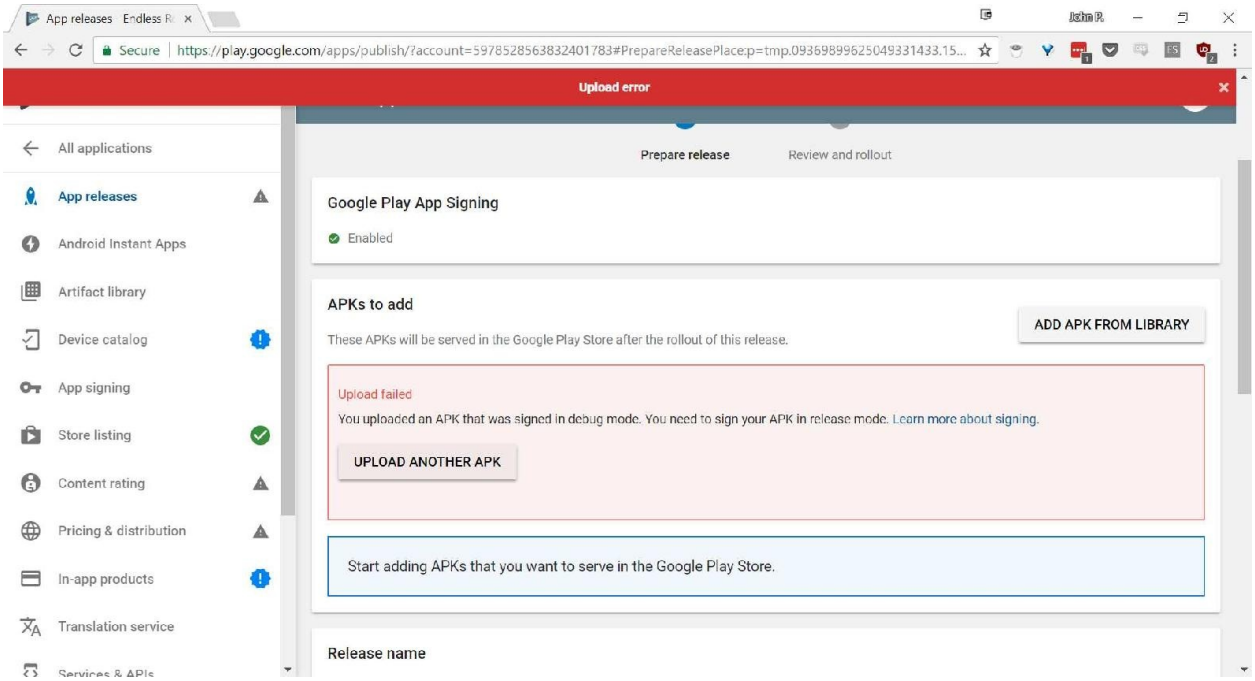
14. From there, click on the Create Release button. You'll then be given the opportunity to enroll the app into Google Play App Signing. Go ahead and click on Continue and accept the terms if you would like. Afterward, you will be brought to a screen to allow you to add an APK:



15. Click on the Browse Files button and go to the folder you've exported your

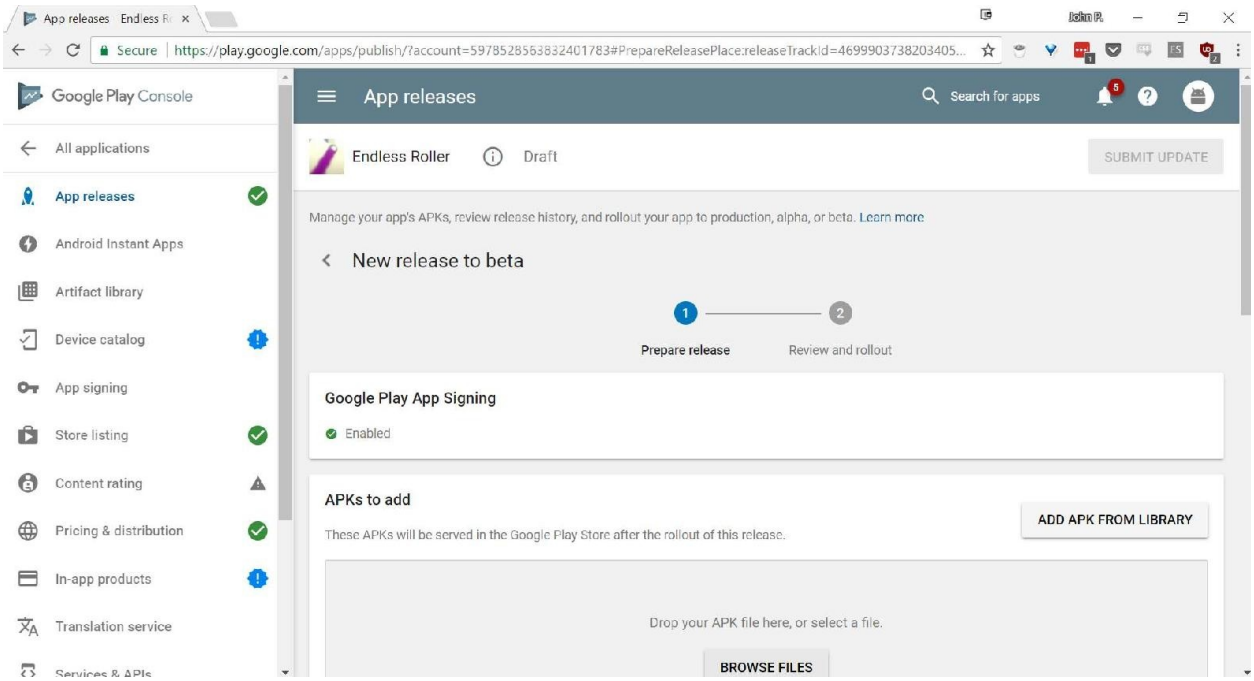
game to.

If you exported your game like we did in Chapter 2, *Setup for Android and iOS Development*, you may notice an error like the following:

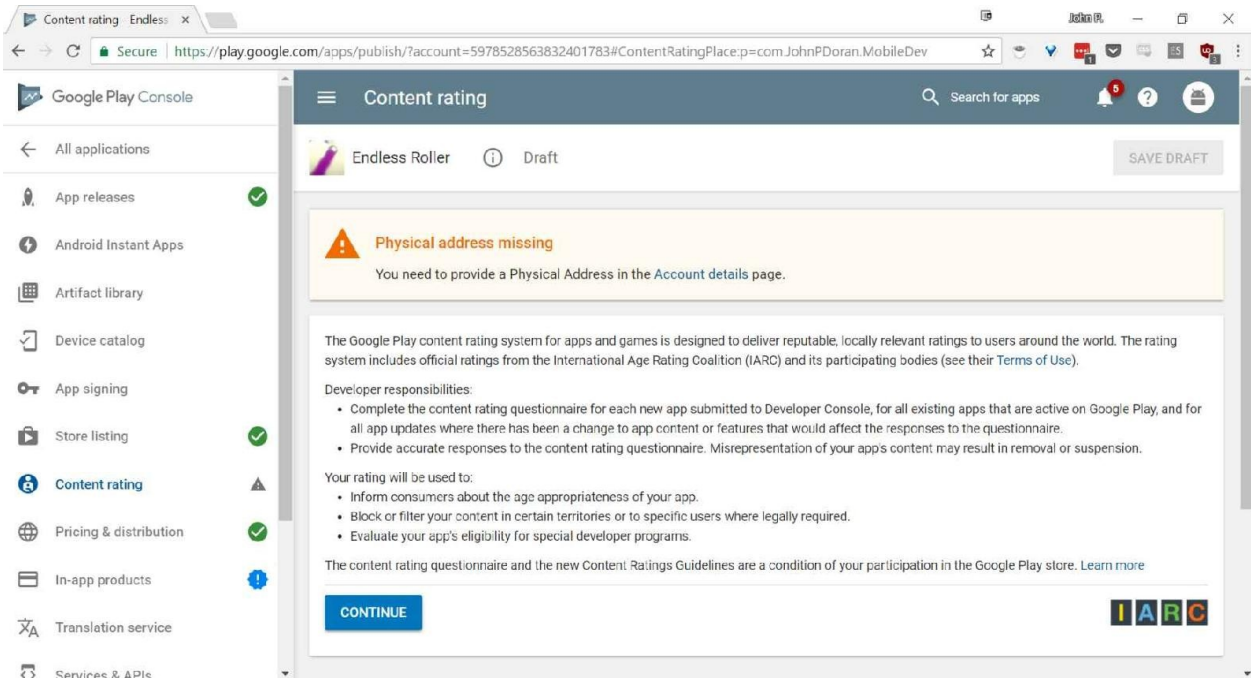


Ensure that you follow the instructions in the *Building a Release Version of our Game* section.

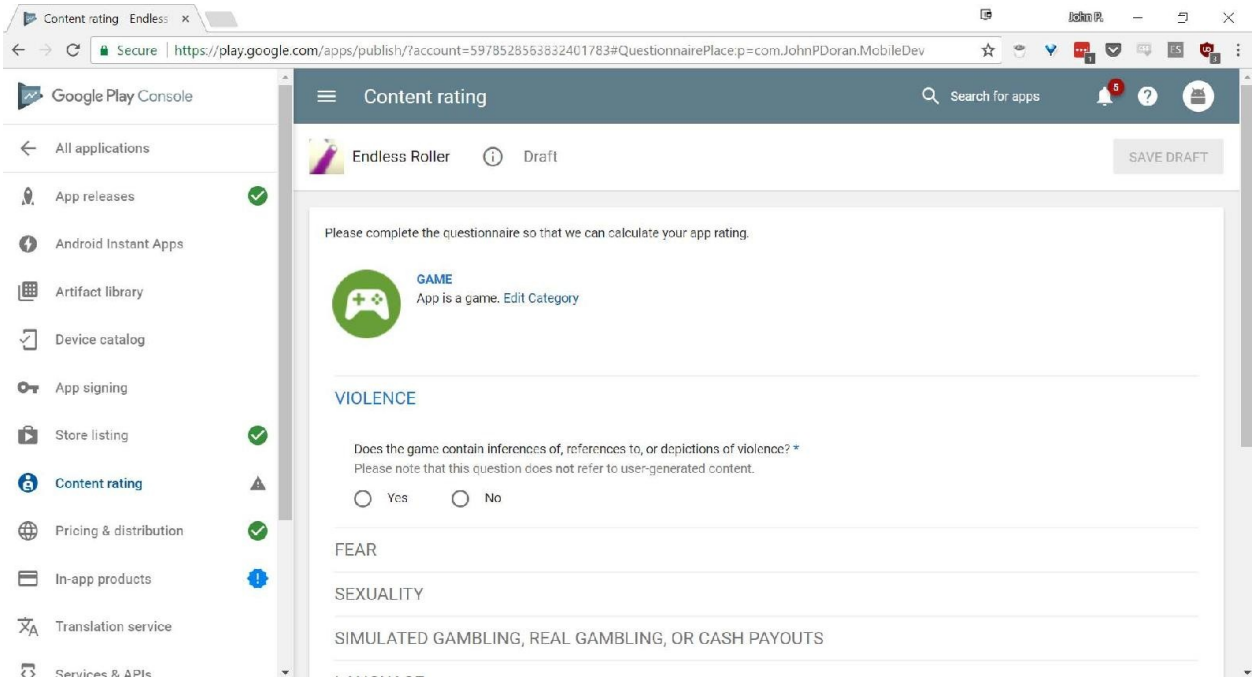
16. If all goes well, you'll be brought to a screen that looks as follows:



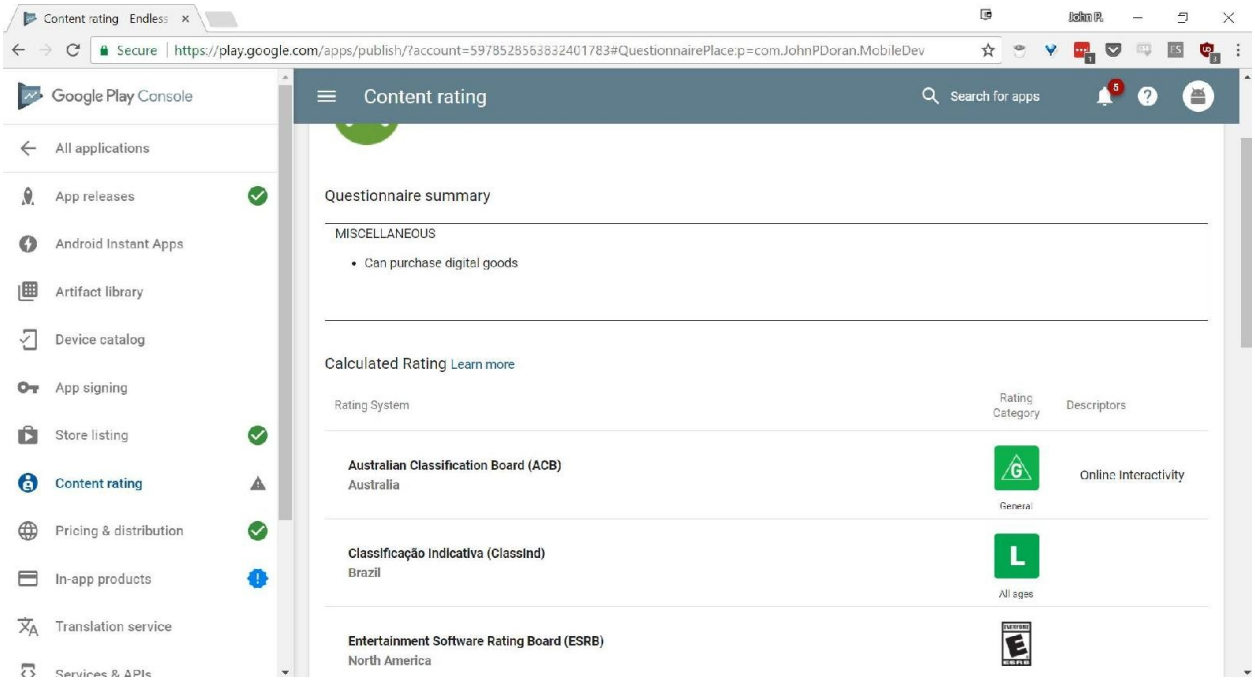
17. If you scroll down, you should see a version of the game on the screen. You'll be asked what is new in this release. I wrote initial release and then clicked on the Save button.
18. Next, click on the Content rating button. You may be required to put in a physical address. If so, click on the Account details page and fill it out, and then click on the Continue button on this page:



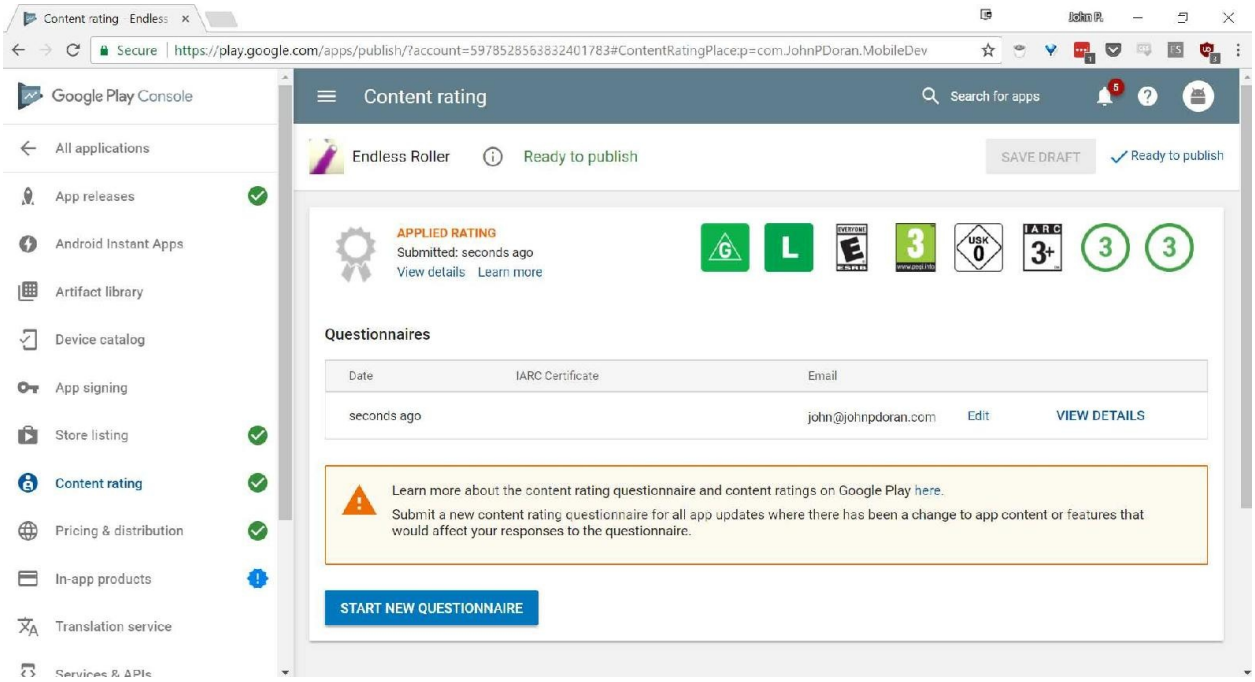
19. From here, you'll need to enter in your email address again and then select your app category. In our case, it's likely Game:



20. Then, answer each of the questions offered until you arrive at the Save Questionnaire button that you will click and then click on the Calculate Rating button.
21. Afterwards, you should see a calculated rating for you to note:



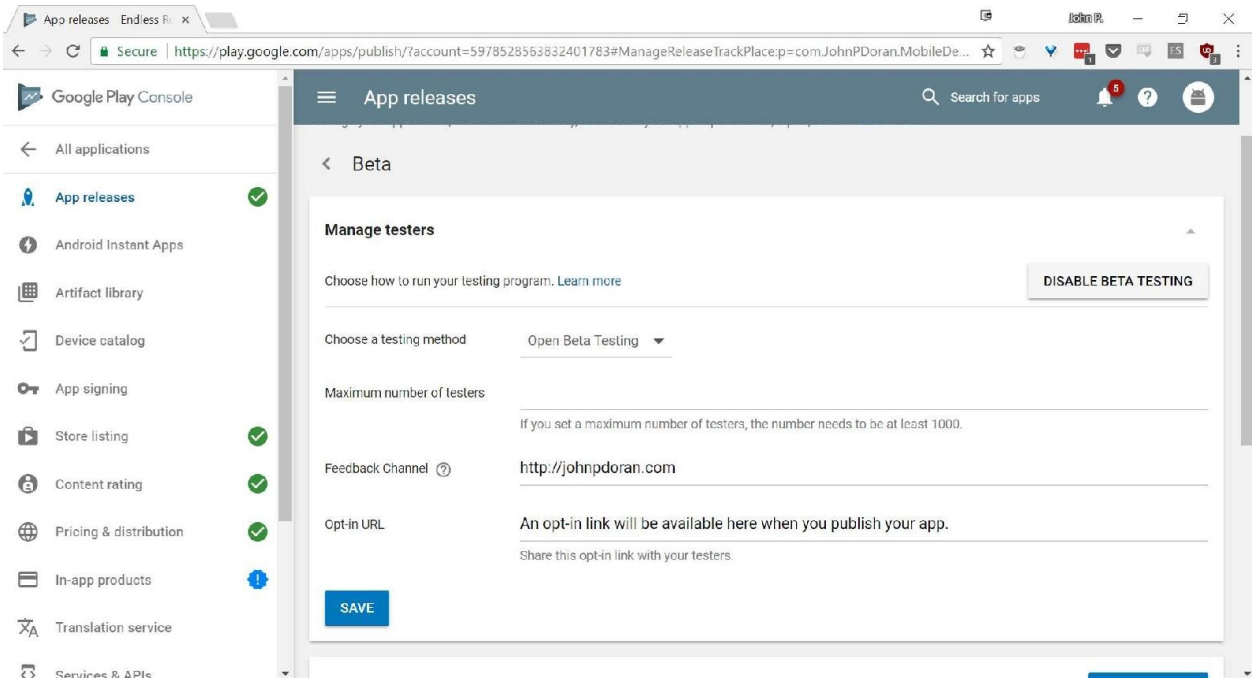
22. Scroll down all the way to the bottom and then click on the Apply Rating button. If all goes well, you should notice that the top of the screen says Ready to publish. Click on that button:



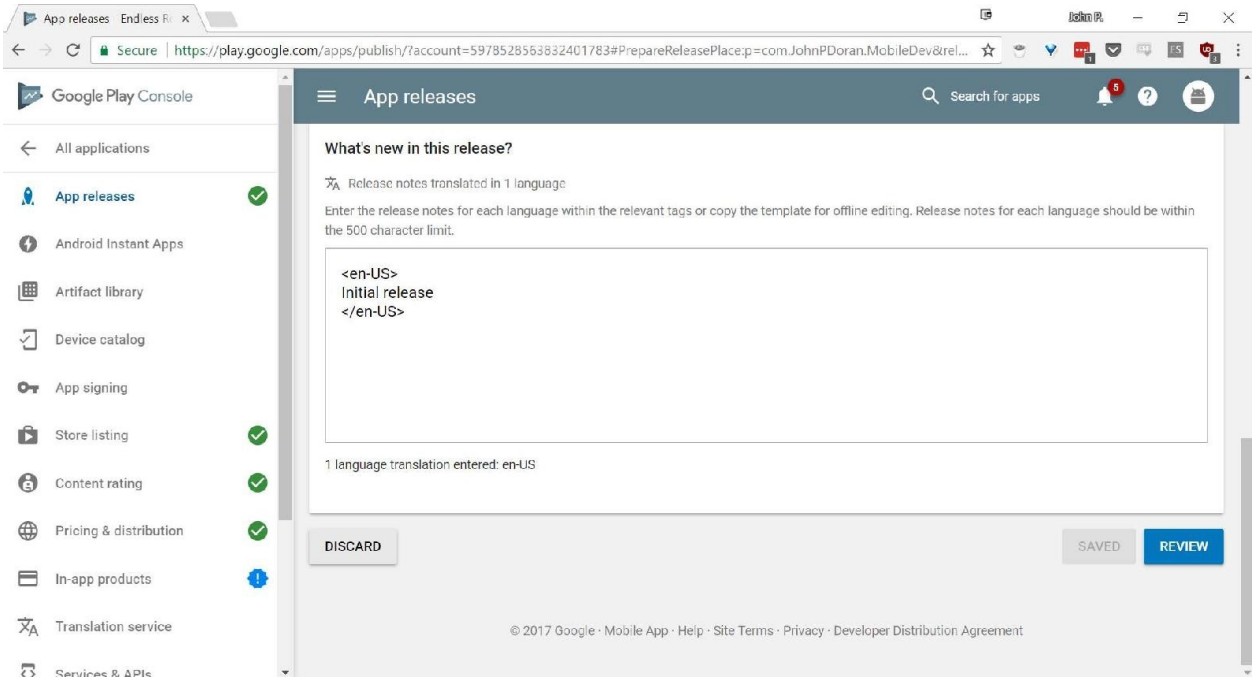
23. Click on the button that says Manage Releases. From there, scroll down to the Beta section again and then click on the Manage Beta button. Here,

you're able to select your method of Open Beta testing.

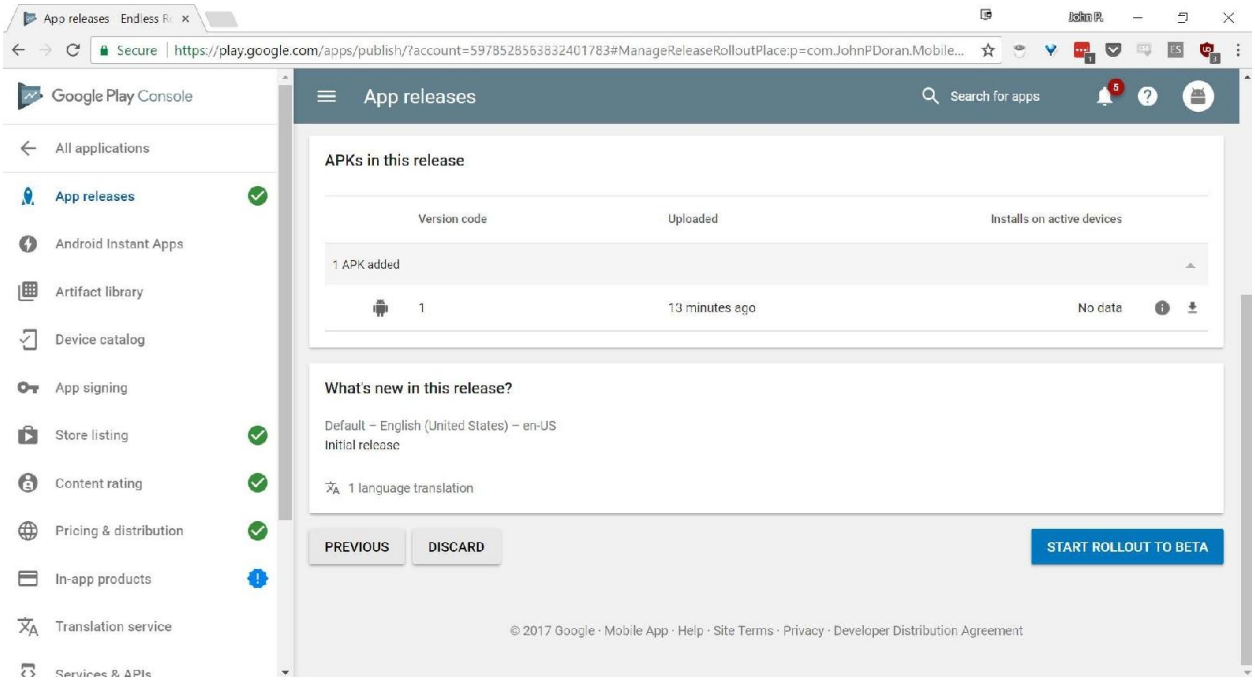
24. Under Choose a testing method, select Open Beta Testing. Afterward, you can select a Feedback Channel to how you want people to provide feedback. Afterwards, click on the SAVE button:



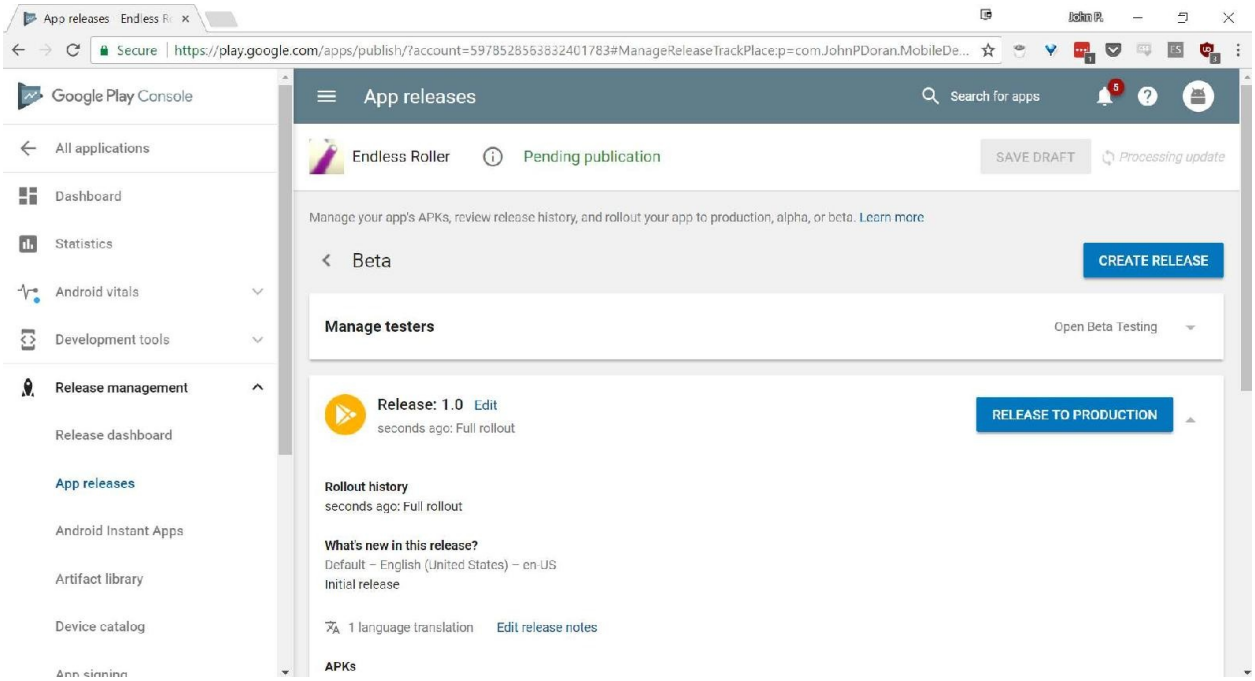
25. Next, return to the App Releases page and click on the Edit Release button under Beta. From there, click on the Review button on the bottom of the screen:



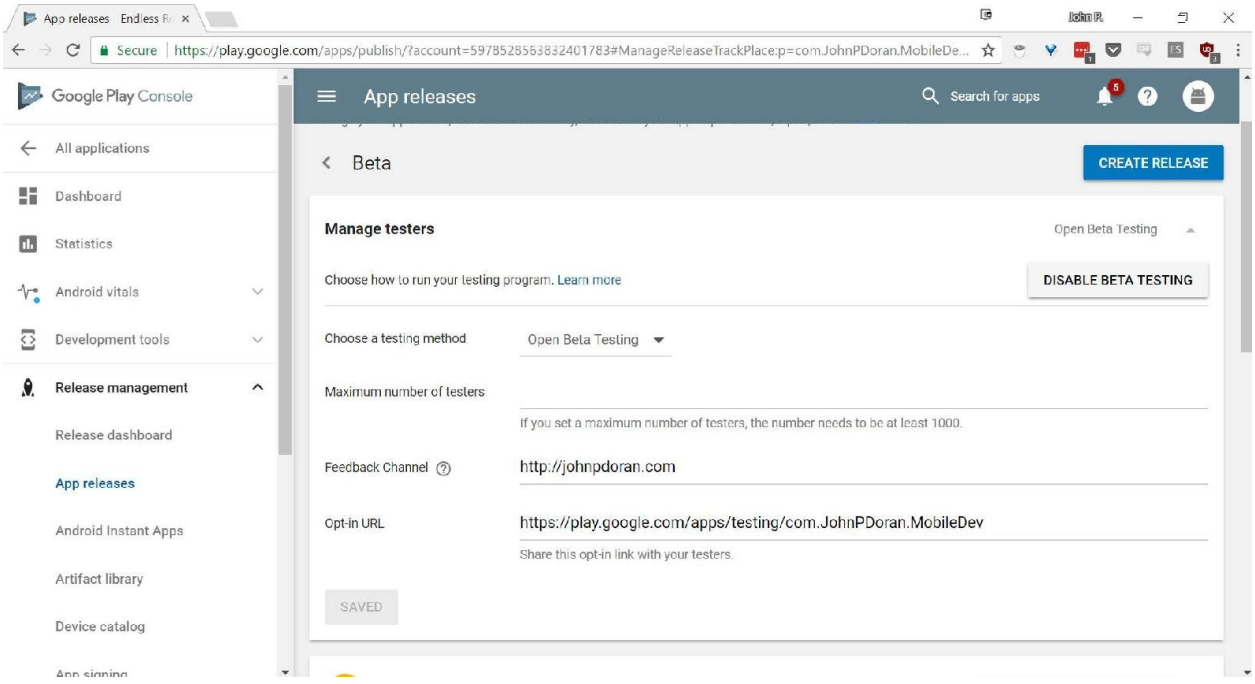
26. Finally, scroll all the way down, and you'll see the START ROLLOUT TO BETA option; click on it:



With that, our game is currently pending publication:



After waiting a moment, your game should be published and you can share it with the world. If you go to the Manage testers section, you should notice an Opt-in URL that you can share and have others play:



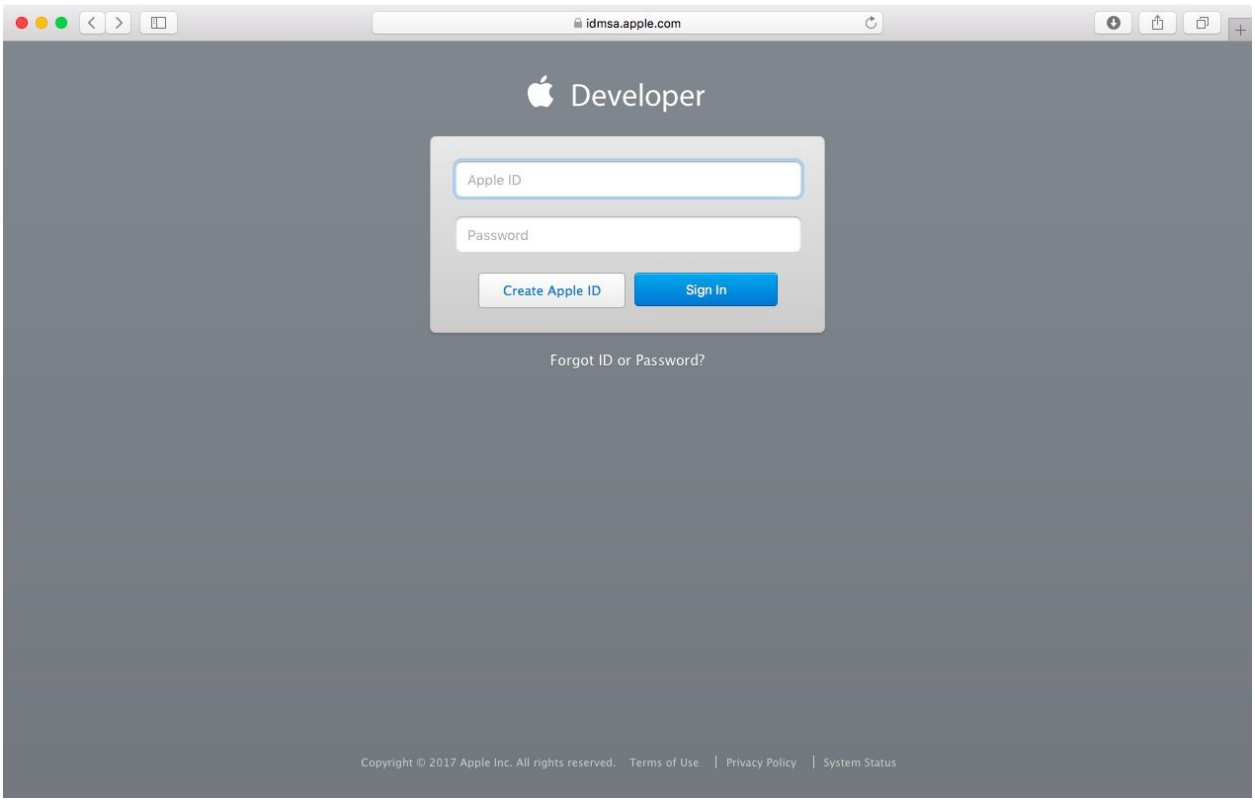
Putting your game on the Apple iOS App Store

Just like the Google Play store, there is an additional fee to put your game on the App Store. Unlike the Google Play store, the fee is $99 plus tax every year. However, a lot of people believe that having their titles on iOS devices is worth the extra cost. In this section, we will go through the process of getting our game up onto the App Store.

Apple Developer setup and the creation of a provisioning profile

In order to deploy to an iOS device, you are required to be on a Mac computer, but before we go onto the iTunes store, we first need to have all of the certificates and permissions figured out ahead of time.

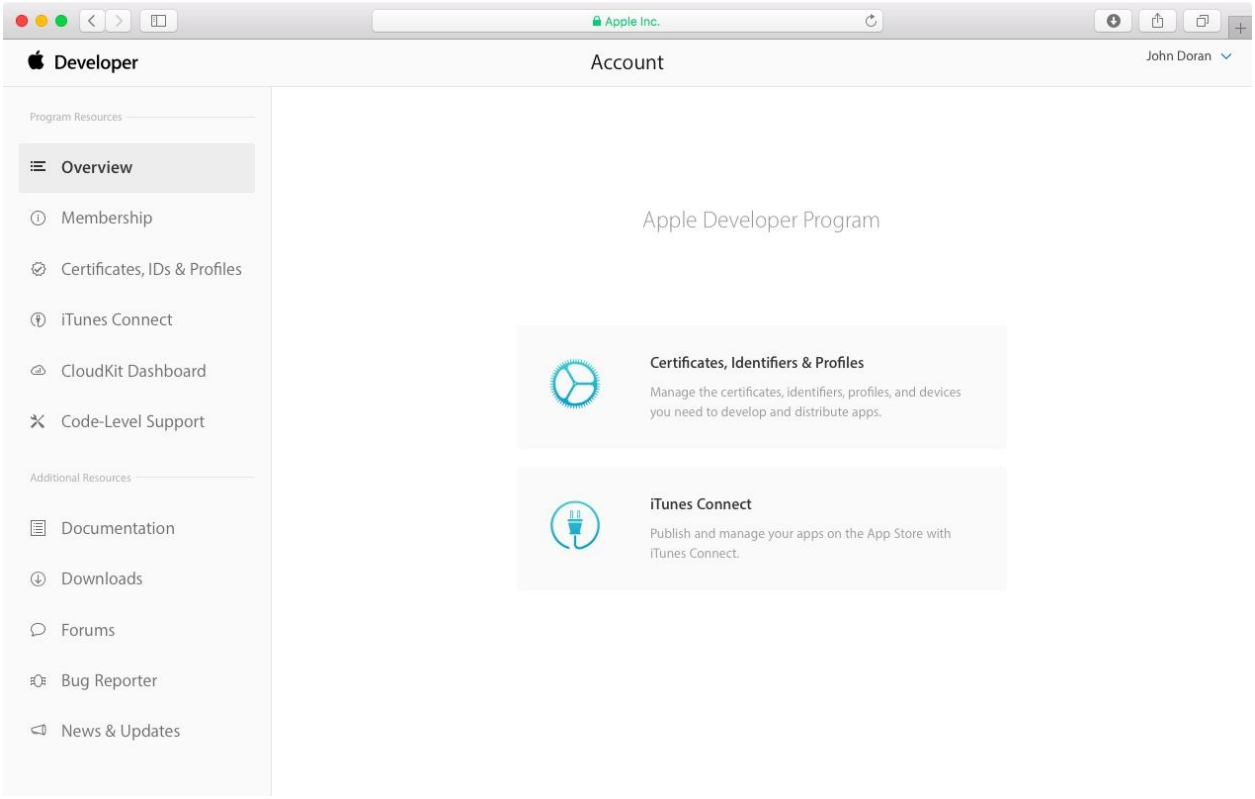
1. With that in mind, on a Mac computer, go to developer.apple.com. From there, fill in your Apple ID and Password and click on the Sign in button:



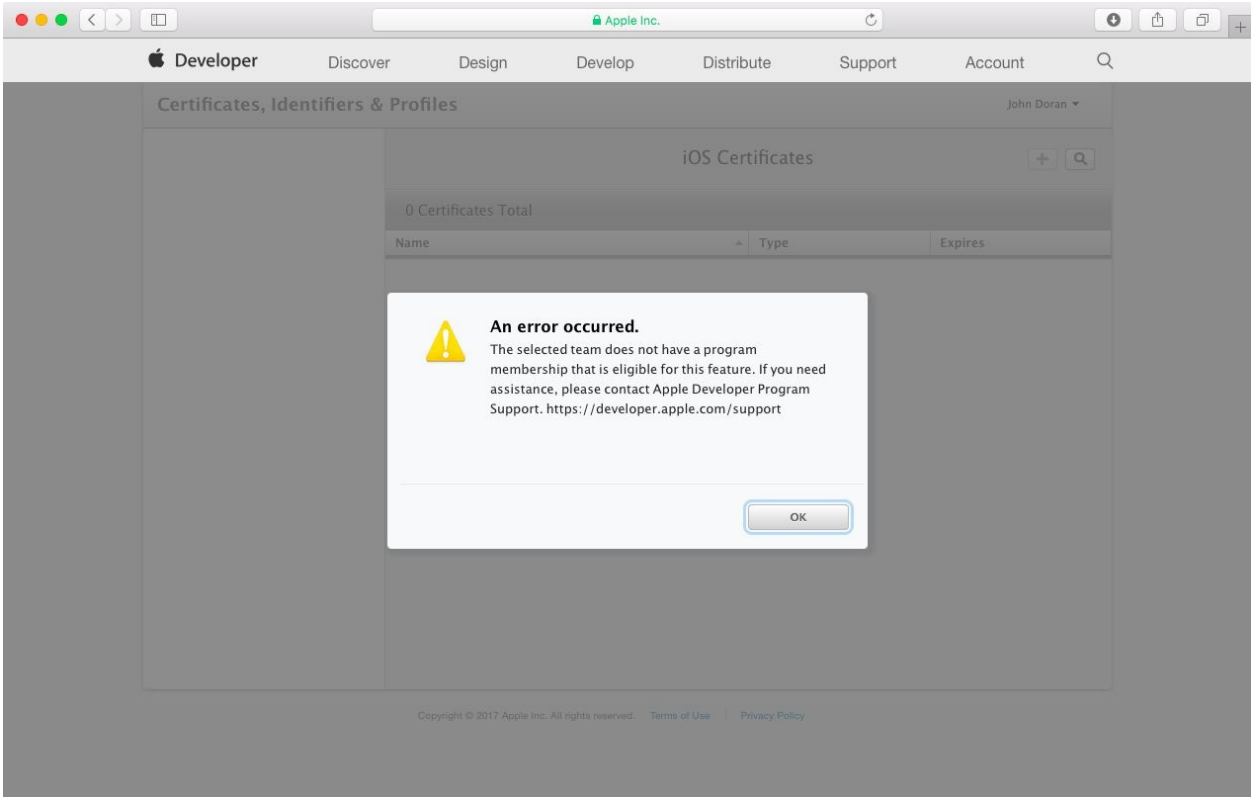
   If you have a two-factor identification, you may need to verify that you are yourself.

2. From there, click on Accounts. Now, at this point, you will need to make the payment for the $99 annual fee. This process should be fairly straightforward, and once you have finished that aspect of things, you will

come to a page similar to the following:

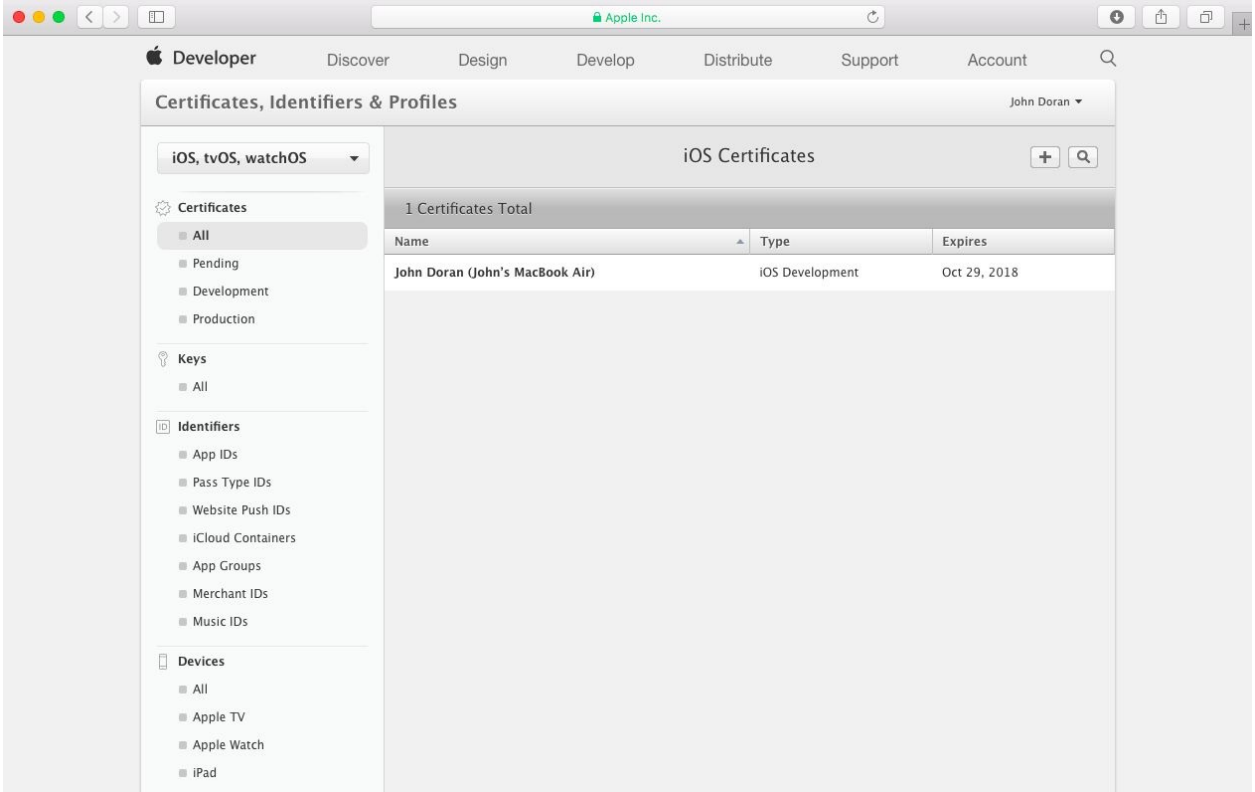


3. Select the Certificates, Identifiers & Profiles screen to start the process of creating apps. If you just paid the $99 fee, you may see an error stating that The selected team does not have a program membership that is eligible for this feature. If you need assistance, please contact Apple Developer Program Support. https://developer.apple.com/support, as you can note in the following screenshot:

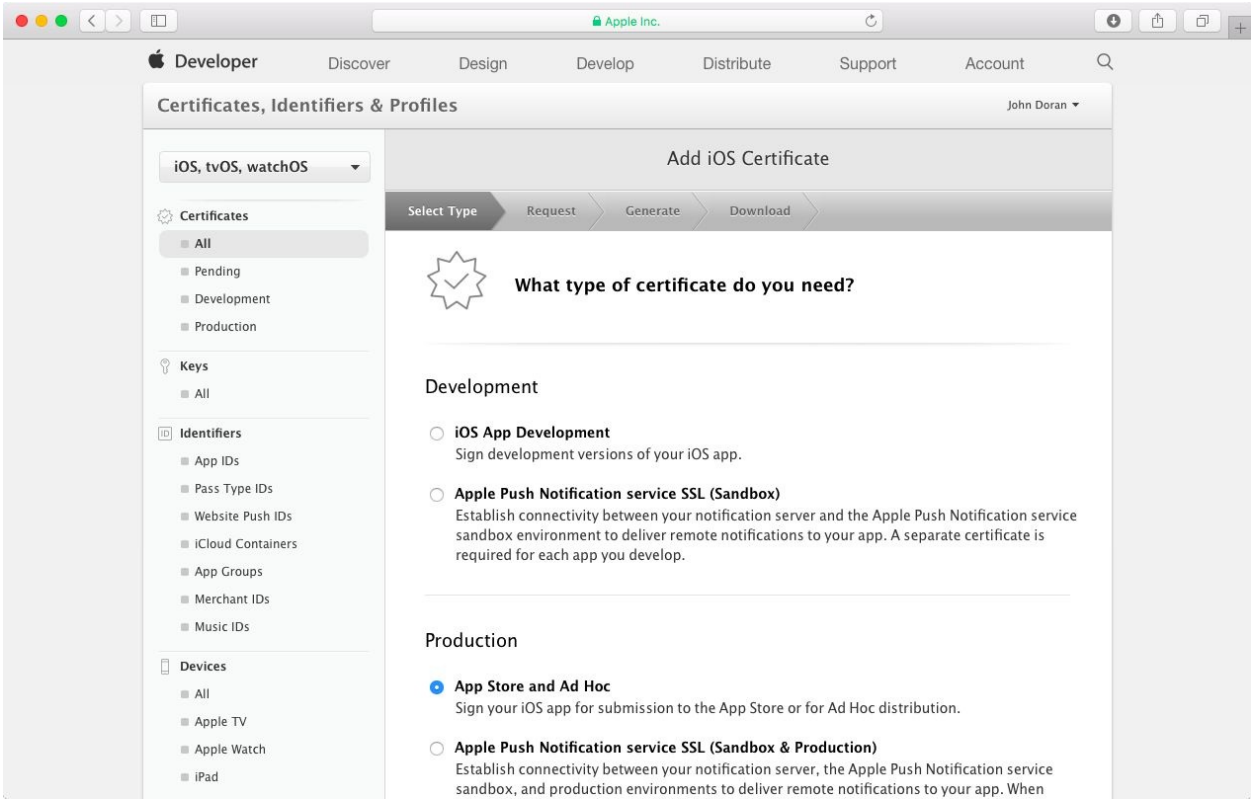


Do not worry, that just means that the payment hasn't processed from Apple's end yet. Try again in about 30 minutes to an hour, and the screen should work okay.

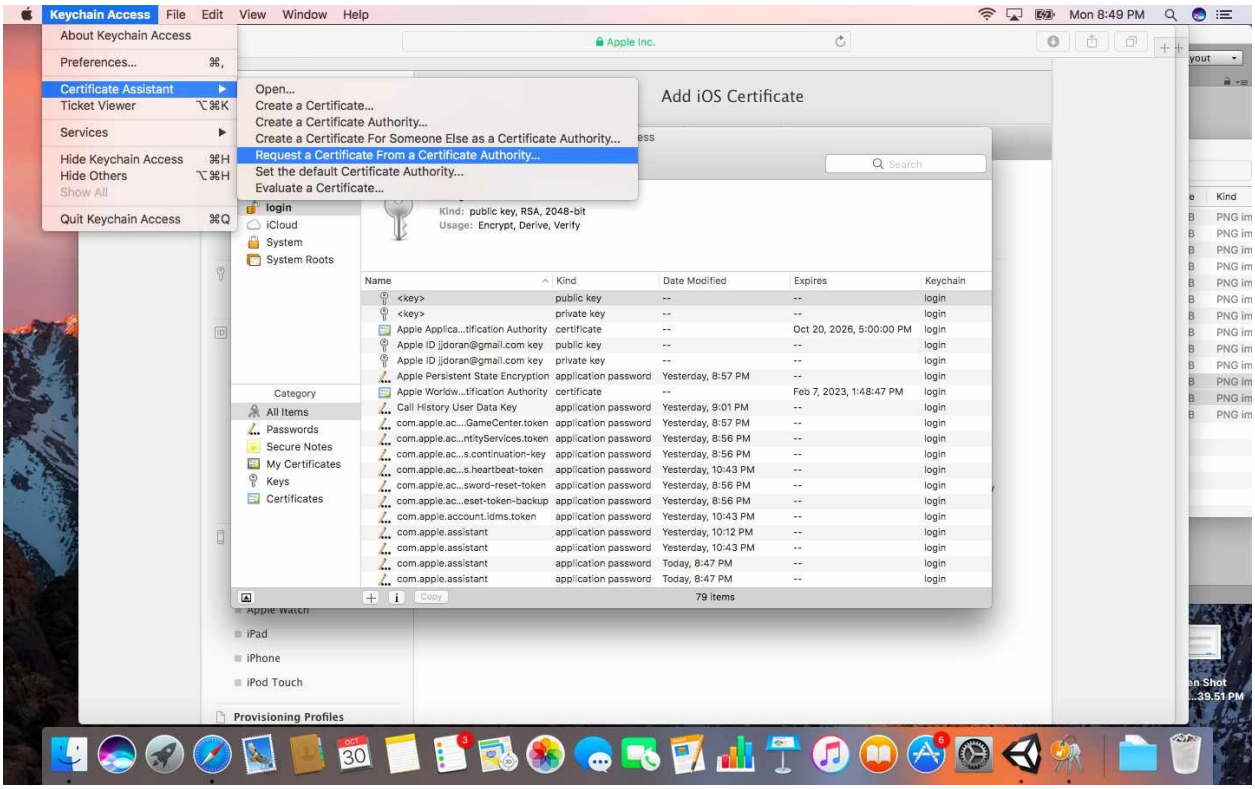
4. We will need to set up some certificates to allow us to export to the iOS App Store. From the All certificates page, click on the top left + sign on the screen:



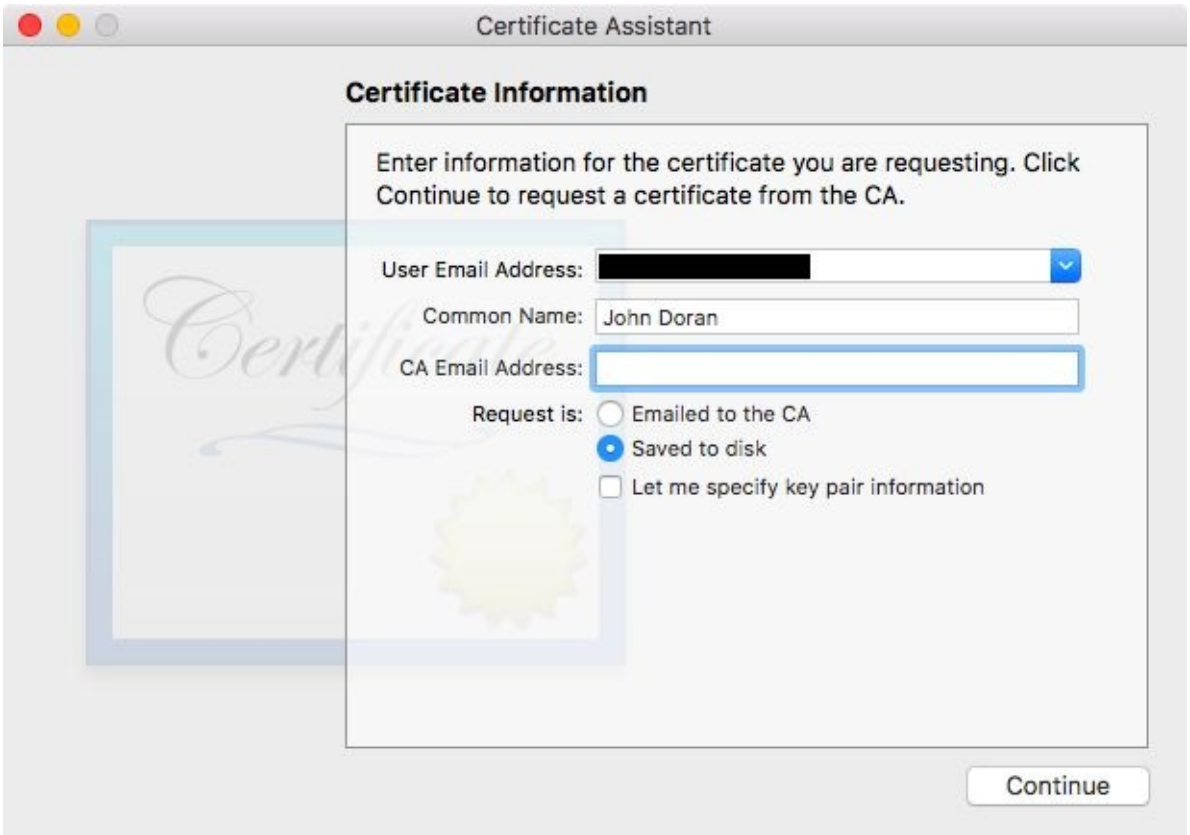
5. When the page asks what kind of certificate we need, select the App Store and Ad Hoc option under the Production section and then click on Continue:



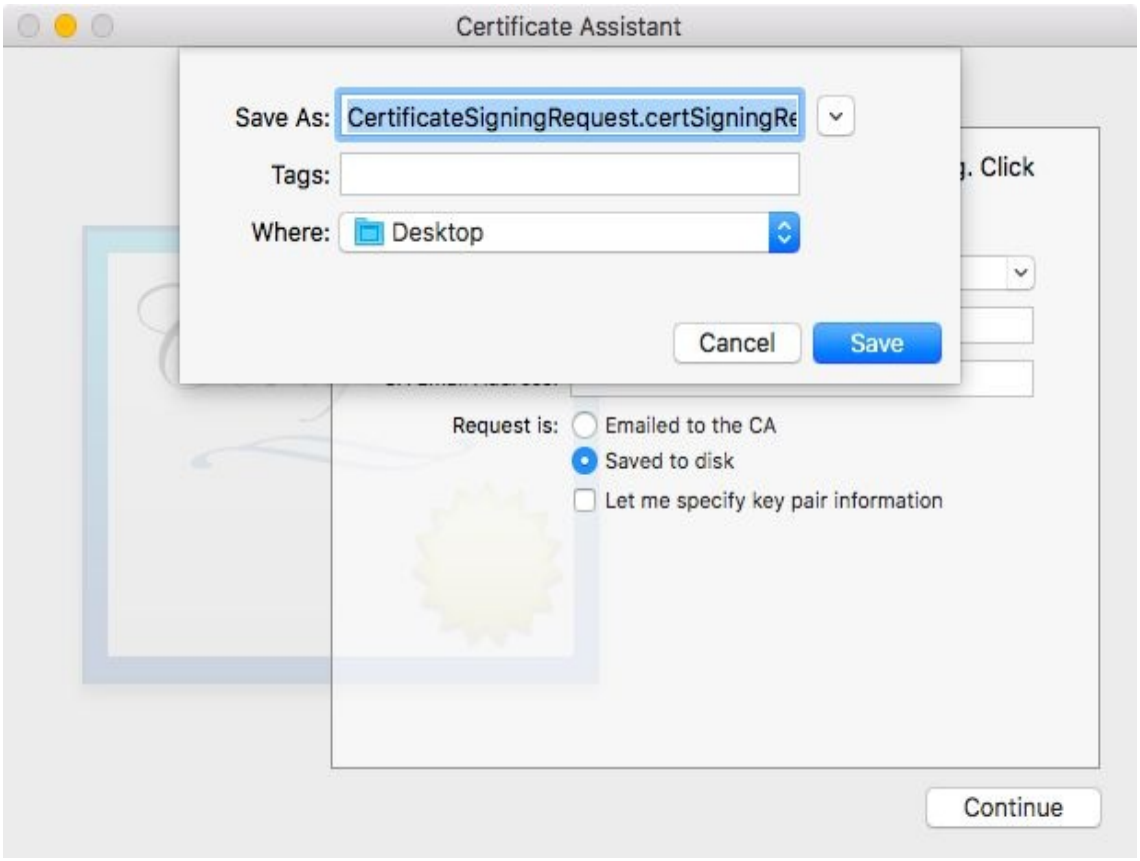
6. Next, we need to create a **Certificate Signing Request** (**CSR**). You'll be brought to a page that goes through the process of creating one, but, in our case, we will start off by opening the `Applications\Utilities` folder on our Mac and opening the Keychain Access program.
7. From there, go to Keychain Access | Certificate Assistant | Request a Certificate from a Certificate Authority... :



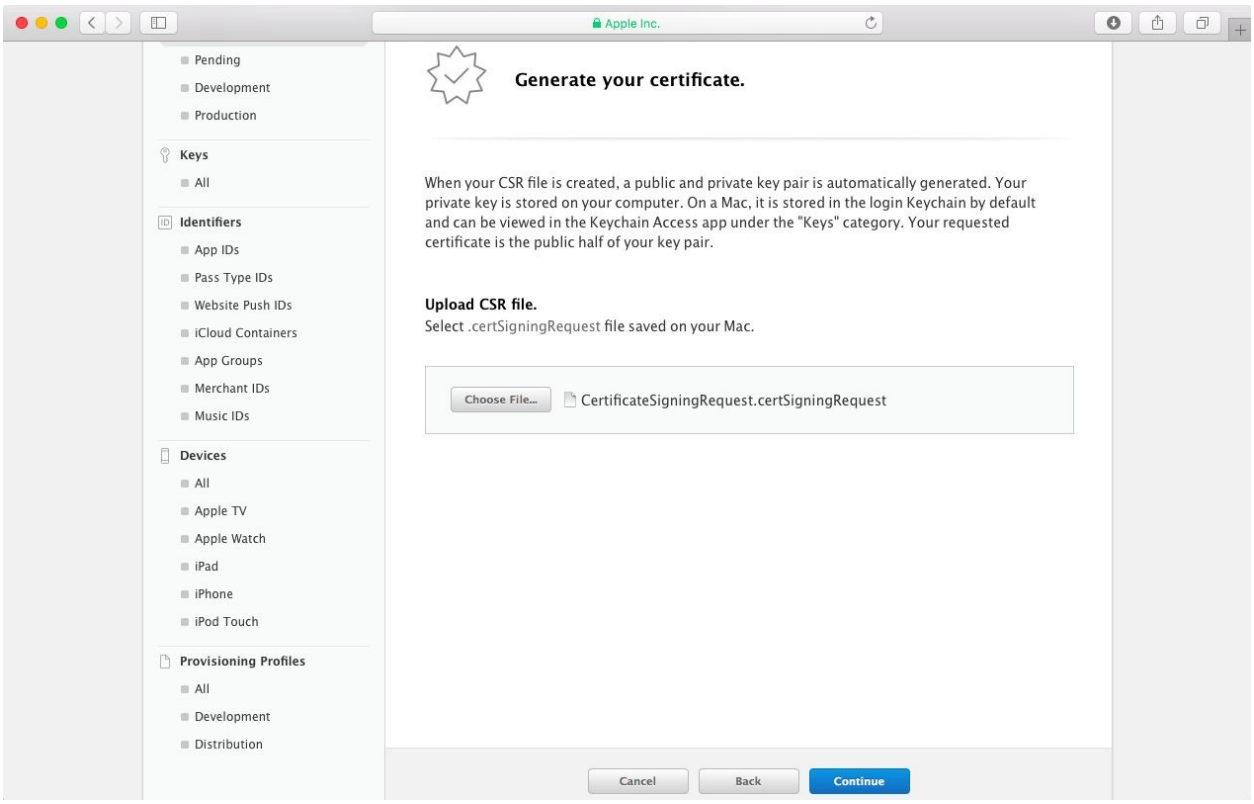
8. Once there, fill in the information with your email address in the User Email Address property, then for the Common Name, put in a name, leave the CA Email Address blank; then, under the Request is: property, select Saved to disk:



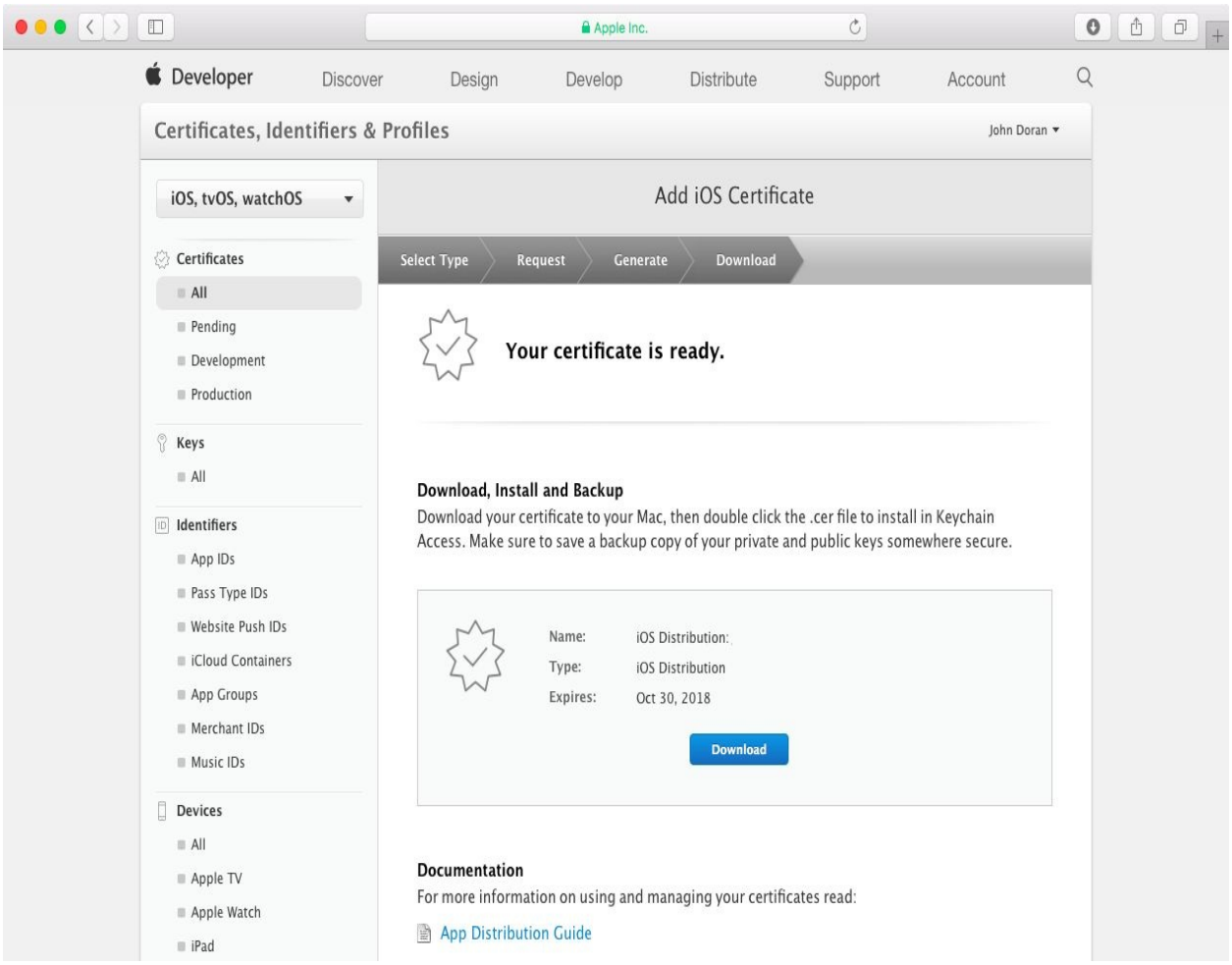
9. Then, click on the Continue button and select a spot to save it. I personally used my Desktop, but you can use anywhere as long as you remember where it is later on. Afterwards, it will state that the request has been created on the disk. Go ahead and click on Done and then return to your web browser:



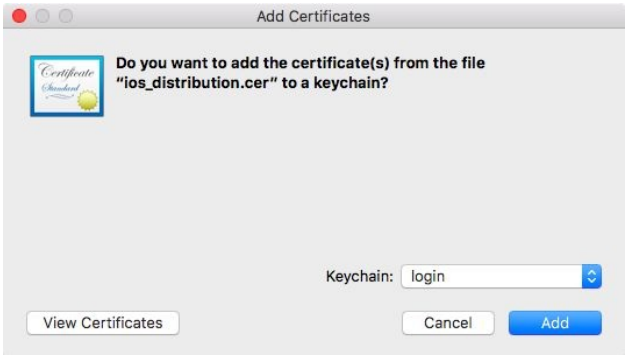
10. Scroll down and then click on the Continue button. From there, you'll be brought to the Generate your certificate page. Click on the Choose File button and then select the file we just created. Then, click on the Continue button:



11. You'll then be brought to a screen saying that your certificate is ready. Go ahead and click on the Download button and save it to your disk:

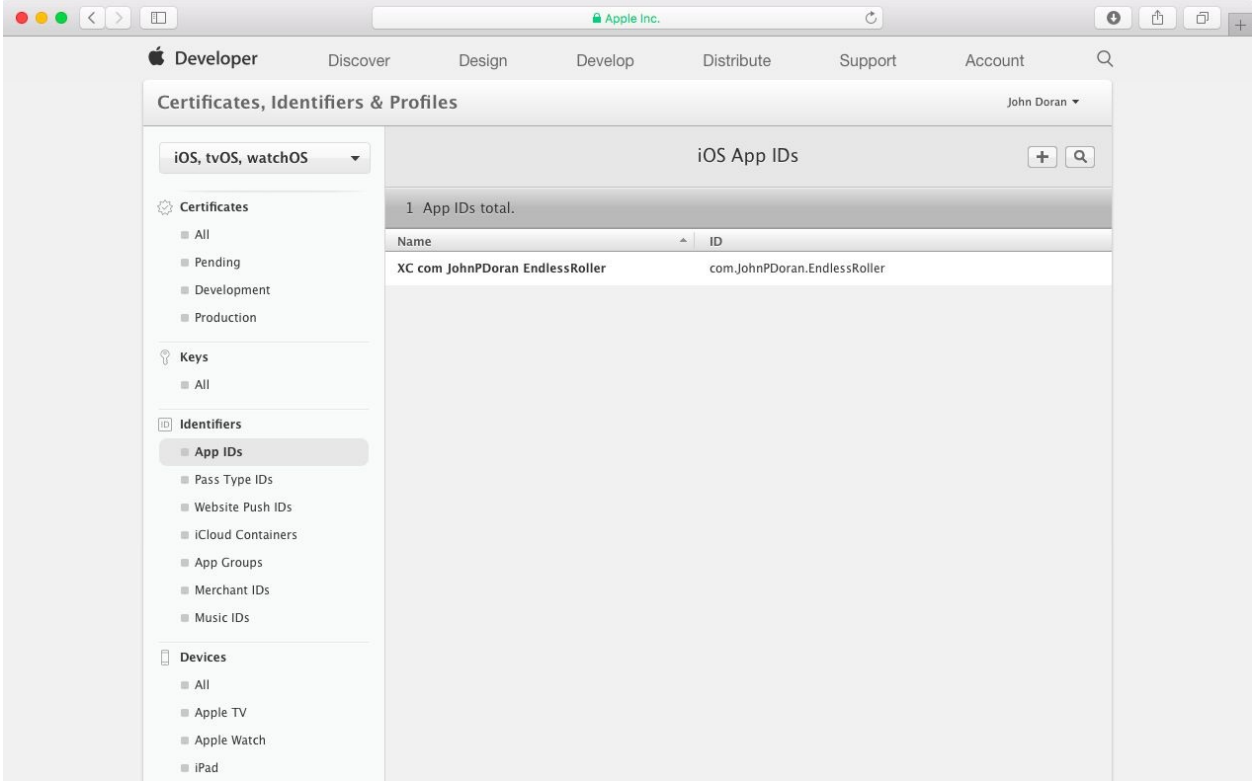


12. Afterwards, double-click on the `.cer` file to install the data into Keychain access. You'll be asked whether you want to add the certificates; go ahead and click on Add:

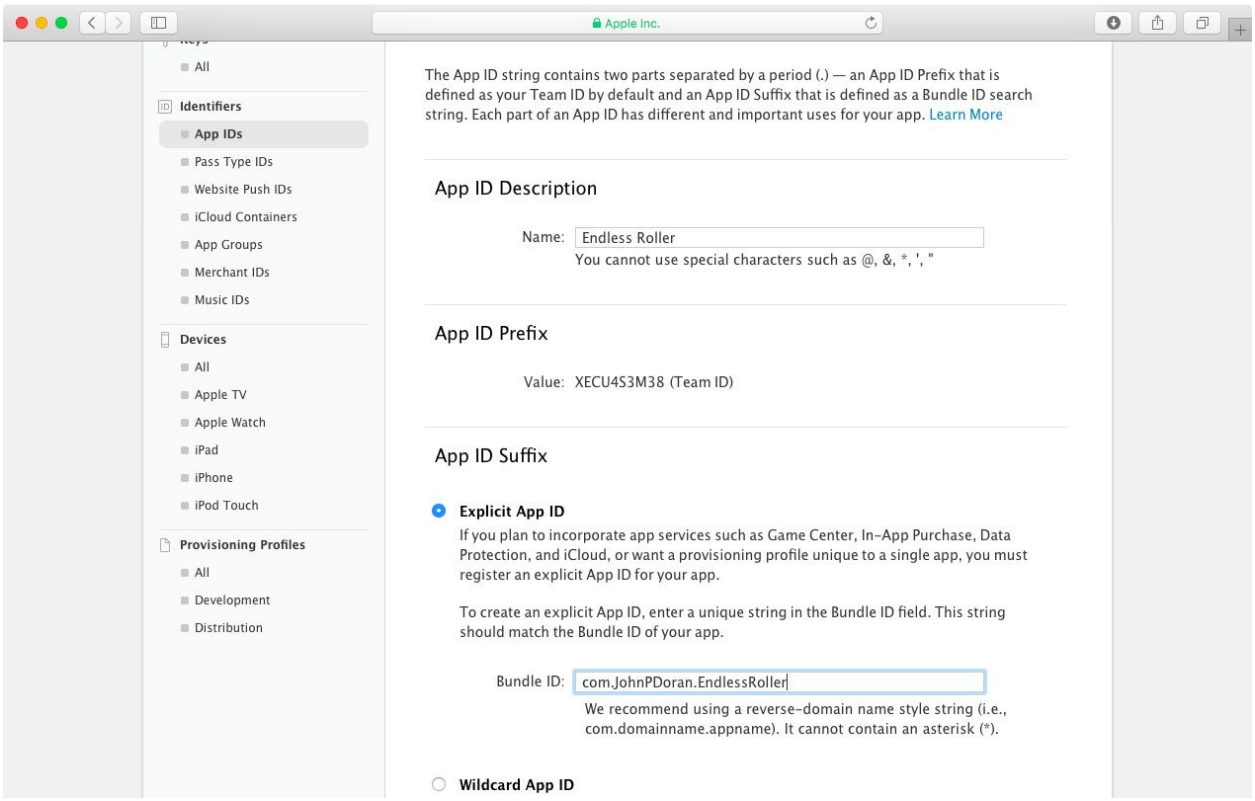


13. The next step is to create an App ID. To do this, go to the left-side bar and click on the iOS App IDs section. I currently have one App ID already due to Xcode opening our `Endless Roller` project, which we can customize by

clicking on the Edit button; however, if you didn't do so earlier and have a different Bundle ID than the ones listed, let's go through the details next:

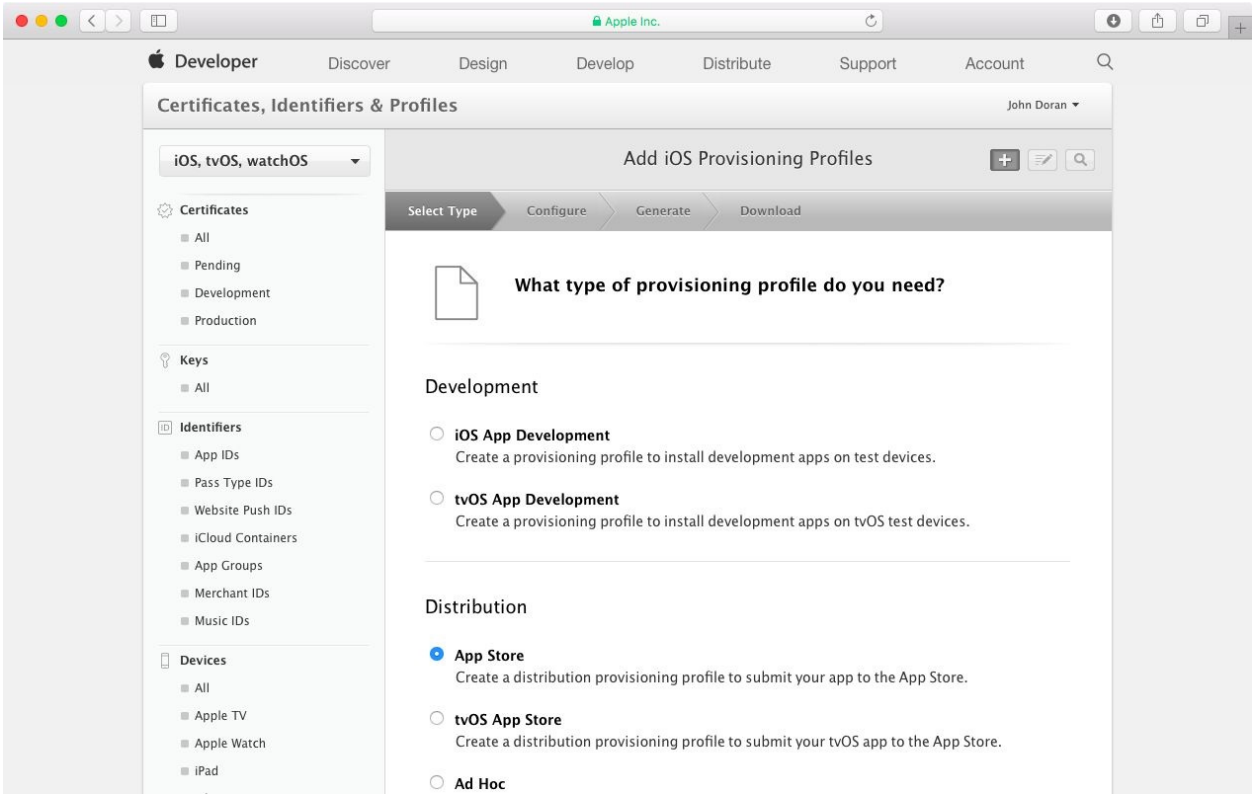


14. We can create a new ID by clicking on the + button on the top-right corner of the screen.
15. From there, under App ID Description, put in the name of your game--in my case, I used `Endless Roller`. Then, under the App ID Suffix, put in the Bundle ID in the same manner as it was in Unity. In my case, it was `com.JohnPDoran.EndlessRoller`. Under App Services, select the options that you are using, but, in this case, we're not, so we can just scroll all the way down and then click on the Continue button:

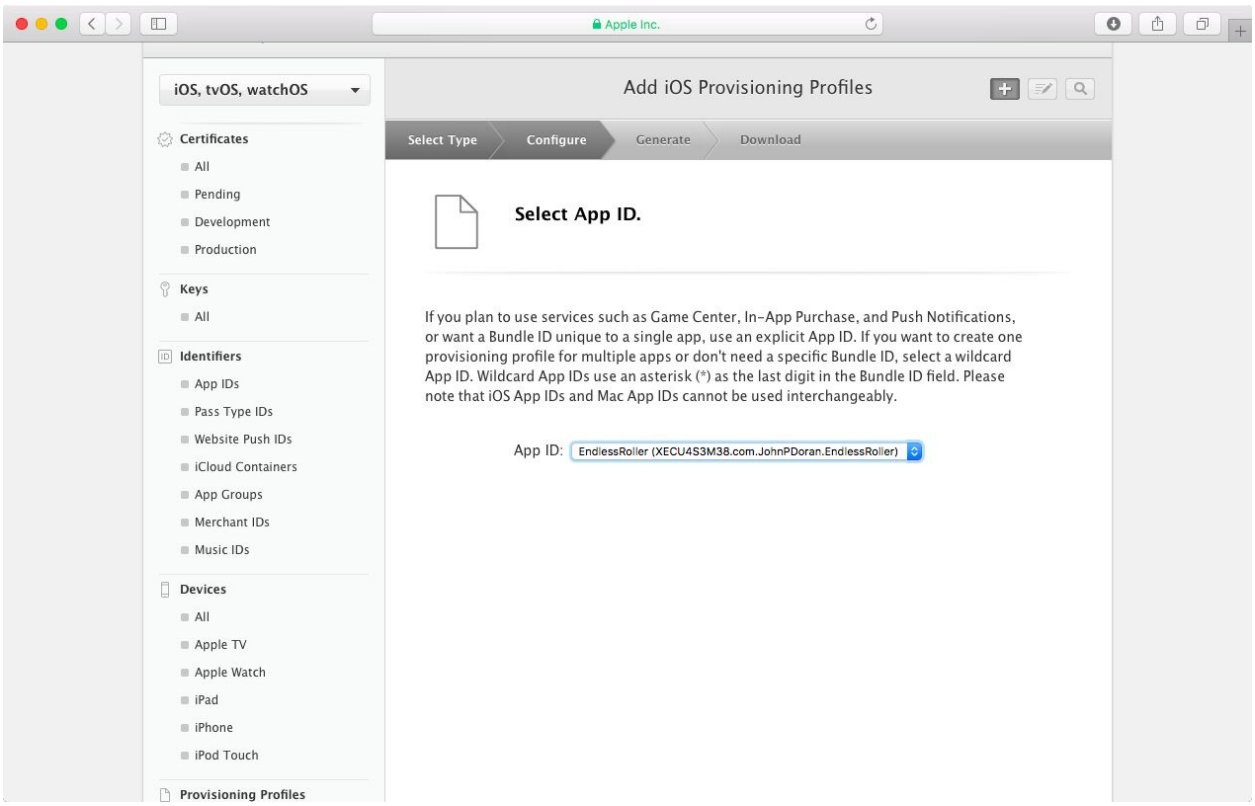


In this case, this would not work due to there already being an ID with this specific Bundle ID. This is why you need to have unique ones. With that in mind, I just went in and edited the original App ID to `Endless Roller` and then completed it.

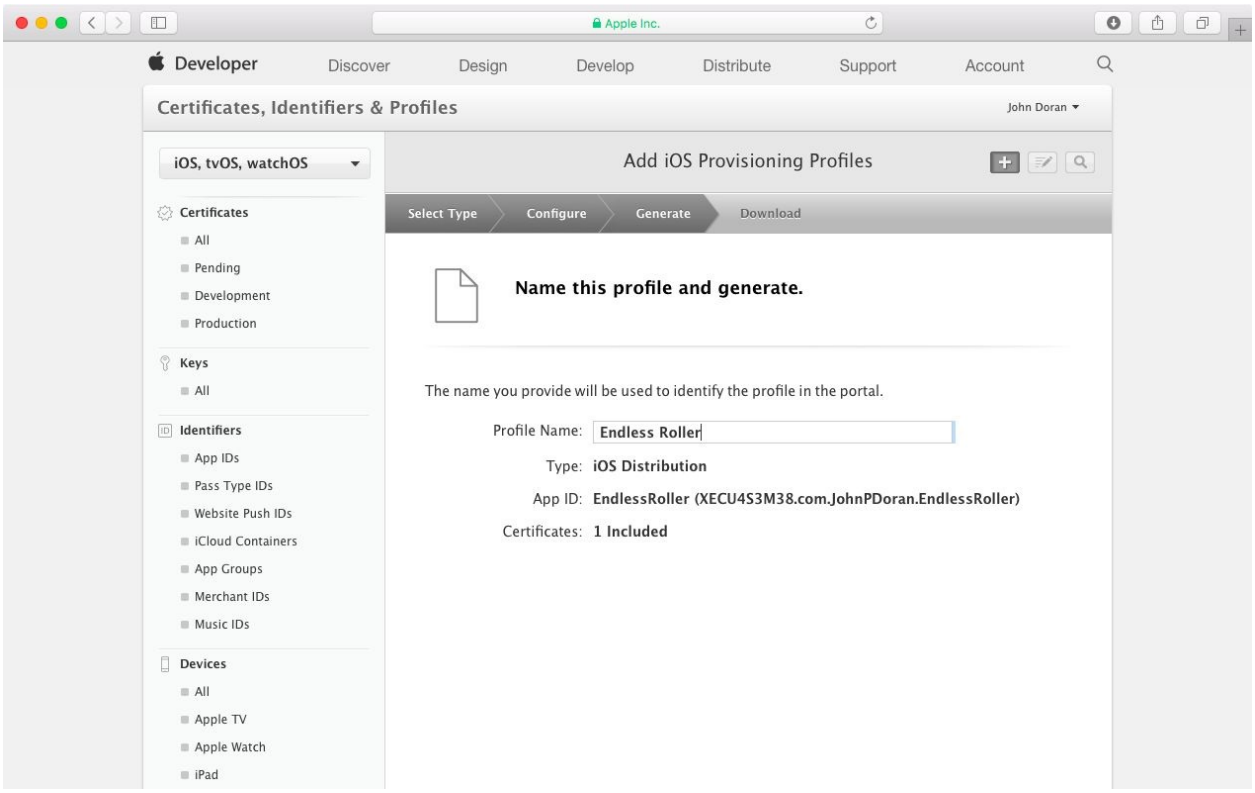
16. The last aspect we will need to set up here is a Provisioning Profile. To do this, click on the All button under the Add iOS Provisioning Profiles section. From there, click on the + on the top left. Under Distribution, you are going to select App Store and then click on Continue:



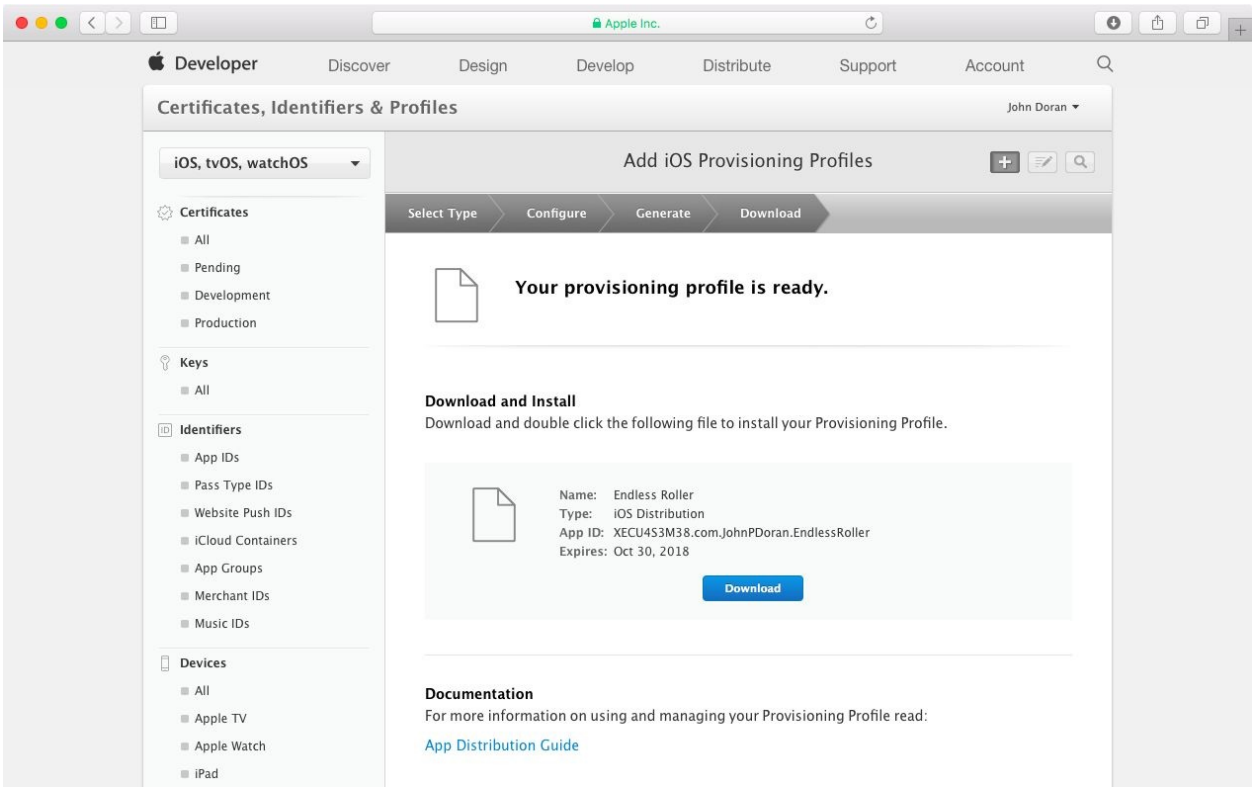
17. From there, you'll need to select your App ID. `Endless Roller` may be selected, otherwise, search for it in the drop-down list and select it, and then click on Continue:



18. Then, select your certificate and click on Continue.
19. Finally, we will need to put in a Profile Name--I'll put in `Endless Roller`--and then click on Continue:



20. You'll be brought to a page with the profile. Go ahead and download it and keep it safe as we'll need to use it later:

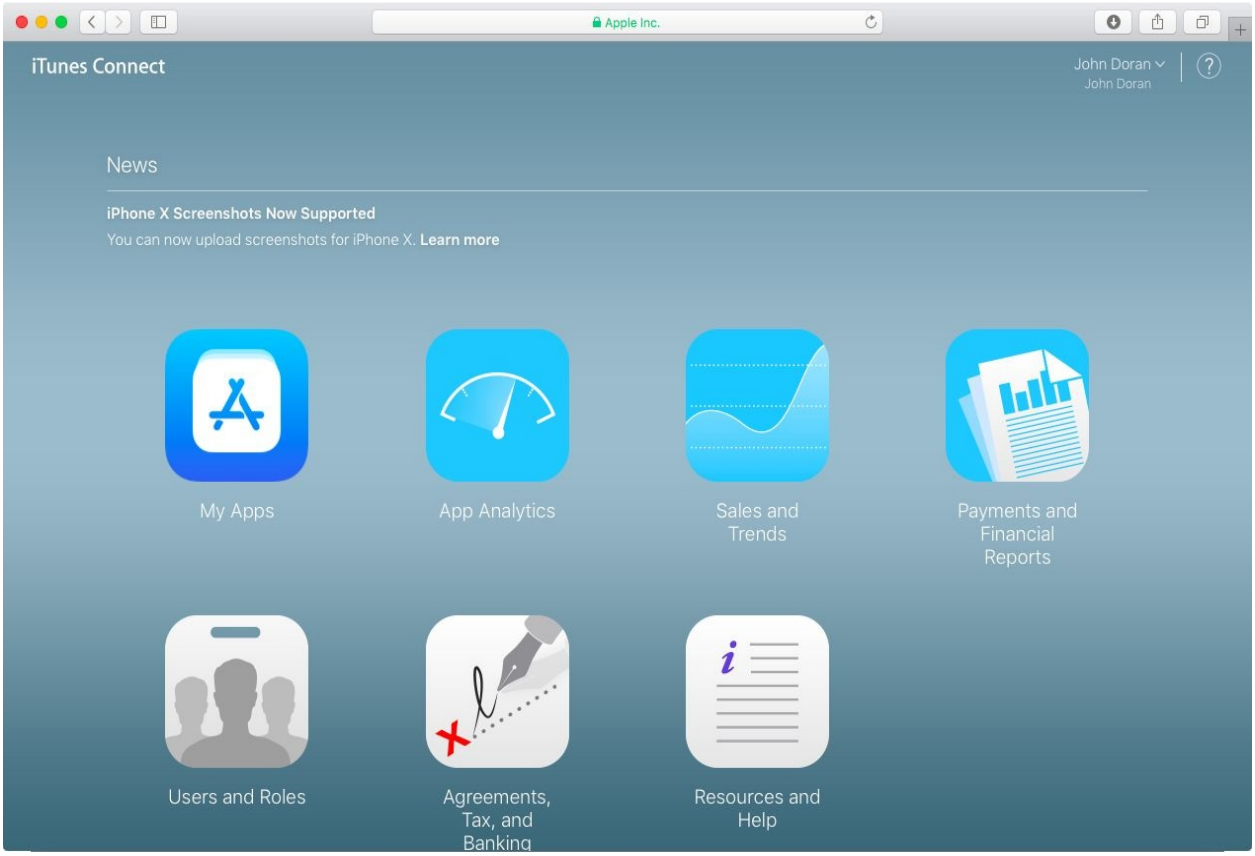


With that, our provisioning profile is ready.

Adding an app onto iTunes Connect

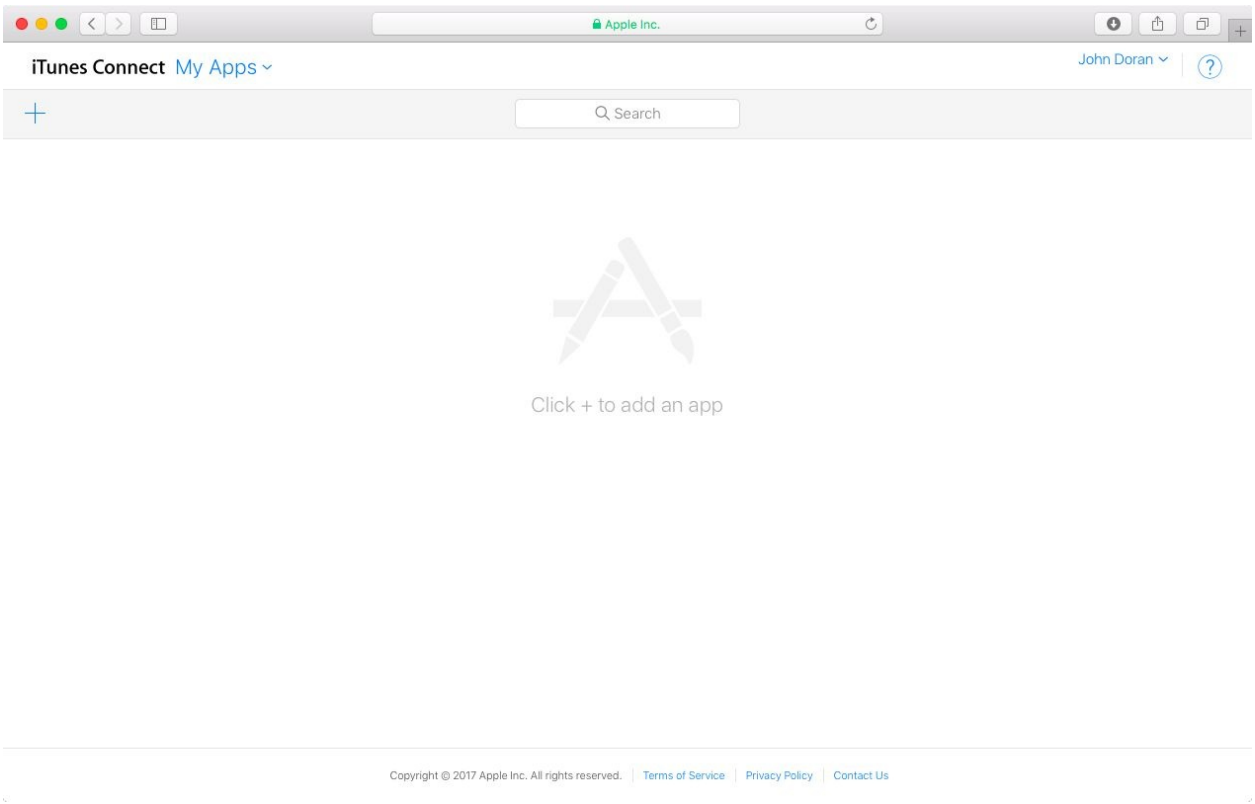
Now that we have the provisioning profile, we can actually put our app on the store; to do that, perform the following steps:

1. In your web browser, go to `itunesconnect.apple.com` and click on the My Apps button:

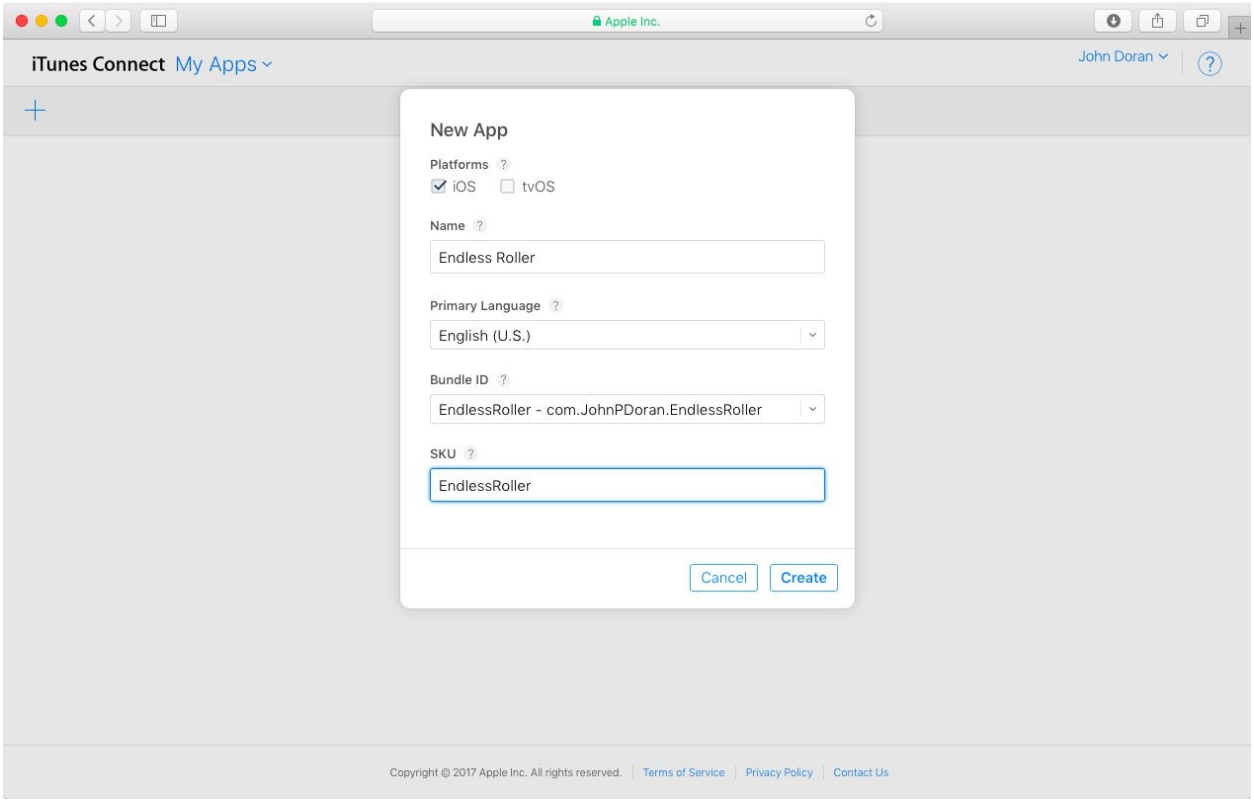


   If you intend to sell your apps, you will also be required to go to the Agreements, Tax, and Banking section and put in your banking information.

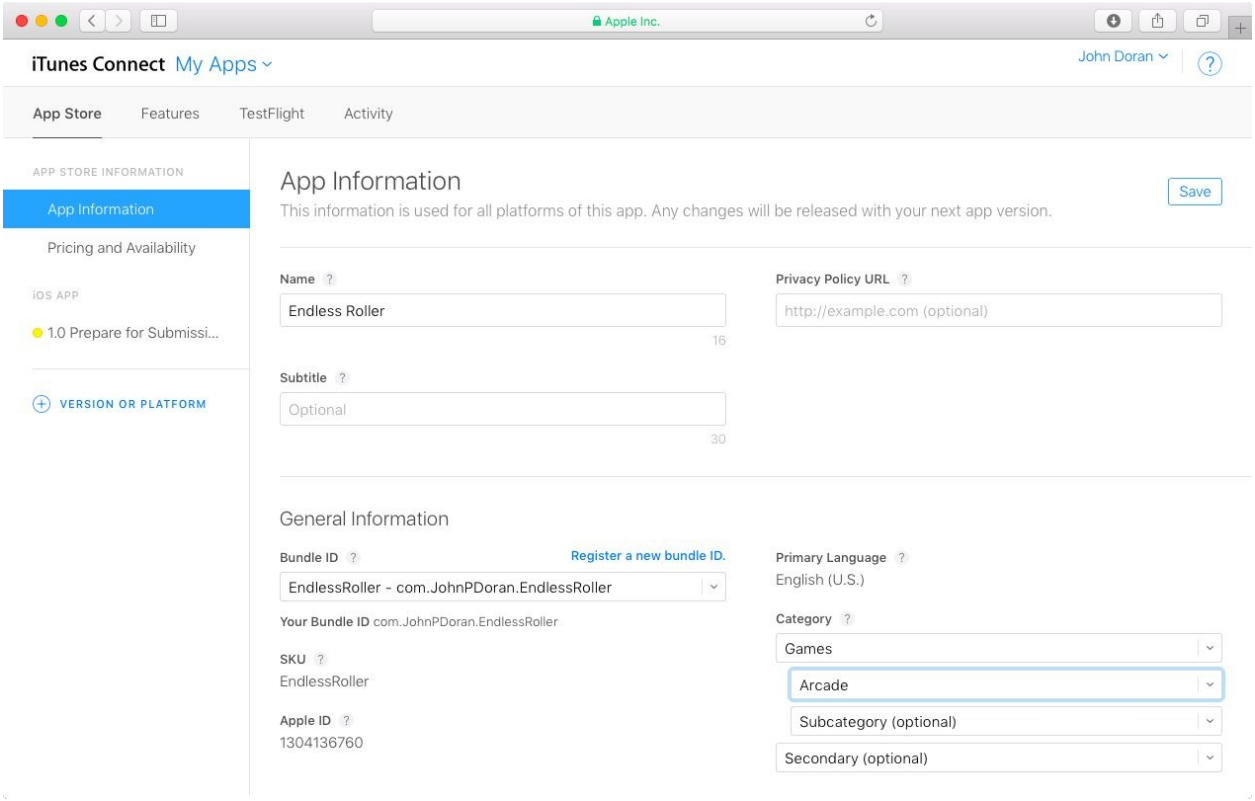
2. From there, go to the top-left corner and click on the + icon to add a new app to our profile by selecting New App:



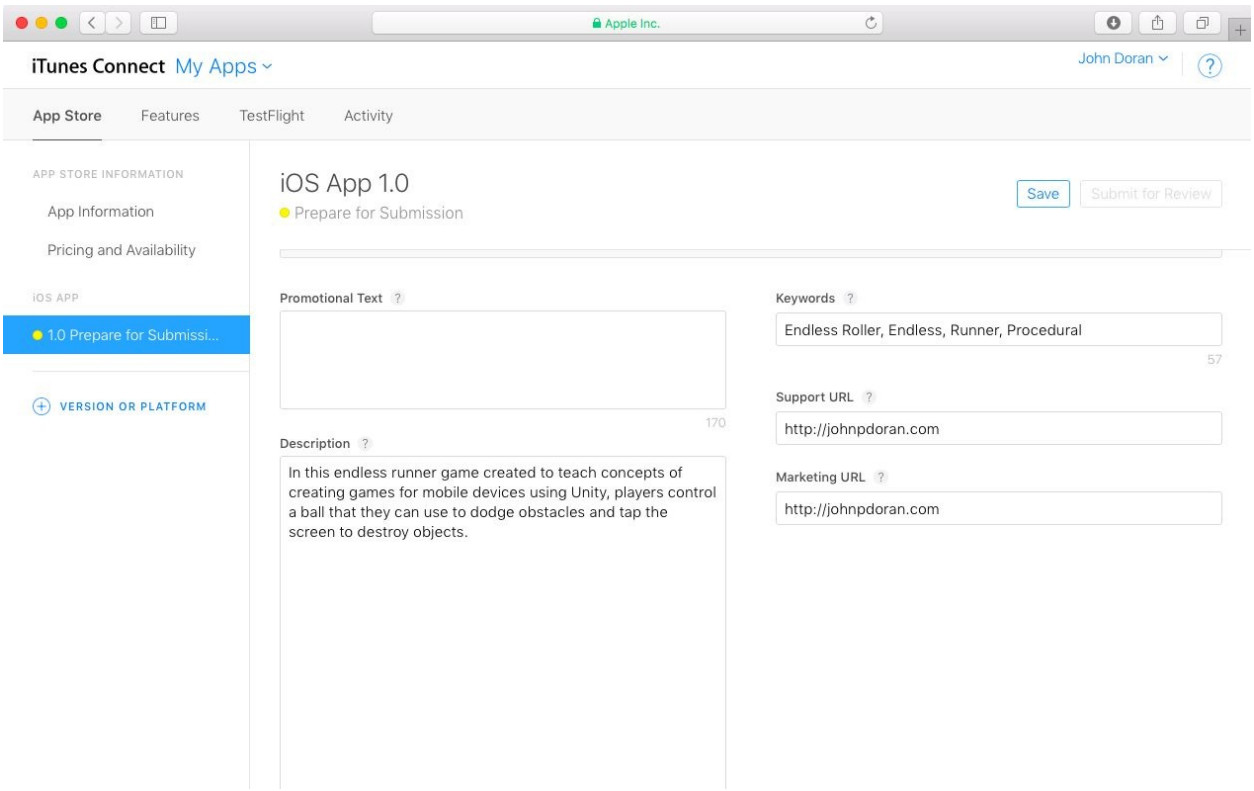
3. On this menu, fill in iOS as your Platforms and put the name of your game under Name. Apple requires each name to be unique, so keep in mind you will not be able to use `Endless Roller` again. Under Primary Language, select English (U.S.) and then select your Bundle ID and then under SKU put in an identifier (I used `Endless Roller`). Then, click on the Create button:



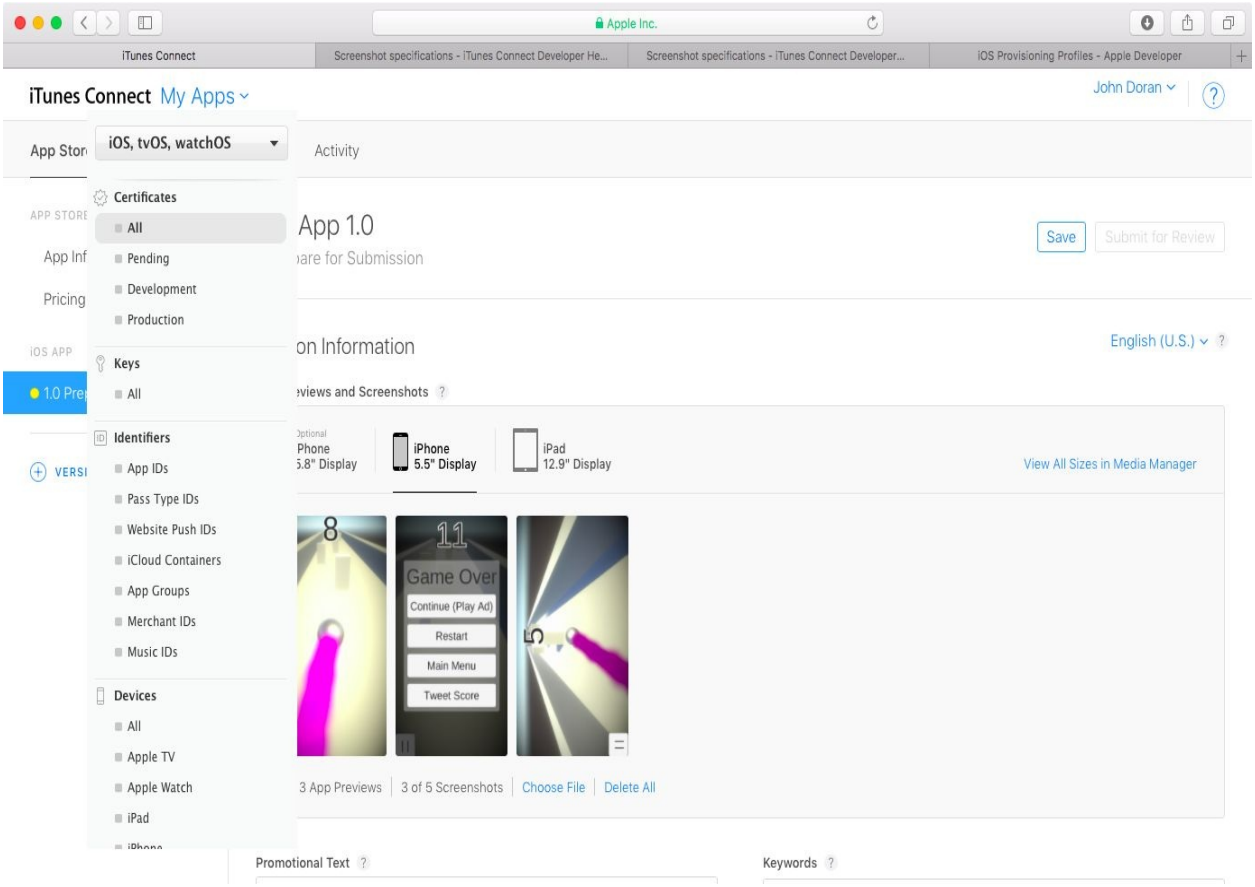
4. You'll then be brought to the App Information screen. From there, change the Category to Games and then under the Subcategory, put in Arcade, then click on Save:



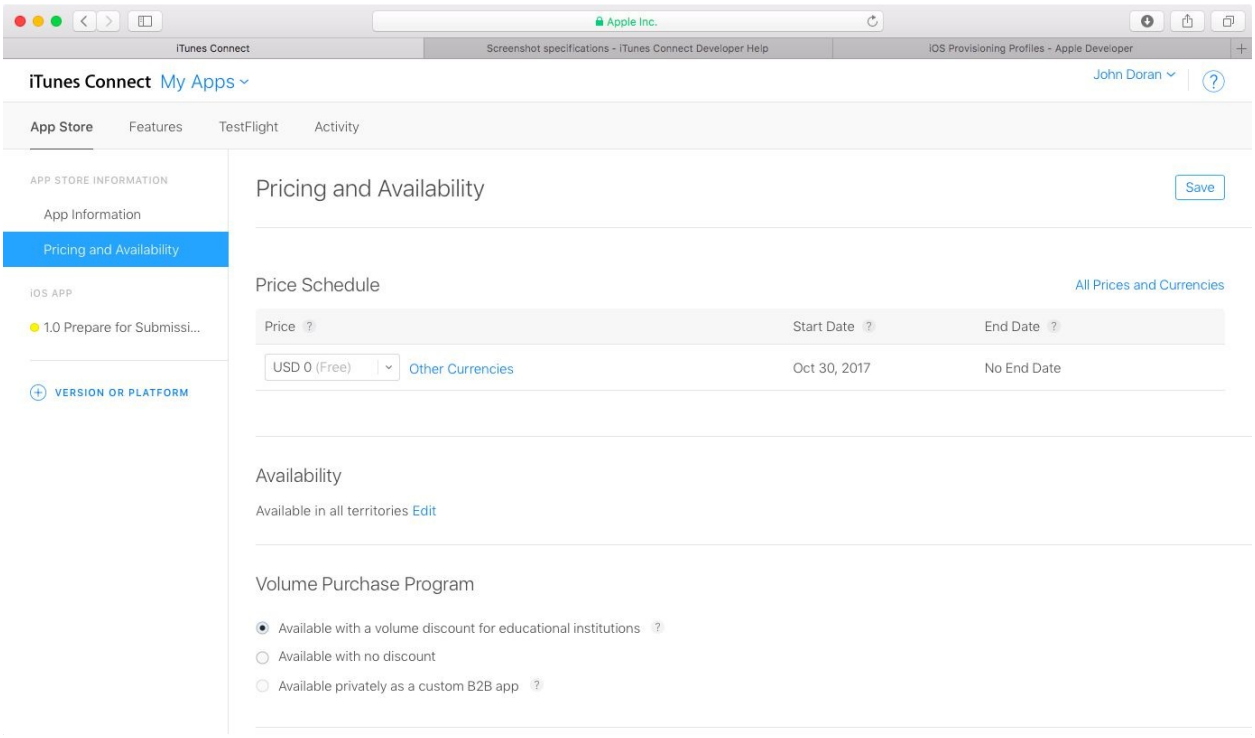
5. Next, go to the 1.0 Prepare for Submission section and click on it to start filling in the information for the title. Start off by filling in the Description textbox with the information that you used earlier on Google Play. Then, under Keywords, put in possible things that people would search for to find your game:



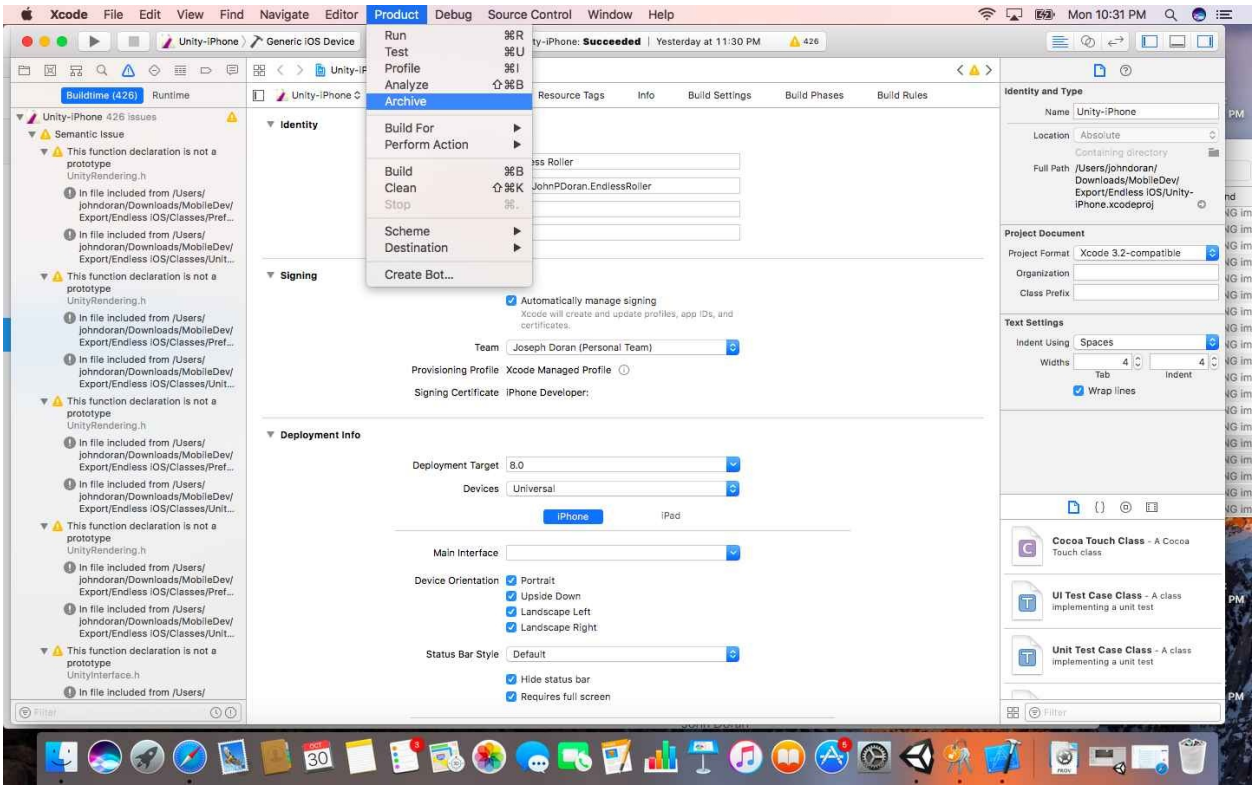
6. We'll then need to provide an App Icon to be used. The image must be 1024 x 1024 ina PNG format. Under Copyright, go ahead and put your name.
7. Lastly, you'll need to provide some screenshots of your game to use. If you click on the iOS Screenshot Properties page, you'll see details on how your screenshots should be created (specifically, the size of the images). The one used in this chapter is for the iPhone 5.5" display, but you can also submit for the optional 5.8 one to support the iPhone X:



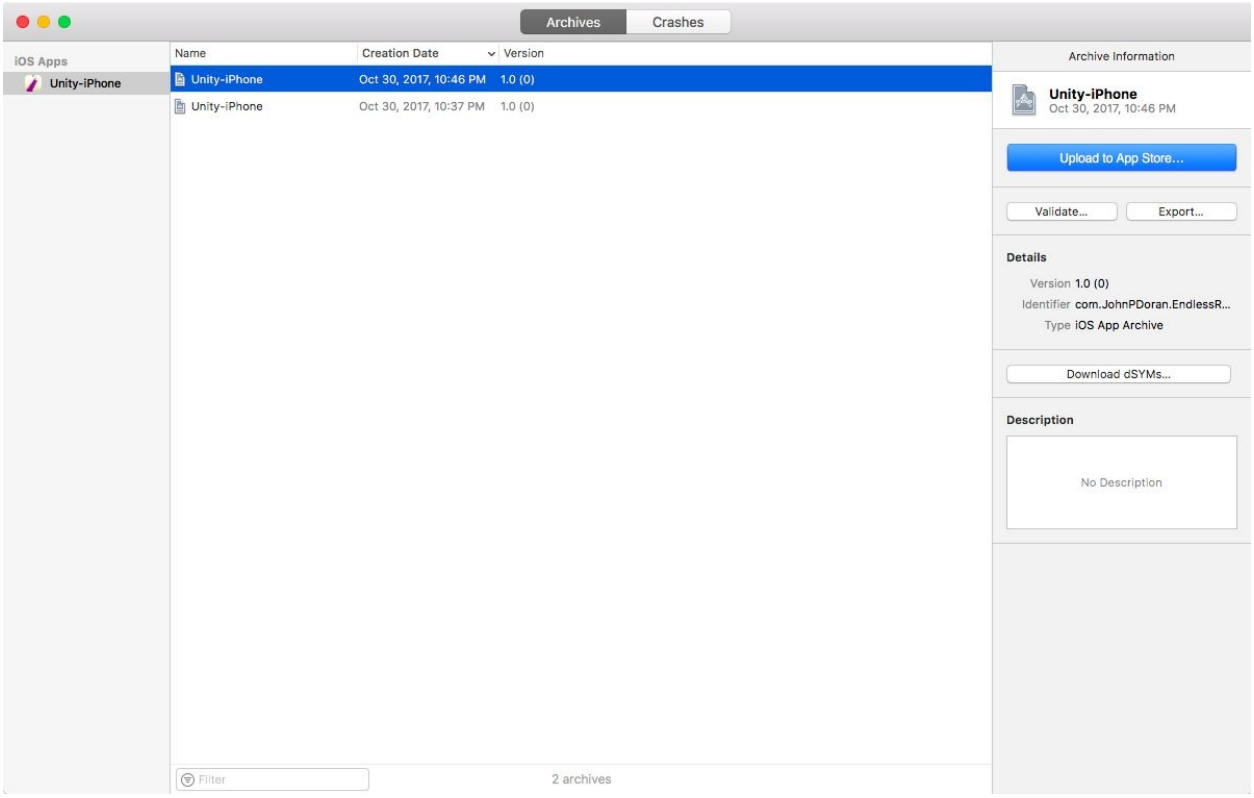
8. Note that in the Build section, it states that you need to submit your build using Xcode. Let's go ahead and do that after we finish up the last step.
9. Go into the Pricing and Availability section and select a price. In my case, I'll be using USD 0 (Free), but, as always, you can pick what you'd like. Since there's no cost under the Volume Purchase Program, go ahead and select Available with no discount since there's no reason for there to be one, and then click on the Save option:



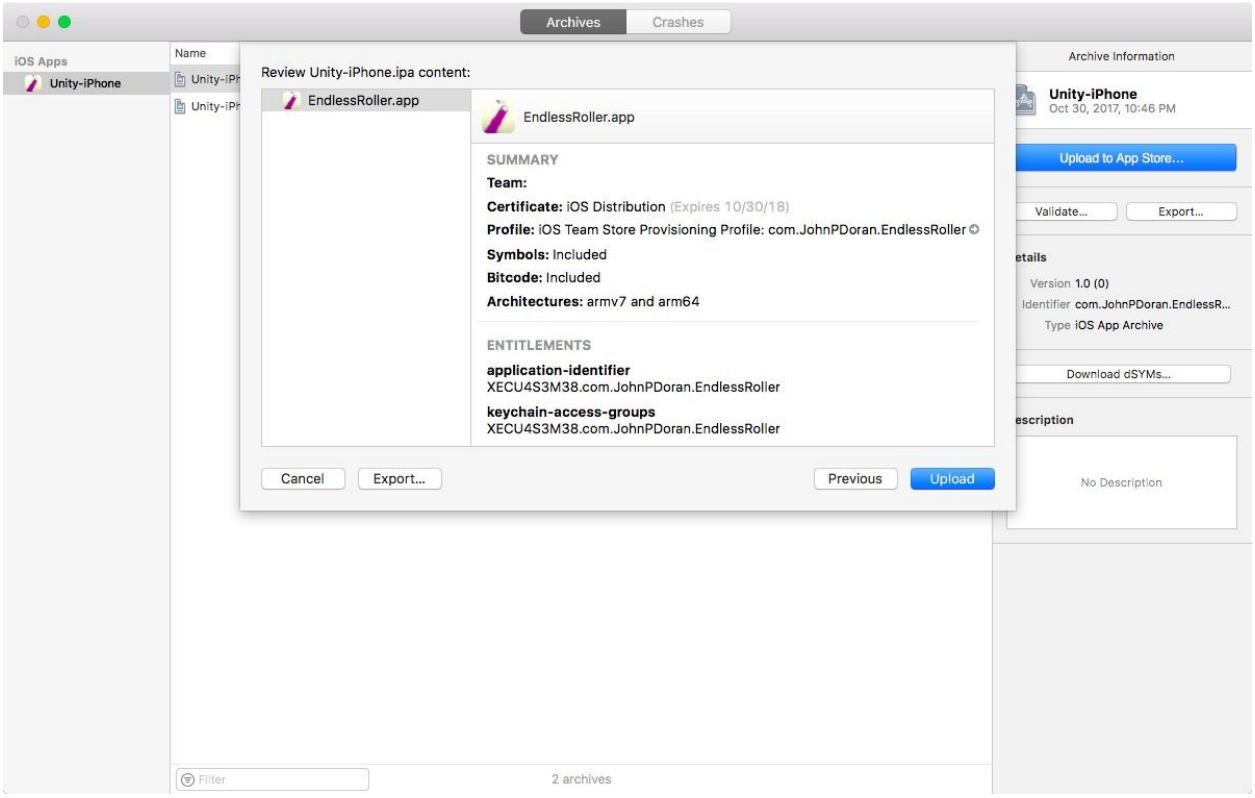
10. Once all of the information is filled in, go ahead and open up Xcode again and your exported project (follow the same steps as in Chapter 2, *Setup for Android and iOS Development*). From there, go to Product | Archive and wait for it to finish:



11. This generally takes a while, so wait for it to complete. You may be asked to use an access key; go ahead and click on the Allow button.
12. Upon finishing, you should be brought to the following menu. Go ahead and select the Upload to App Store... button:

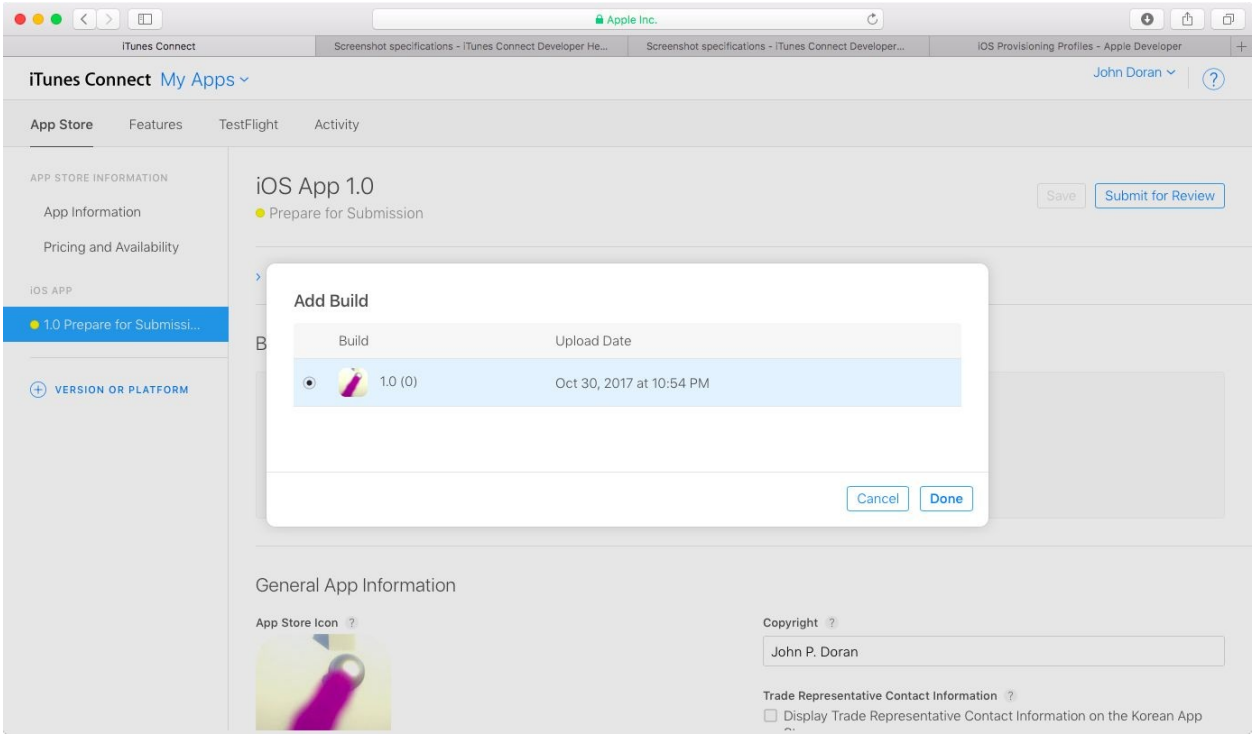


13. You'll be asked to select some options. In general, use the default options, and afterward, it will give you an `.ipa` file uploaded to the store. Before uploading, it will give you one last look with information about each aspect of the project. Go ahead and click on the Upload button and wait for it to finish:



This will not show up immediately on iTunes Connect; you may have moment or a couple of hours before it's updated. However, once it is r see it under the build section we mentioned earlier.

14. Once it's loaded up, you should be able to click on the Select a build button before you submit your app option.
15. From there, select the build we created and then click on the Done button:



16. Then, click on the Save button. Once finished with everything and double-checking all of your information, you can go ahead and click on the Submit for Review button to wait for feedback from Apple.

Generally, it takes up to 3-4 weeks for first-time developers to receive feedback, although it can be longer or shorter, depending on the season. As you release more and more titles, it takes less time each time around. If approved, you'll receive an email that lets you know that the app is up or they will have details on things needed to be approved before being placed on the store.

Summary

With that, you learned how to publish our games on both the Google Play and Apple iOS App Store. You first learned how to build a release copy of your game, then learned how to put the game onto Google Play by setting up the Google Play Console and then finally to publish your app on the store.You then learned how to put a copy of the iOS version of your game on the App Store and all of the setup involved there.

I hope that you've enjoyed this exploration of features and that you continue to explore the possibilities of this space. Feel free to let me know what you're up to and I wish to see your projects up on the market and see what you come up with.