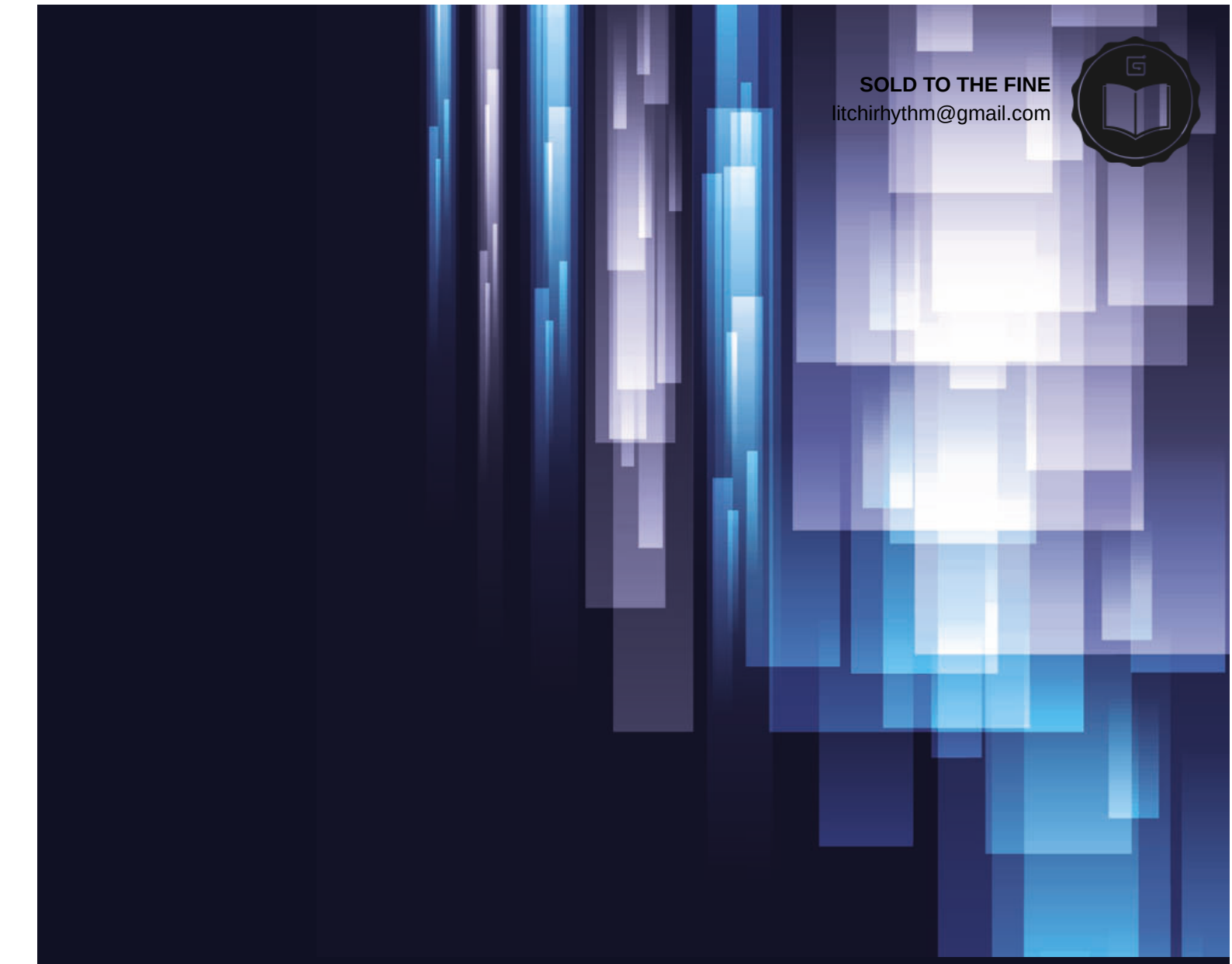


Weekend Code Project: Unity's New 2D Workflow

A Quick-Start Guide To Making 2D Games In Unity 4.3

By Jesse Freeman

Table Of Contents

Book Resources

Thank you for buying a copy of my book. I hope you enjoy the project and I have spent extra time to make sure that you not only have all the lessons you need to make a simple 2D game in Unity 4.3 but have the artwork and code to do it as well. To get started, here is a list of all the things you will need:

1. Download the free copy of Unity 4.3+ for Mac or PC from http://unity3d.com/unity/download.
2. Download the Super Paper Monster Smasher Art Pack from http://bit.ly/spms-free-art.
3. Don't like typing out the code? Here are the scripts in the book http://bit.ly/u2d-scripts.

This should be everything you need to actually follow along in the book. There are a few other apps I highly suggest to help you make 2D games. While I don't cover these in the book, it's always helpful to know some additional tools for making artwork, packaging that artwork and creating sounds, which are all critical to any game. Here is a list of some of the apps I use on a daily basis:

- Aseprite – If you are doing pixel art, this is one of the best editors out there. Not only is it free and open source, it's ideal for making pixel animation and even supports importing and exporting sprite sheets.
- Bfxr - This app is perfect for generating simple 8-bit sound effects for your game.
- Shoebox – This is a great app to help you package up your artwork into sprite sheets and texture atlases.
- Audacity – Sound editor and file converter, which is useful for cleaning up sound effects and looping music as well as converting them into different file formats.

If you enjoy this book and my work, please make sure to visit my website for more great articles on game development and signup for my mailing. If you see an error in this book or have a question? Feel free to send me an email at info@gamecook.com.

Also, special thanks to Kristen Kelly (editor), Chaim Krause (tech editor), Sean McCracken (code inspiration), Joel Hooks (motivation) & everyone who bought the book. Cover image source http://bit.ly/1dfCTJz. Version 2.0.0 February 11, 2014

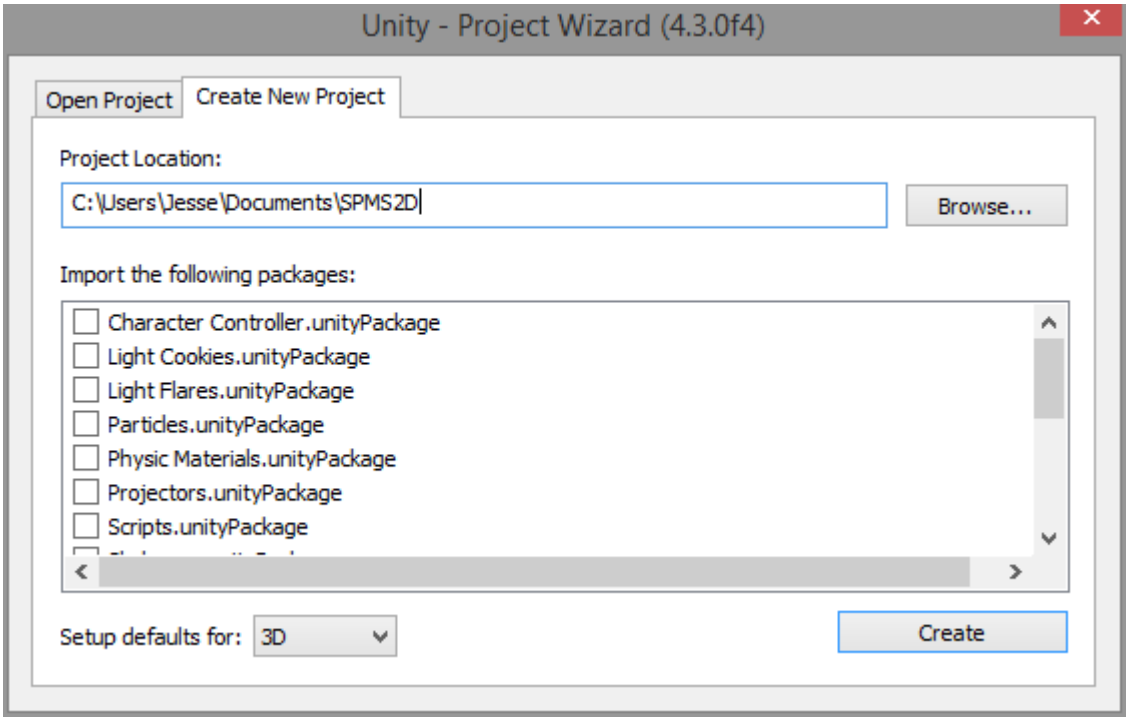
Introduction to Unity

Right now, Unity is the de facto game framework and IDE for a lot of the success stories you read about on multiple platforms. The IDE is very polished and easy to use. Previously, with Unity being a 3D tool meant there was a certain level of knowledge you needed before getting started. Now with the addition of an all-new 2D workflow, things have gotten a lot easier for Game developers looking to build simple, non-3d games. Unity supports three languages: Unity Script (which is similar to JS), C#, and Boo. Unity now has a free version that supports exporting to desktop and mobile but forces you to display the Unity logo when the game starts up. The Pro version gets pricey but adds lots of must-have features for more advanced game developers. There are also additional licenses you can get that export to console, such as the Xbox, PS3, Wii U, and more, but those licenses cost even more. Unity has a Web Player that relies on a plugin, so while you can export your games to the Web, they won't be playable on mobile browsers.

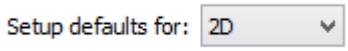
Over the next section, we will take a look at the IDE itself and how to navigate around it. There are a lot of resources out there on Unity, but for people who have never opened it up before, this will help get you acquainted with the basics in the IDE.

Creating a New Project

When you create a new project in Unity, you will see the following wizard:



As you can see, you are given an option to set the location of where you want to create your project, as well as packages you can include when it is created. The final thing to note, which is new in Unity 4.3, is the 2D setup tab at the bottom of the window.

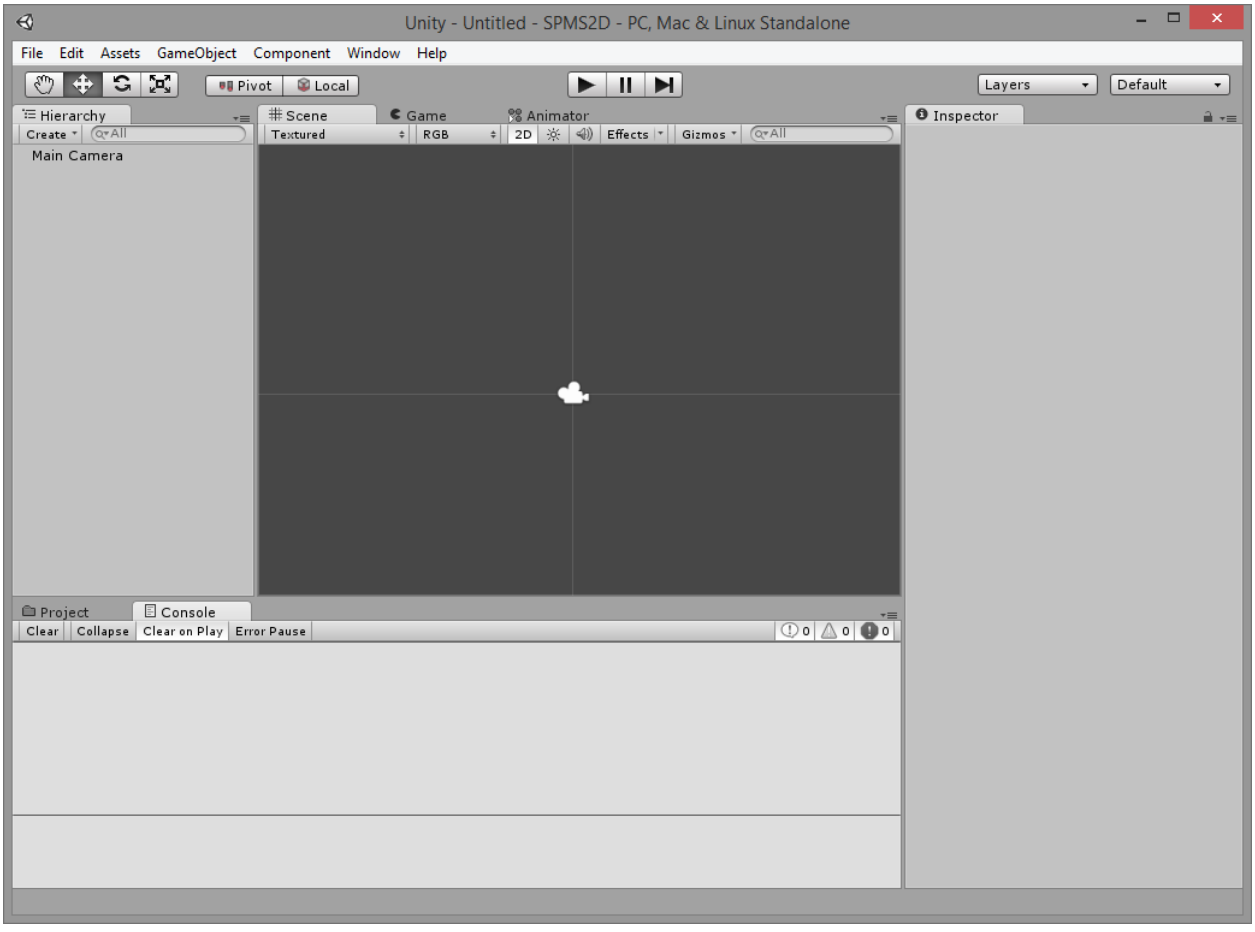


By setting this to 2D, your project will automatically be configured for 2D game development. Let's create a project called SPMS2D and toggle 2D.

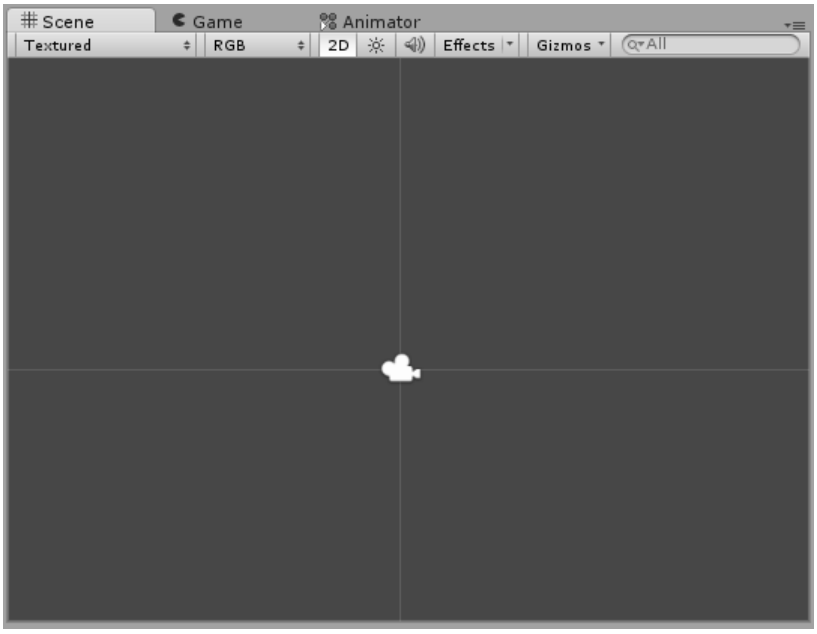
Once you create your project, you will go into the editor and see the "Welcome To Unity" screen. I highly suggest going through some of the videos to learn more about how Unity works and get a sense for the workflow. Most of it is geared toward 3D, but it still applies to the stuff we will go over in this book. I also suggest checking out the forums since there is a lot of really good information on there that can help you out when you get stuck as you are getting started learning Unity.

How to Use the IDE

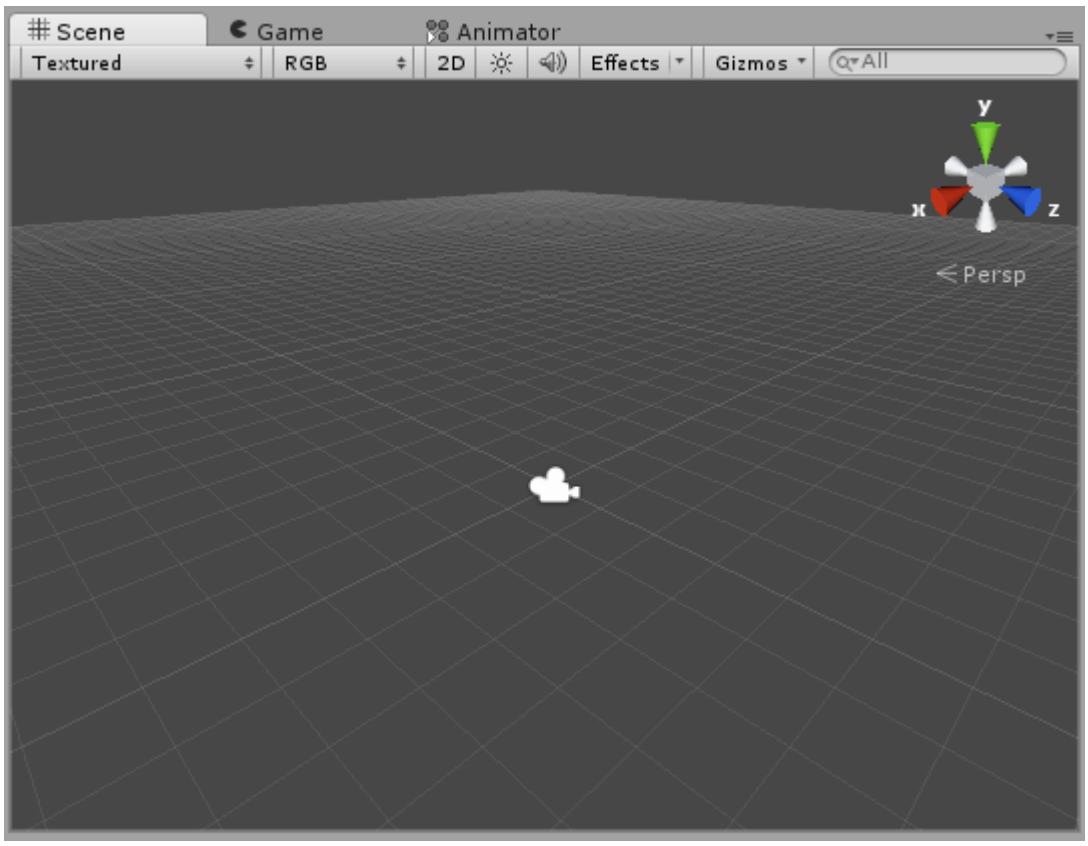
At first, Unity may be a little intimidating, but I personally find that the new 2D mode actually simplifies things greatly. Here is a high-level overview of the IDE. When you first open up a project in Unity you will see the following window.



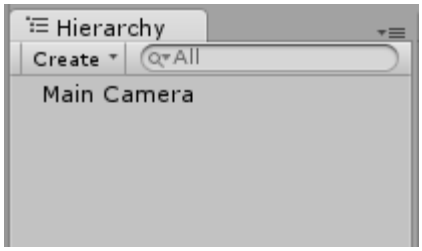
Let's go over each section of the IDE's window by window. First, we'll start with the main area.



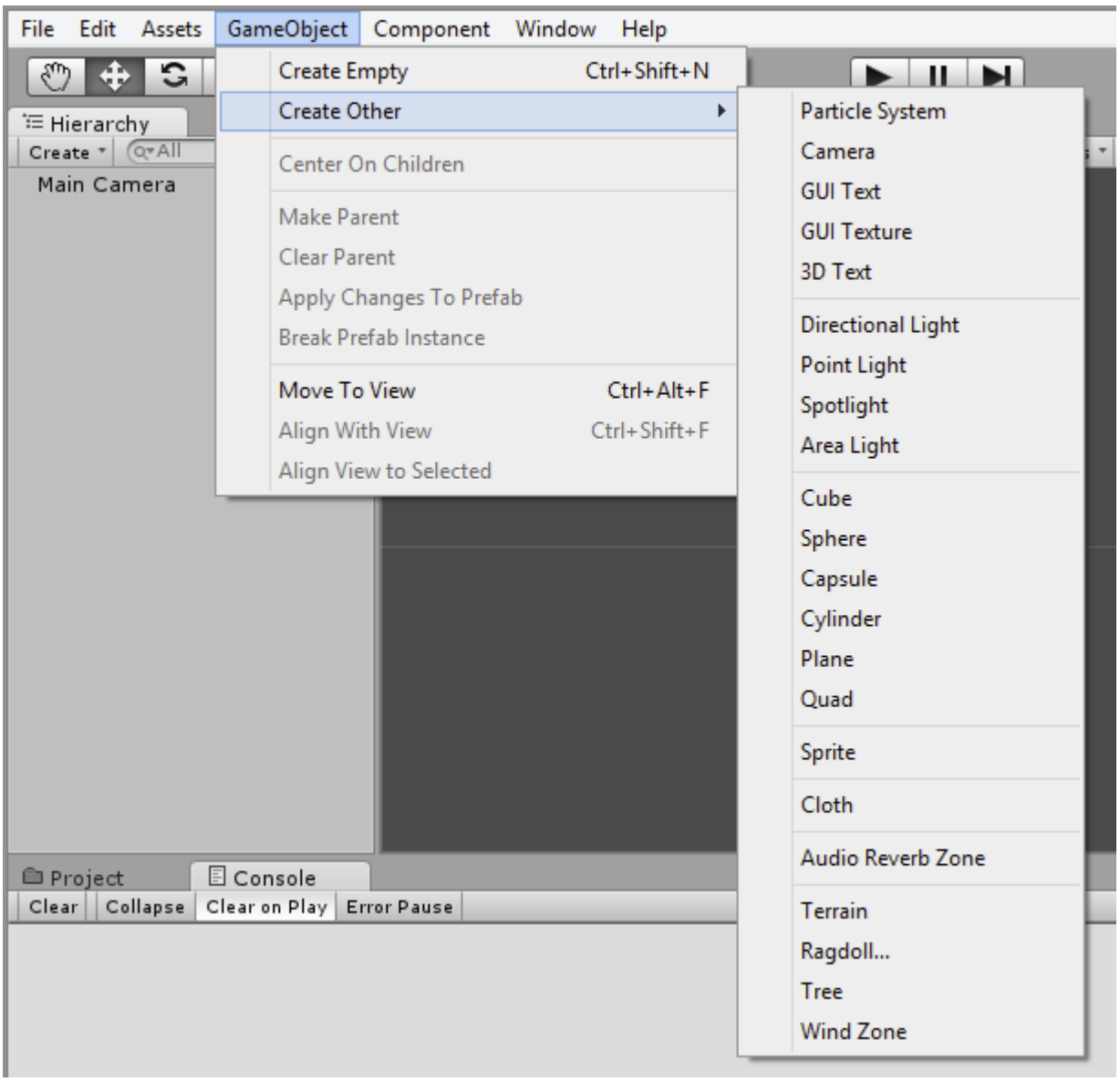
This is where you will lay out objects, preview and test your game, and also work on animations and other visual-based activities. As you can see from the tabs, the Scene and Game tabs are already open. Below the tabs, you will see a few quick-access menus. The most important is the 2D toggle, which is already activated. By unchecking it, you will go back into Unity's native 3D view.



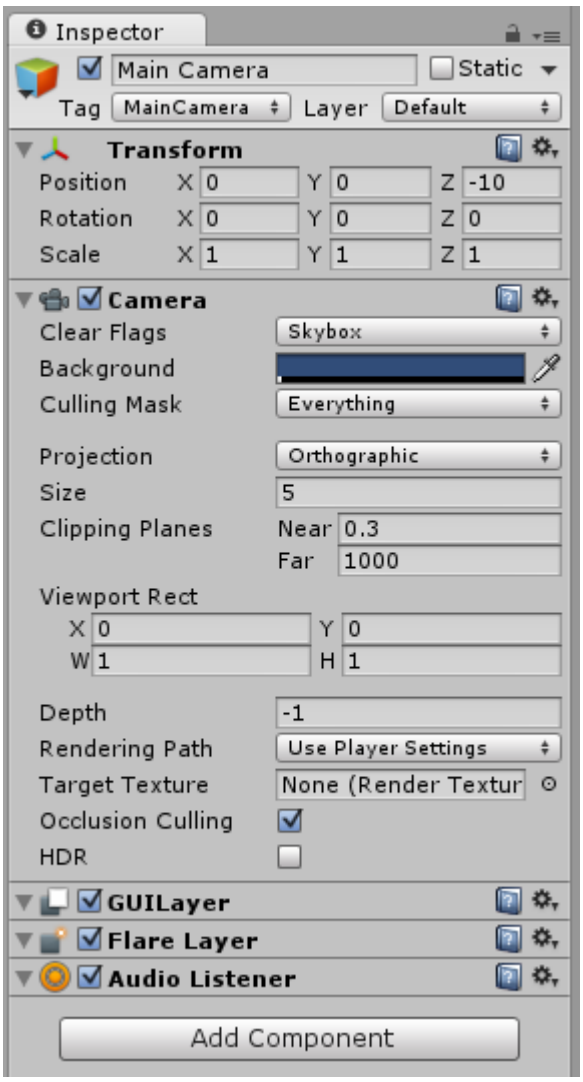
I'm not going to go into the 3D navigation tools since we will be focusing on 2D instead so let's look at the Hierarchy panel.



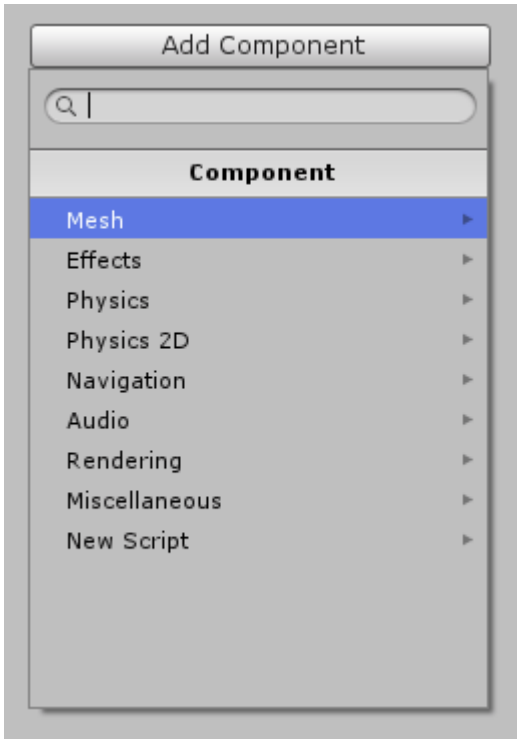
As you add stuff to your scene, you will be able to see and access them from here. For now there is a single object, which is the camera. Also, you have access to a Create shortcut menu, which we will be using a little later on. The same options can be accessed in the IDE's top menus as well.



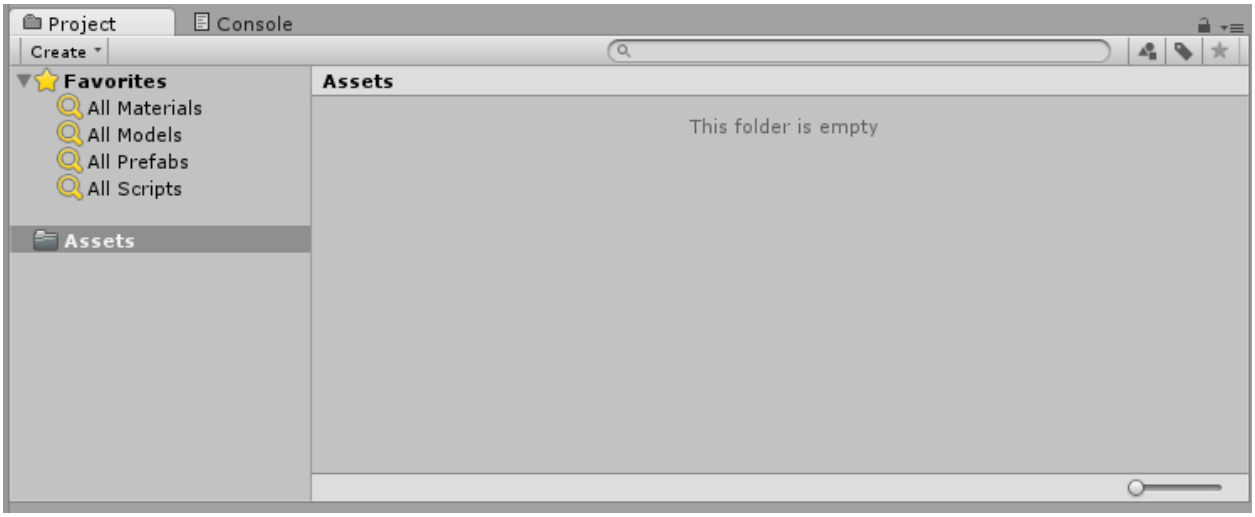
While you are looking at the Hierarchy tab, select Camera so we can discuss the Inspector panel next.



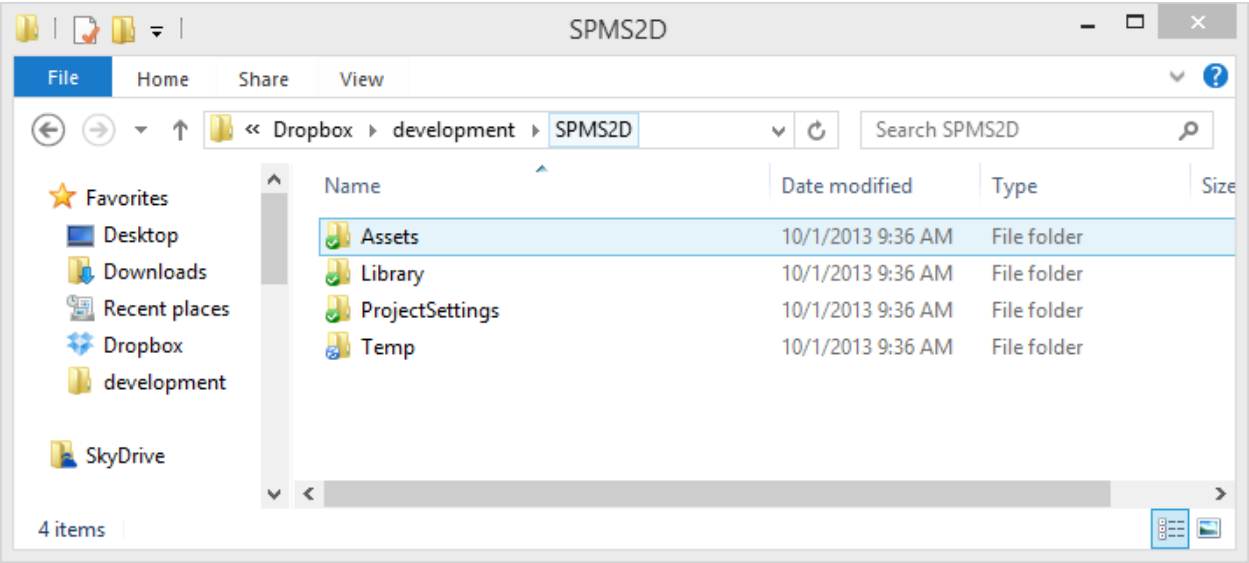
Here you can see all of the properties that make up the camera for our game. Unity does a great job of visualizing all the parts that can be configured on each object in your Scene. The Inspector panel not only allows you to modify values, even on the fly while the game is running, but it also lets you add additional functionality onto any GameObject via the Add Component button.



We will get into some of these components a little later on, especially scripts, which will make up a huge part of your coding experience. For now, it's important to note that a GameObject is anything in the game itself and may be the camera, the player, or even more complex objects like levels, UI or even utilities we build to help us visualize elements in the game. Next up is the Project tab.



Think of this as a view into the project folder itself. To help you better understand it, go to where you created your project on your system and open it up.



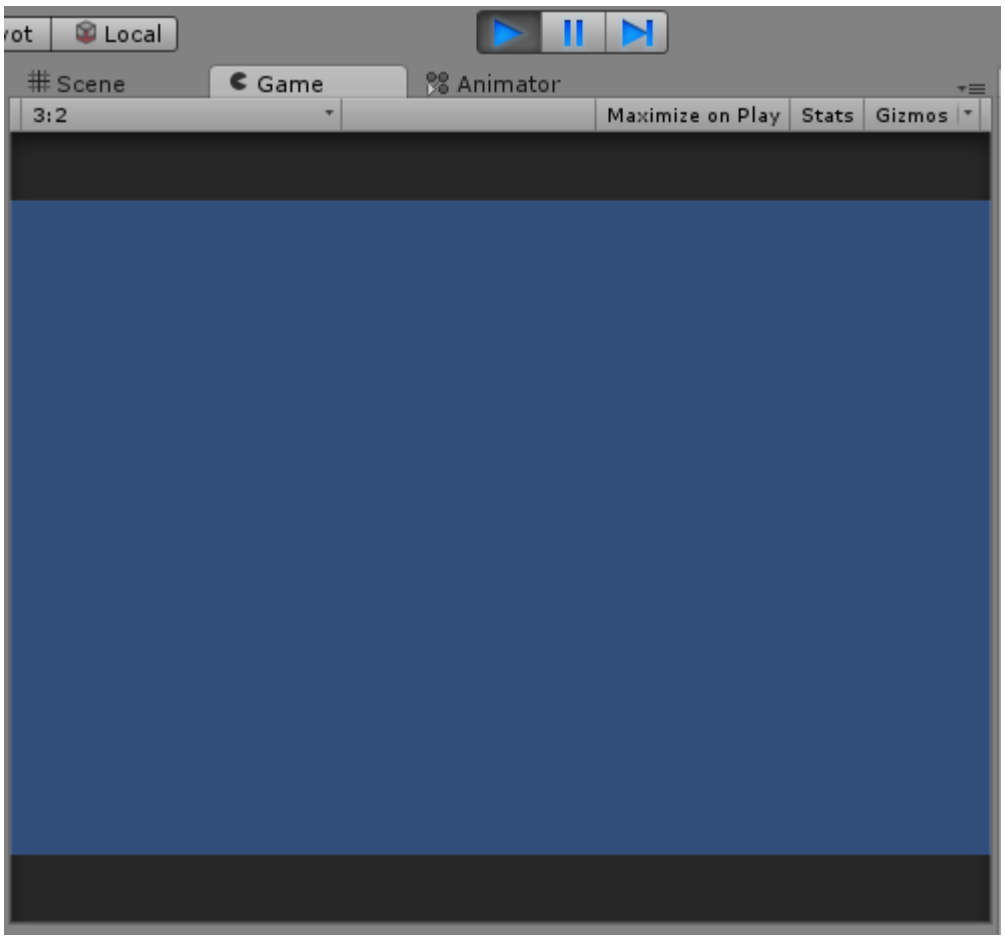
As you can see, in addition to all of the additional files that make up the Unity game project, you will see there is an Assets folder. This becomes your default location for everything you put in your game. Here we will store artwork, sounds, scripts, and prefabs, which are reusable pre-configured GameObjects.

The final two things I want to cover should be self-explanatory but are always useful to review. In the upper-left corner you have a set of controls to help you navigate the Scene window.

To be honest, most of these tools don't do much in 2D mode. They are designed for navigating the 3D space. The one you will use the most is the Hand tool, to simply drag the screen around on the x- and y-axis.

The other set of controls handles playback and allows you to test the game.

Simply hit Play to start the testing in the Game tab.

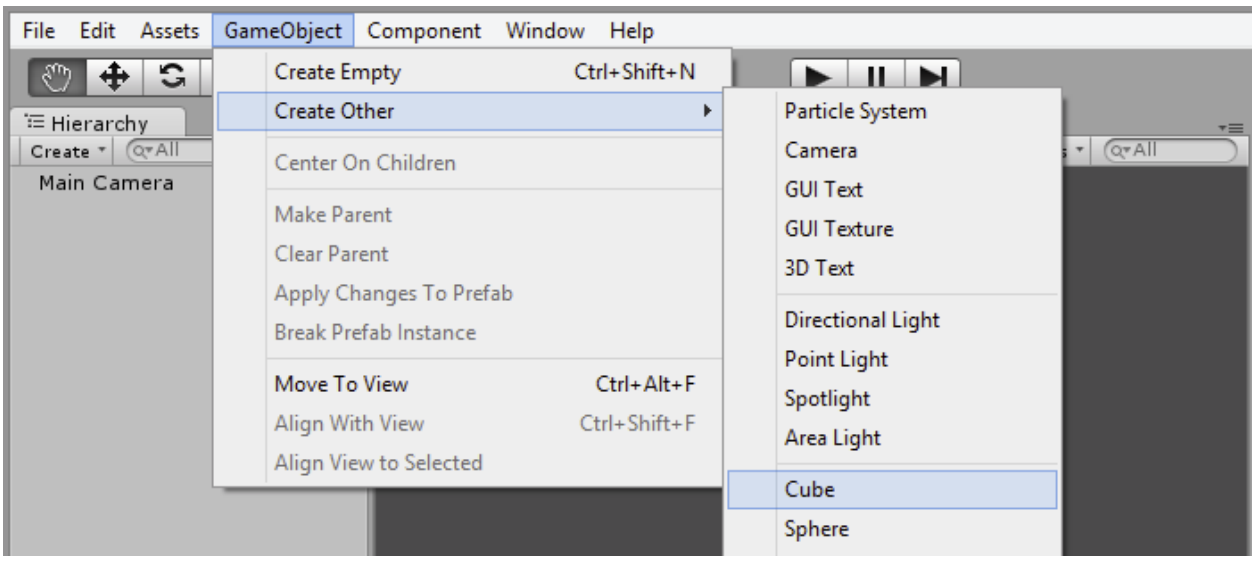


You will also notice the Pause and Step-Forward buttons. These are incredibly helpful in allowing you to move through the game frame by frame to see what is going on. You can also go back to the Scene tab while playing the game and modify values of GameObjects at runtime. It's important to note that any changes you make in the Inspector panel while running the game don't get saved; it simply allows you to try things on the fly without having to stop testing, make a change, and recompile.

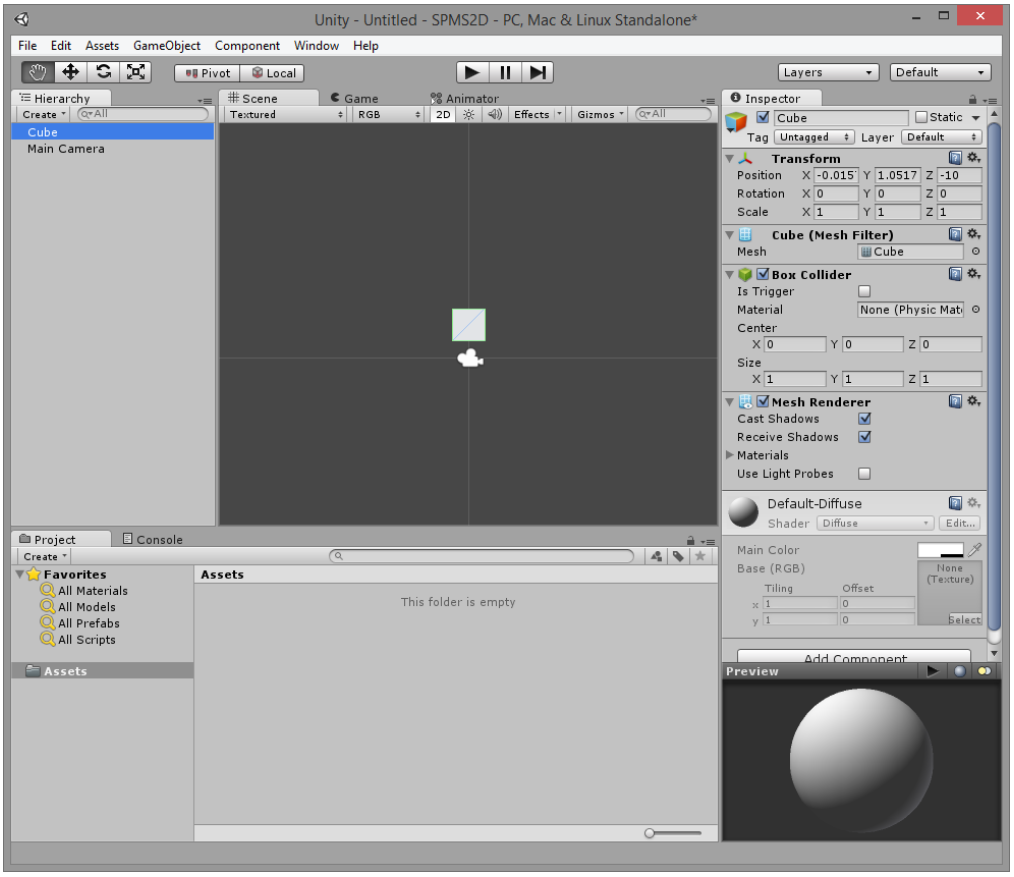
GameObjects

Before we get into the coding and specific 2D tools, I wanted to talk about GameObjects, which are the building blocks of a Unity game. As you begin to flesh things out, you will start by creating these GameObjects in the Scene. Some of them will represent your Player, bad guys, and level while others will make up utilities and Gizmos, which we will learn about later on, to help you manage other GameObjects in the game.

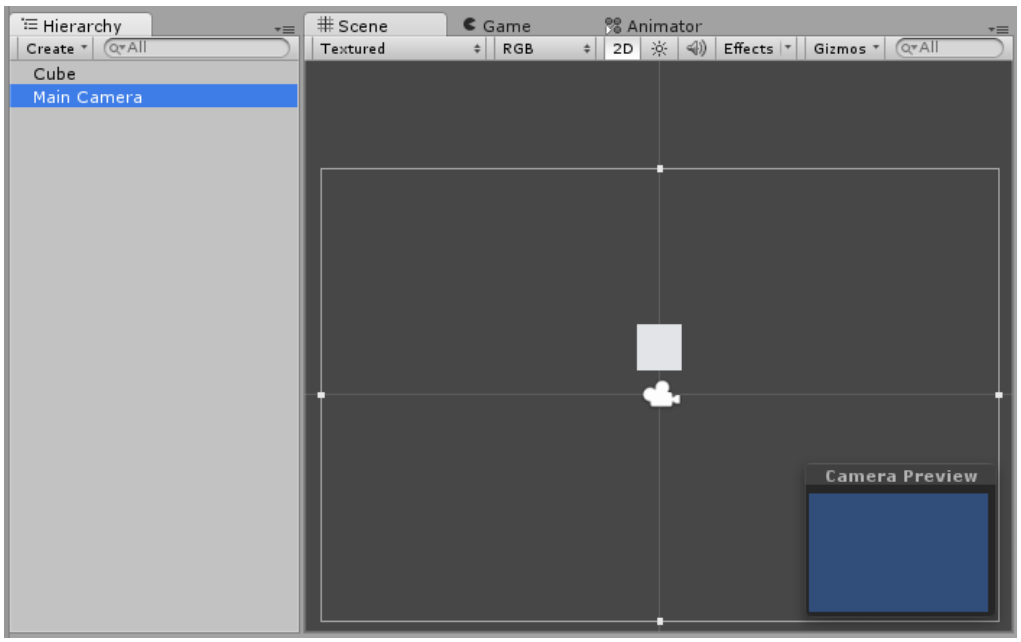
To get you used to working with GameObjects, let's just create a simple cube and position it in our Scene. Go to the GameObject menu at the top, or the Create menu from the Hierarchy panel, and select Cube.



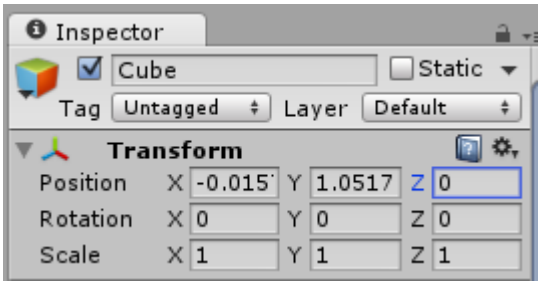
Now, if you check your Hierarchy panel, you will see the cube and the camera. Select the cube to bring it up in the Inspector panel.



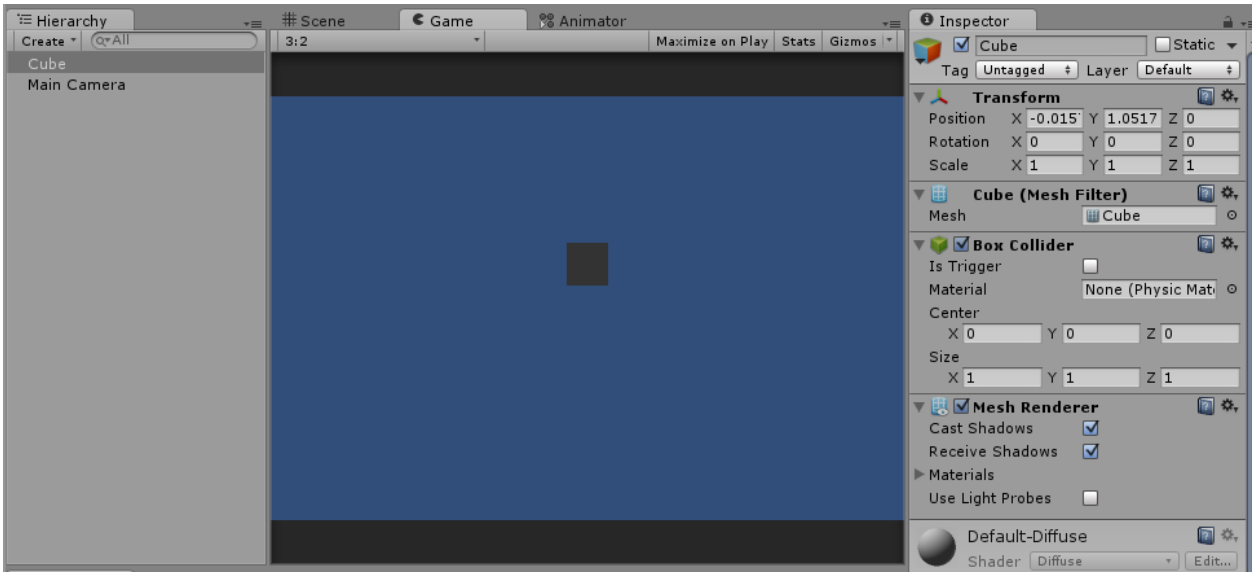
If you run your game now, you will not see the cube. Likewise, you can click on the camera to preview the Scene without even hitting Run.



We can fix this by adjusting the cube's z-axis in the Inspector window to 0.



Now, if you run the game, you will see the cube.



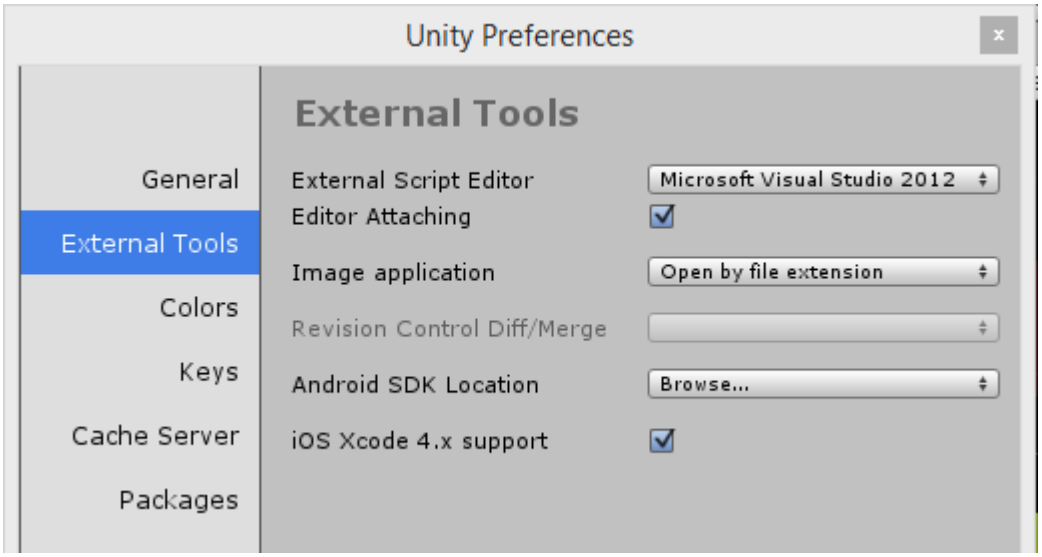
Notice that we still have the Inspector panel open with the cube. You can continue to play with the cube's properties while the game runs. If you roll over any number property, you will see the cursor change into a two-sided arrow, and you can use that to click and drag the value in the field up and down. Give it a try with the x and y properties of the cube. Since we are in 2D mode, modifying the z-axis will not do anything eventful. When you stop running the game, all of the values will reset themselves.

Before we move on to making a game, let's learn about scripting in Unity with C#.

Introduction to C# in Unity

I am a big fan of C#, and while Unity Script is useful, eventually you will need a little more flexibility in your code, so I decided to focus on using C# for our game. Because of this, I wanted to do a quick primer on C#, scripting in Unity, and some of the more common APIs you will be using when we build our game.

As we dig deeper into the code, you will need to pick an external editor since Unity doesn't have one built in. By default, Unity ships with MonoBuilder, but you can just as easily switch it out for something a little more robust, such as Visual Studio is you are doing your development on Windows. Simply go into the Edit > Preferences menu and change the path to the external editor.



Once you have picked an editor you like, you are ready to start coding. In this book, I will be showing the code from Visual Studio, but MonoDeveloper will work exactly the same on Windows and Mac.

Data Structures

C# is a strongly typed language and is very similar to Java or ActionScript 3. If you have some background in JavaScript, it should feel familiar as well. Let's take a look at the basics of the language. To start with, we will look at variables and lists, which are the building blocks of any language. To make a variable, simply declare it, like so:

```
var name = "Jesse Freeman";
```

While C# is a typed language, it supports type inference, meaning that when you declare a variable you don't always have to define the type. There are a few primitive types you should know about:

```
var string name = "Jesse Freeman";
var int age = 34;
var bool alive = true;
```

When it comes to numbers, you should also understand more complex types, such as `Float`:

```
var float speed = 4f;
```

Notice that we use the `f` suffix for our float. Floats are numbers that contain decimals and are used for more granular positioning within the game world, as well as in the built-in physics engine. `Ints` are useful for whole numbers where you don't need to perform calculations that would generate fractions, such as a player's health. You should always be aware when using `Int`, `Float`, or other number types because you may be forced to cast it to a specific type in order to perform the calculation and you may incur a performance penalty. Here is an example:

```
var float example = (float)age * speed;
```

You can learn more about the different types in the C# reference docs at http://bit.ly/1bwLAKW. The next set of building blocks is `Array` and `List`. An `Array` is collection of multiple variables, usually of the same type.

```
var String[] stringArray;
```

With arrays, you can store additional items by calling `Array.Add`:

```
stringArray.Add("Jesse");
stringArray.Add("Freeman");
```

You can access properties in an `Array` by their index, which is a unique number for each location in the `Array`. In C#, arrays start at 0. Here is how I would create a string with my full name from the above `Array`:

```
var name = stringArray[0]+" "+stringArray[1];
```

Unity also supports `Lists` as part of the C# library. In order to use this, you will have to import its package at the top of your script which I will show you how to do later on. For now, here is an example of a list:

```
var List<int> listOfInts;
```

This is what we would call a generic List. The term generic refers to a dynamic type we can assign at runtime. Generics are a core part of the language, and something you should get familiar with as you gain more experience coding in C#. In this example, we can create a list with a type of `Int`. Now this generic list can contain whole numbers. For performance, you will want to use lists over arrays when you don't know the exact size of the data and expect to be adding or removing values at runtime. An `Array` by contrast has a set number of items you can have in it and shouldn't attempt to modify the length at runtime.

The last data object I want to talk about is a vector. Unity makes use of a `Vector3`, but now with the addition of the 2D workflow, they now rely on `Vector2D` a lot more. Here are examples of both:

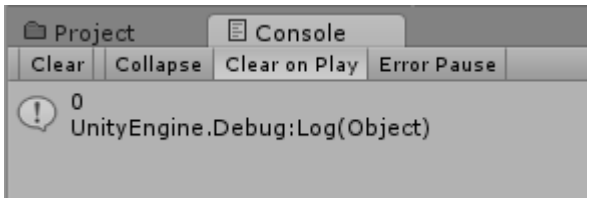
```
var v3 = new Vector3(0,0,0);
```

```
var v2 = new Vector2(0,0);
```

Vectors allow you to store 3D or, in this case, 2D points in space so you can access the value of `x`, `y`, and `z` from the vector. Here is an example:

```
Debug.Log(v3.x); // will output 0
```

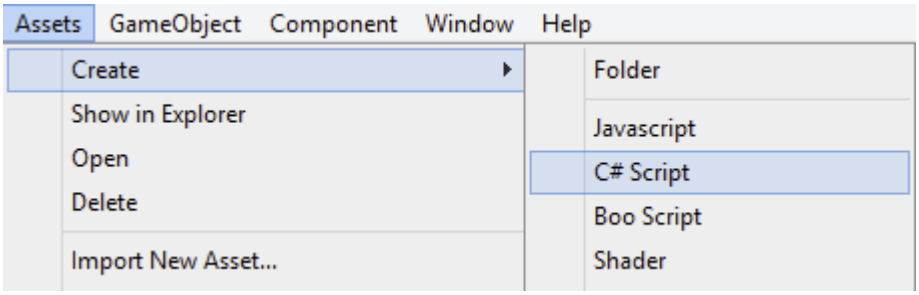
One quick note is that you see I am calling `Debug.Log`. This will output the value to the console window.



This is similar to most other languages, such as JavaScript's `console.log` and ActionScript's `trace`. It's also the first step in debugging your apps if you ever need to see the value of an object or property.

Classes

C# takes full advantage of classes, interfaces, and a host of other ways of packaging up code for reusability. Unity itself is built on a very easy-to-grasp composition system, which it favors over inheritance. Let's look at how to make a simple class. Go to the Create menu and select a new script.



As you begin to create a new script, you will notice you can select from the three built-in languages. Select C# and call the script `HelloWorld`.

Unity will stub out the code you need for your class for you. Here is what it will look like:

```
using UnityEngine;
using System.Collections;

public class HelloWorld : MonoBehaviour {

	// Use this for initialization
	void Start () {

	}

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

```
        }
    }
```

As you can see, you won't need to memorize how to create a class from scratch, so I will simply focus on two main parts of the script: the import block and the methods. By default, each new class extends from `MonoBehavior`. There are numerous methods you can take advantage of, but we'll start with the first two: `Start` and `Update`.

Let's go back to our previous example of a generic list. While this isn't the traditional way of doing a Hello World example, I'll be able to show off in a little more detail what it is like to import external libraries, create properties on a class, and output that to the console window. To start, look at the top of the class at the import statement and add the following:

```
using System.Collections.Generic;
```

Now, just before the `Start` method, we are going to add a property. Properties are variables that are scoped to the instance of the class itself. That is a fancy way of saying that anything inside a block of code will have access to its contents. Depending on how we declare the property, other classes will have access to it as well. C# supports `private` and `public` variable denotations. If you don't declare one, it will automatically default to `private`. Add the following property above our `Start` method:

```
public List<string> displayText;
```

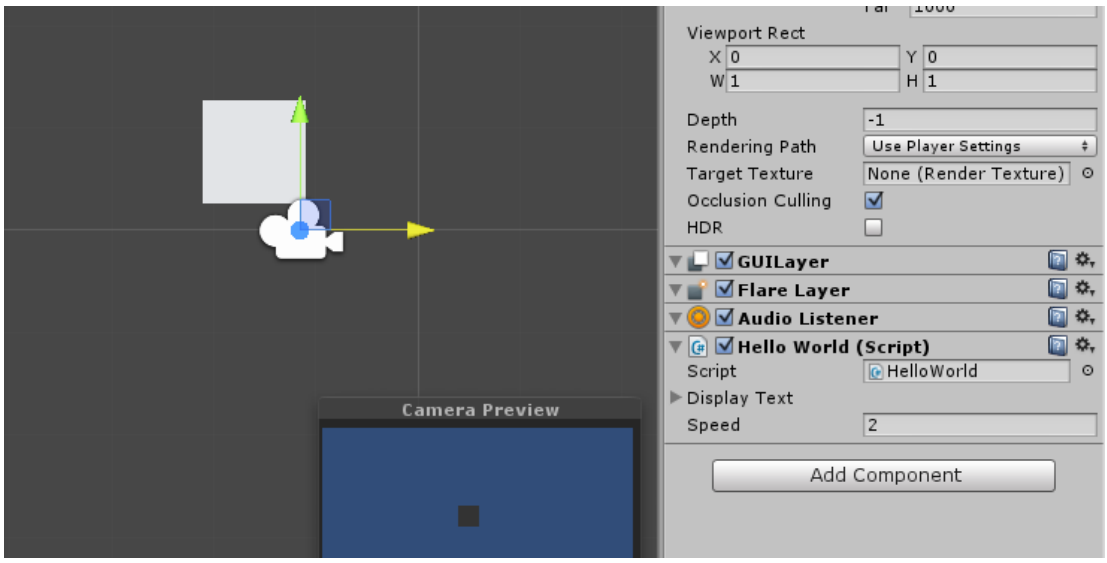
Now, in our `Start` method, add the following:

```
displayText.Add("Hello");
displayText.Add("World");
```

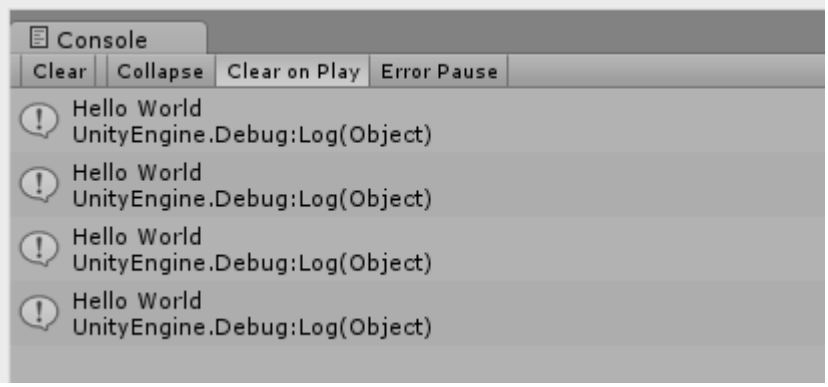
Now we need a way to display this text. Add the following to the `Update` method:

```
Debug.Log(displayText[0] + " " + displayText[1]);
```

Here you can see we are using the `Debug.Log` method again, and we are accessing the first and second values of our list. It's important to note that arrays and lists are 0 based in C# just like Java, AS3, and JS. Now we have a script that will output `Hello World` to the console tab on each frame, but we don't have a place to put it. Go back to the Scene and select our camera. From here, scroll down to the bottom of the Inspector panel and select Add Component. Now select our `HelloWorld` script and it will attach itself to the camera.



Now, if you run the game and look at the Console tab, you will see `Hello World` outputted on each frame.



While this is a simple example, let's do something a bit more interesting. Go back into your script and let's have the camera scroll to the right. The camera is a GameObject just like any other primitive you add to the Scene via the Create menu. All of these GameObjects share some common properties, with position, size, and scale being just a few of them. Also, all of these GameObjects share the same API we are taking advantage of right now, which are the `Start` and `Update` methods. Let's look at how we can programmatically move the camera. To get started, let's add a new property called:

```
public float speed = 2f;
```

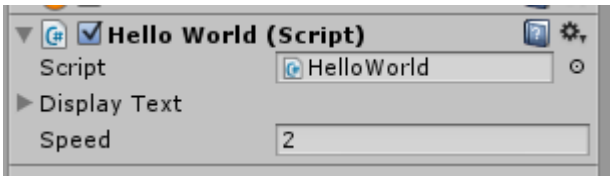
Next we'll want to replace our Hello World `Debug.Log` call with the following:

```
transform.Translate(new Vector3(speed, 0, 0) * Time.deltaTime);
```

As you can see, we have taken the `transform` position property and are modifying it with a new `Vector3` that contains our `speed` value and is multiplied by the `Time.deltaTime`. If you simply increased the `x` position by `speed`, you are not taking into account any slowdowns in the game itself. By using the delta between each frame, you are able to keep your movement consistent from frame to

frame, regardless of dips in the frame rate. This isn't a magic formula; all it means is that your GameObject will move at the desired distance over time, so if the frame rate drops, it will look jerky but won't slow down or speed up as the FPS naturally fluctuates based on other things going on in the game.

If you run the game, you will see the camera move. It will appear like the box we created earlier is simply scrolling off the left side of the screen. Stop the game and take a look at the camera's Inspector panel. If you scroll down to the script area, you will see we now have a new input field called Speed we can alter.



This is an important concept in Unity. Basically, any public property on a script can be edited from the IDE in the Inspector window. So, while it's important to create default values for your properties, just know that you can alter those values on an instance-by-instance basis, and even temporarily at runtime when you are debugging. This is incredibly powerful and something we will be taking advantage of later on. Not only does this work with simple properties, such as `Strings`, `Numbers`, and `Booleans`, but it also works with collections, such as `Arrays` and `Lists`.

If you remember back to our first example, we had a list called `displayText`, which we also made public. You should see it in the Inspector panel as well, but the value is hidden. Simply click on the arrow next to its property name. You can even add new items to it by changing the `Size` value.

So, at this point, you should have a basic concept of how scripting works in Unity. Everything we covered here will be re-introduced in the following chapter. There is just one last thing I want to cover, which is how to think through scripts for GameObjects.

Composition over Inheritance

If you come from a traditional computer science background or have worked with other object-oriented programming languages before, you may be familiar with the argument on composition over inheritance. One of the cornerstones of OOP languages is the concept of polymorphism. While inheritance plays a large role in game development, Unity strives for the use of composition as much as possible. I consider scripting in Unity to follow the decorator and component design pattern. Each script

adds additional functionality to your GameObject, but they are mostly self-contained. While some scripts rely on others to function, as we will see in the game we end up building, we really strive to create small, reusable blocks of code that can be applied to multiple GameObjects. When put together, each of these smaller scripts build up to a greater set of logic that builds up the functionality of each GameObject. While inheritance is possible, I strongly recommend thinking through composition as a means to not only create more modular, self-contained code but to also separate and encapsulate independent game logic. This kind of thinking is critical when it comes to grasping the true power of creating scripts in Unity.

By now I am sure you are itching to get into some code, so let's start our project. Before you begin, you can delete our cube and delete the script from the Assets folder. Or you can simply create a new project from scratch.

Building a 2D Game in Unity

So, now that we have covered all the basics of how to use the IDE and what it is like writing scripts in Unity, it's finally time to jump on in and build a simple game. The name of this game is Super Paper Monster Smasher and the goal of is of the game is to keep the monster, which you control, alive for as long as possible from the endless onslaught of good guy knights. Originally this was a game I created during a 48 hour gam jam called Ludum Dare but since then I have turned it into a demo that is perfect for learning the core skills needed to make a game:

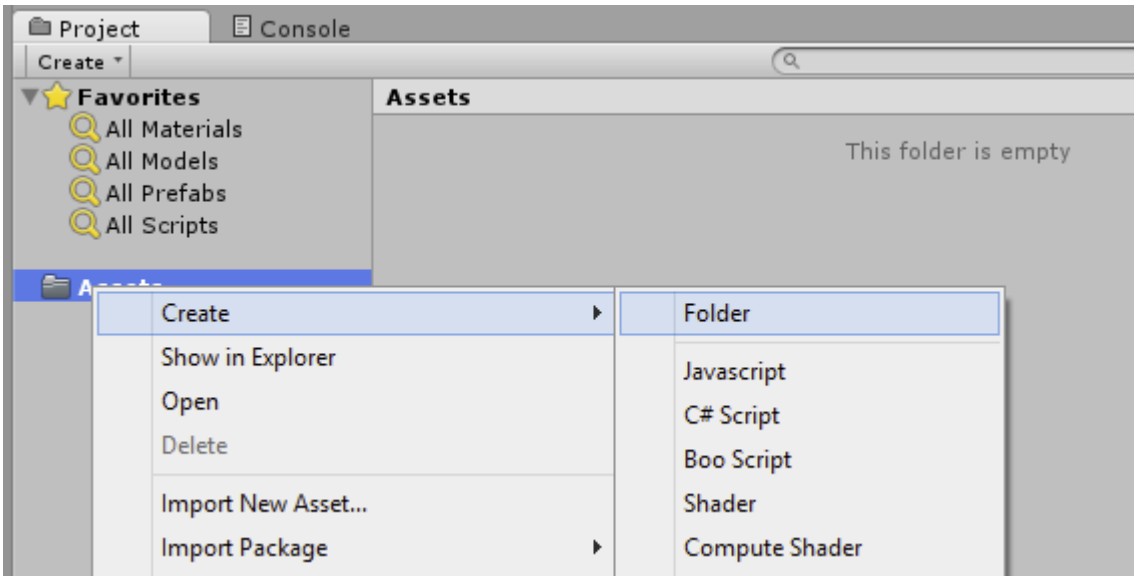
- Managing game assets
- The basics of animation
- Building a level
- Moving the Player
- Collision detection
- Spawning enemies
- Changing from scene to scene

As you can see, we have a lot to cover, so let's jump right in.

Getting Started

You can continue with the project we set up earlier, or start a new one from scratch. Either way you should only have the `Camera` in your project, and have 2D mode toggled.

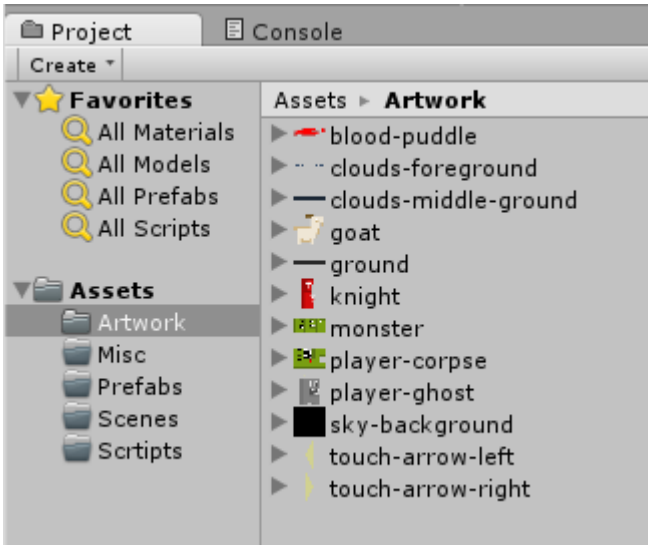
Next we need to import all of the game artwork into the Assets folder. Before we do this, let's create a new folder called `Artwork`. You can easily do this by right clicking on the `Assets` folder itself and selecting Create > Folder.



While you are at it, make the following folder structure:

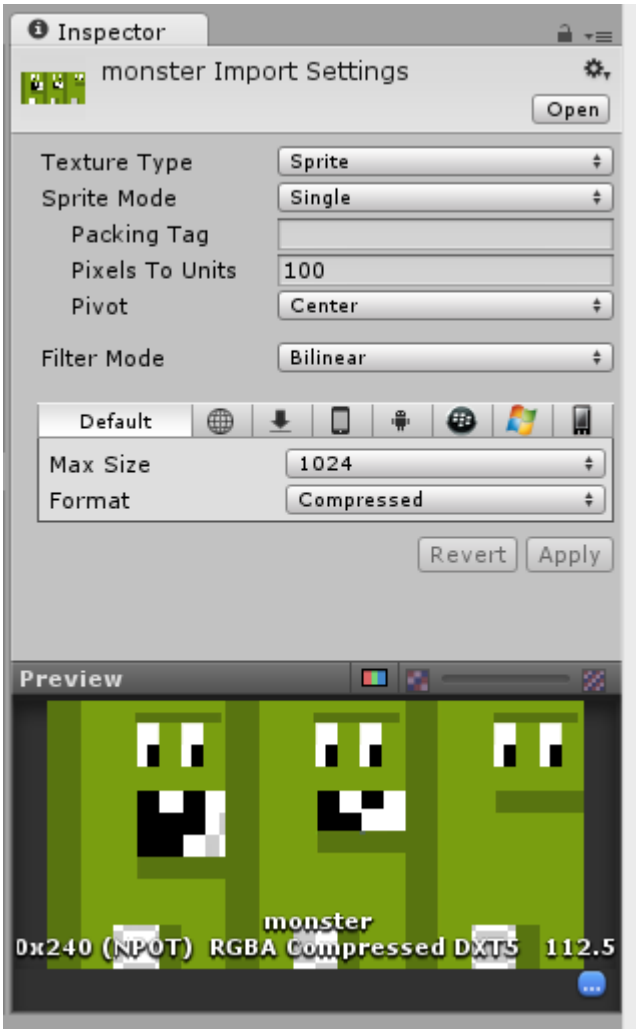
- `Artwork` – Where all of our artwork goes.
- `Misc` – For extra files we create that don't necessarily fall into one of our three main categories.
- `Prefabs` – This is where we will put completed prefabs, which I will talk about later on.
- `Scenes` – This is where you will save game Scenes.
- `Scripts` – This is where we will put all of our scripts.

At this point we can drag the game artwork into our Artwork folder and open it up to take a look.

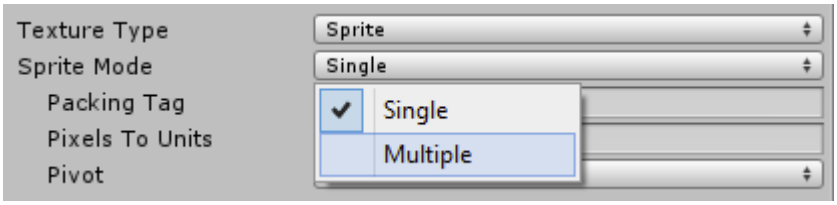


Here you will see we now have everything we need to build our game with. If you select any of these images, you will notice in the Inspector panel that it has already automatically been converted into a sprite texture for you.

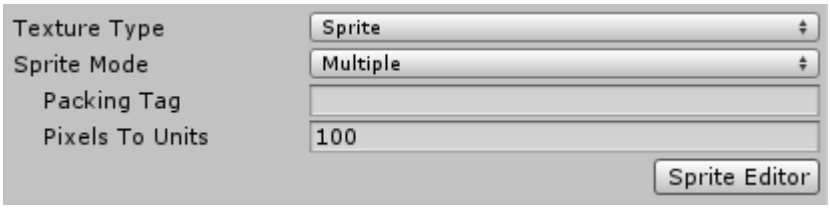
While this is incredibly convenient for a single sprite, you may want to import your artwork as a Sprite Sheet. As you can see from our artwork, the Monster is already set up like this for us. Click on it and look at the Inspector panel.



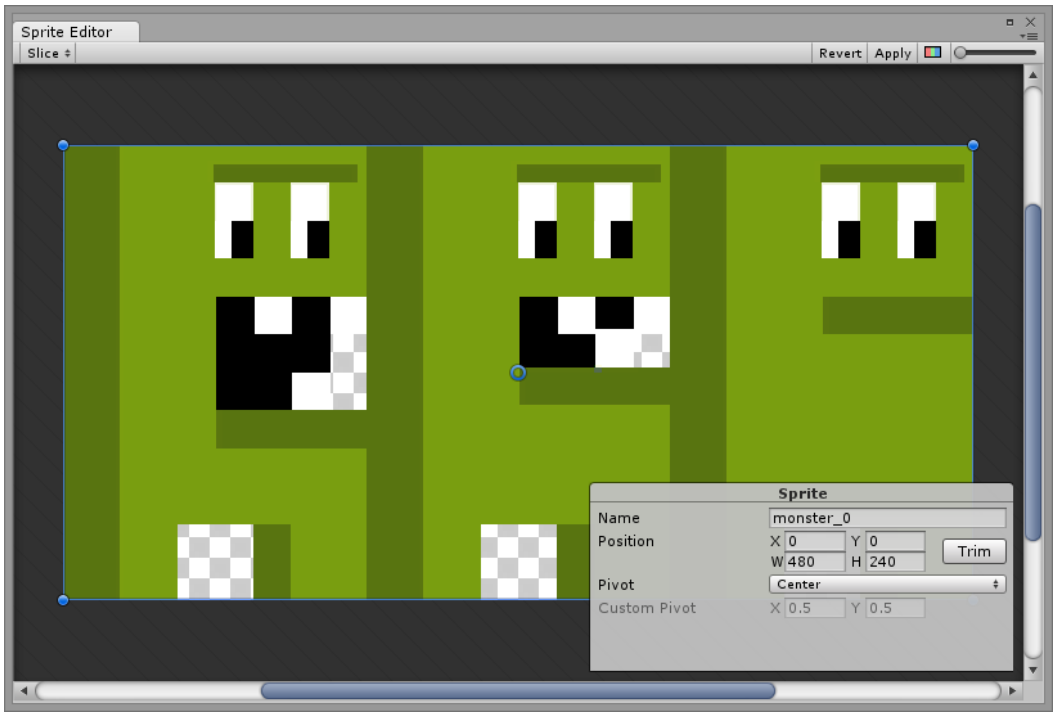
You will notice there is a `Sprite Mode` dropdown under the `Texture Type`. Click on that and select Multiple.



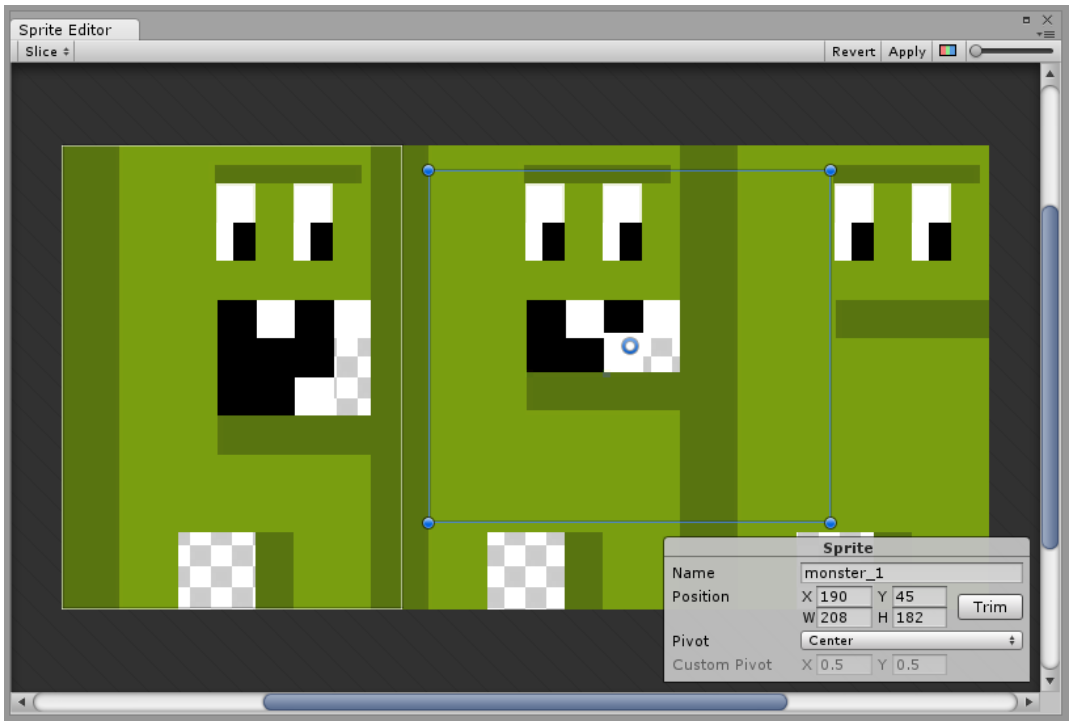
From here you now have access to the build in Sprite editor.



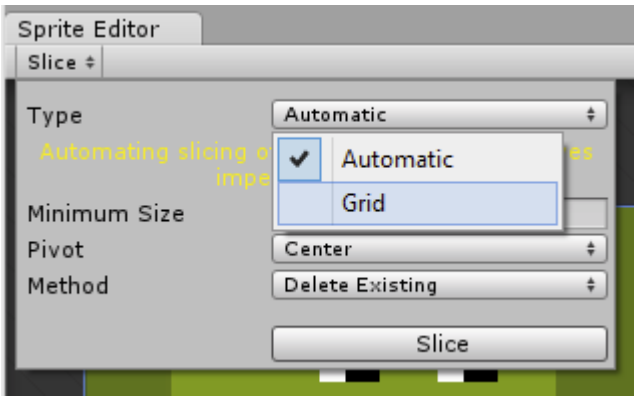
Once you have done this, you can begin cutting up your sprite sheet manually or having Unity do it automatically. Click on the editor and you should see the following tool.



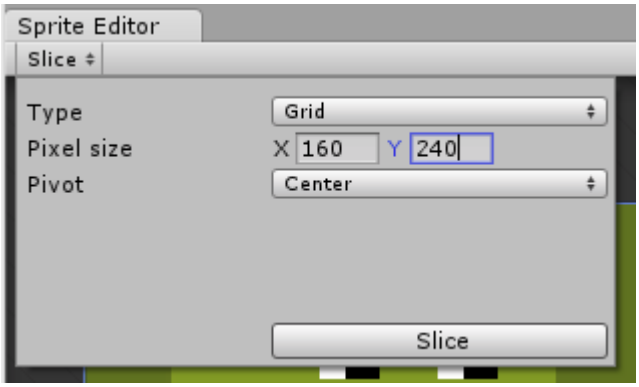
By default, the entire image is set up to be the sprite. You can manually change this by drawing a blue box around the image's border, and you can also draw additional boxes by hand to define more sprites.



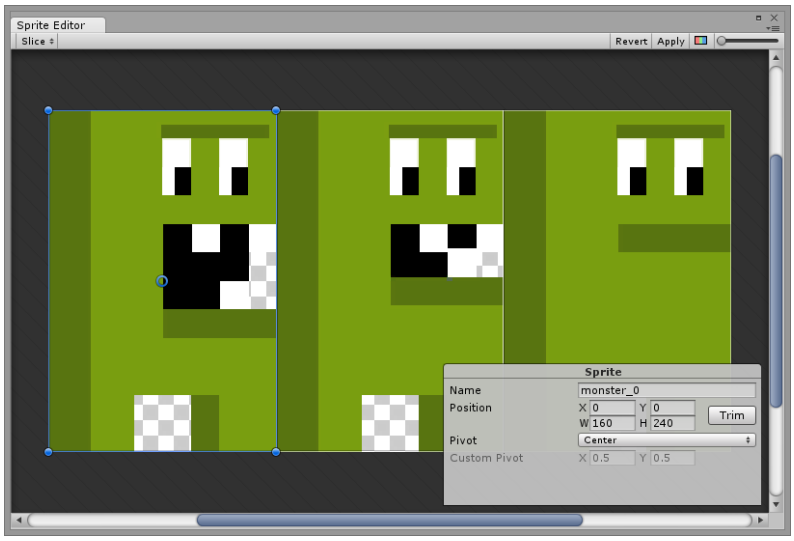
This is great for more complex sprite sheets, but this image can be automatically cut up for us. In the upper-left hand corner, select the Slice tab and change its Type to Grid.



Once you do that, you will be presented with some options to define the dimensions of your sprites.



The `X` and `Y` here represent the dimensions of each sprite. Change the `X` to `160` and `Y` to `240` and hit Slice. The last thing you need to do before closing this menu is hit Apply in the upper-right corner. Once you do that, you will now be able to select each sprite individually and see its blue box outline confirming the sprite was correctly cut up.



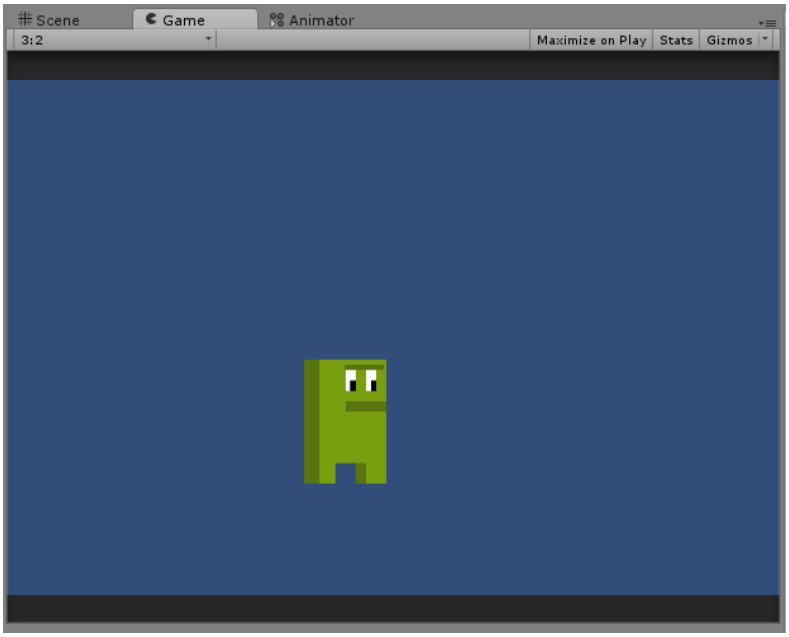
Now, if you toggle the check next to the `monster` sprite in your Artwork folder, you will see each sprite as its own file.

Now we are ready to create an animation for our Player.

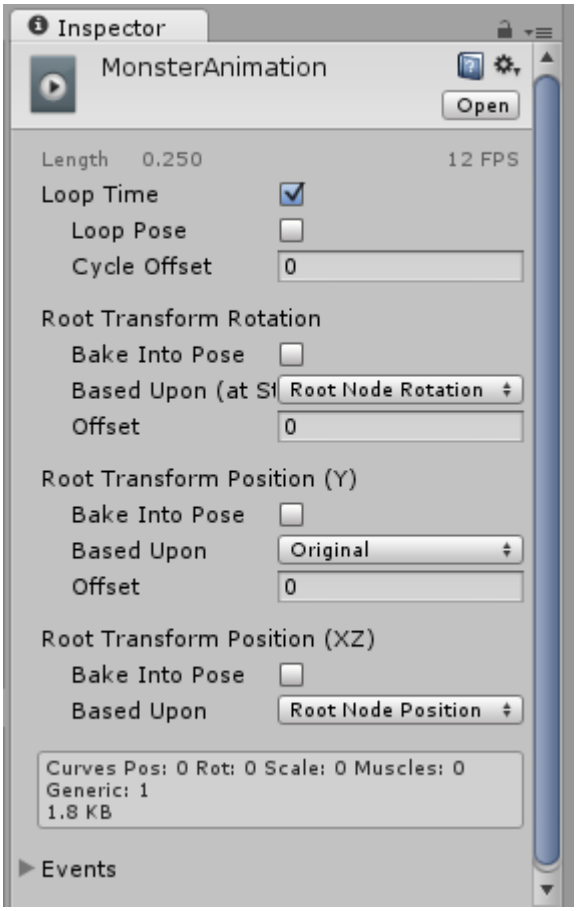
Setting Up Animations

Creating animations in Unity is very easy; it's basically done via drag and drop. To get started, we will need to select the three Monster sprites we created in the previous section and drag them onto the Scene. As soon as you do this, you will be presented with the Create New Animation window asking you to name it and find a place to save it.

Simply call it `MonsterAnimation` and save it to the `Artwork` folder. If your game has a lot of animations, you may want to have a dedicated folder for organizing them better. Once you hit Save, you'll have a GameObject with your animation in the scene. Hit the Play button to see it in action.

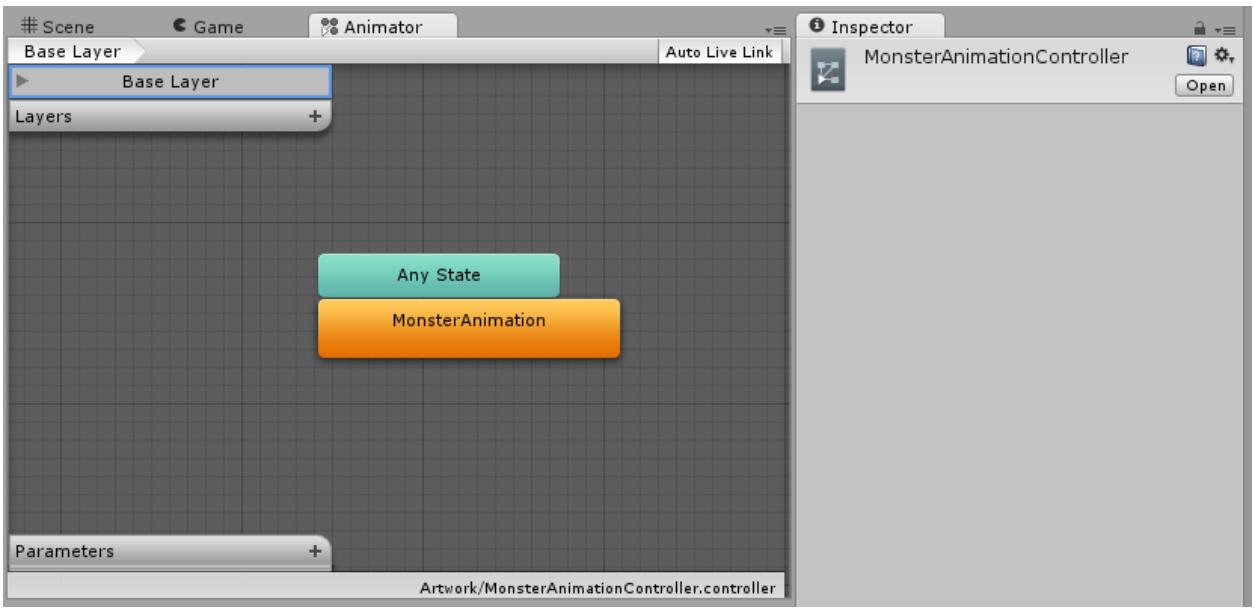


There is not much to look at right now, but this is the basic process for creating a 2D animation in Unity. Let's talk a little more about the two items we just created. We'll start with the `MonsterAnimation`. Click on it and look at the Inspector panel.

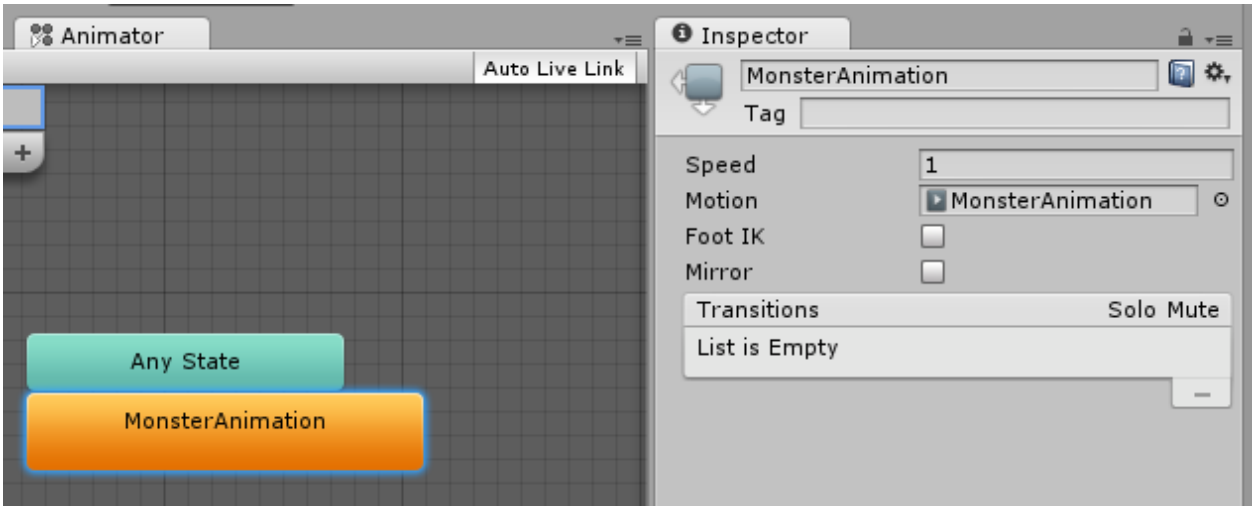


Here you will see some basic information on how the animation will run. A lot of these options actually apply to 3D animations. You should keep in mind that Unity's new 2D workflow was designed to work within the same 3D system Unity developers had been using for years. Now let's select the controller for the animation and, from the Inspector panel, select the Open button. You can find it in your asset folder called `monster_0`, which was automatically generated for you.

Rename it to `MonsterAnimationController` and hit the Open button in the inspector.

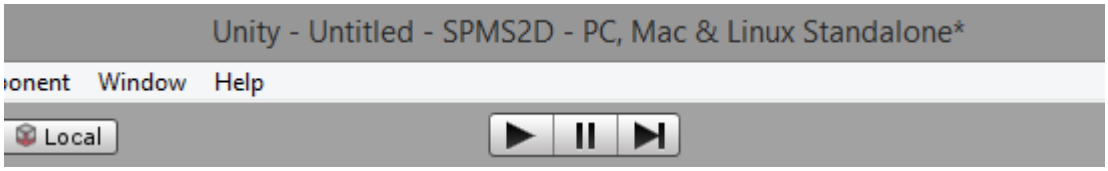


Here you will be presented with the Animator tab. This allows you to chain animations together and create more complicated animation sets. We will only be setting up a single animation for this game, but it's important to note that you can open up the `MonsterAnimation` in this view and modify more of its properties.

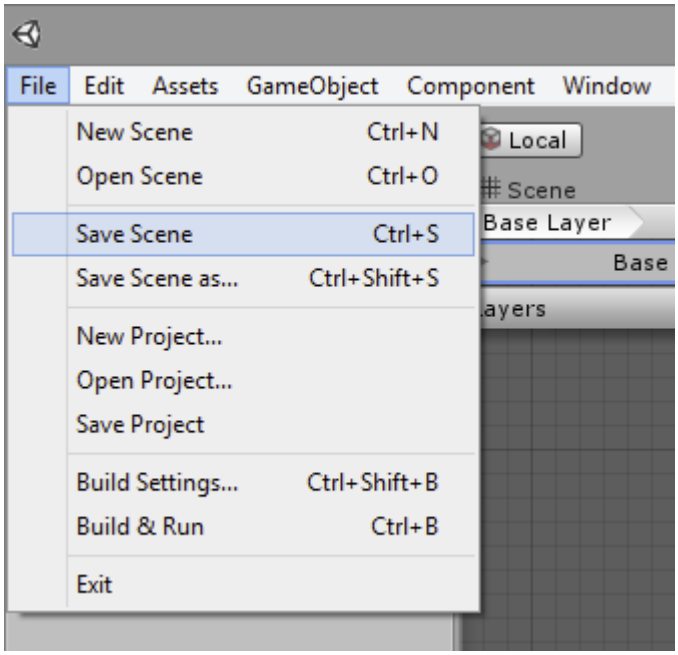


Here you will notice that our `Speed` is set to `1`. You can play with this value to alter the playback of the Monster opening and closing his mouth. Try setting it to something smaller, like `.10`, and playing the game. You'll notice how much slower the animation is running. Return it back to `1`, and let's move on to the next step of our game, which is creating prefabs.

Before we move on, you should save your scene. You'll notice that our game needs to be saved by the asterisk in the title of the project.



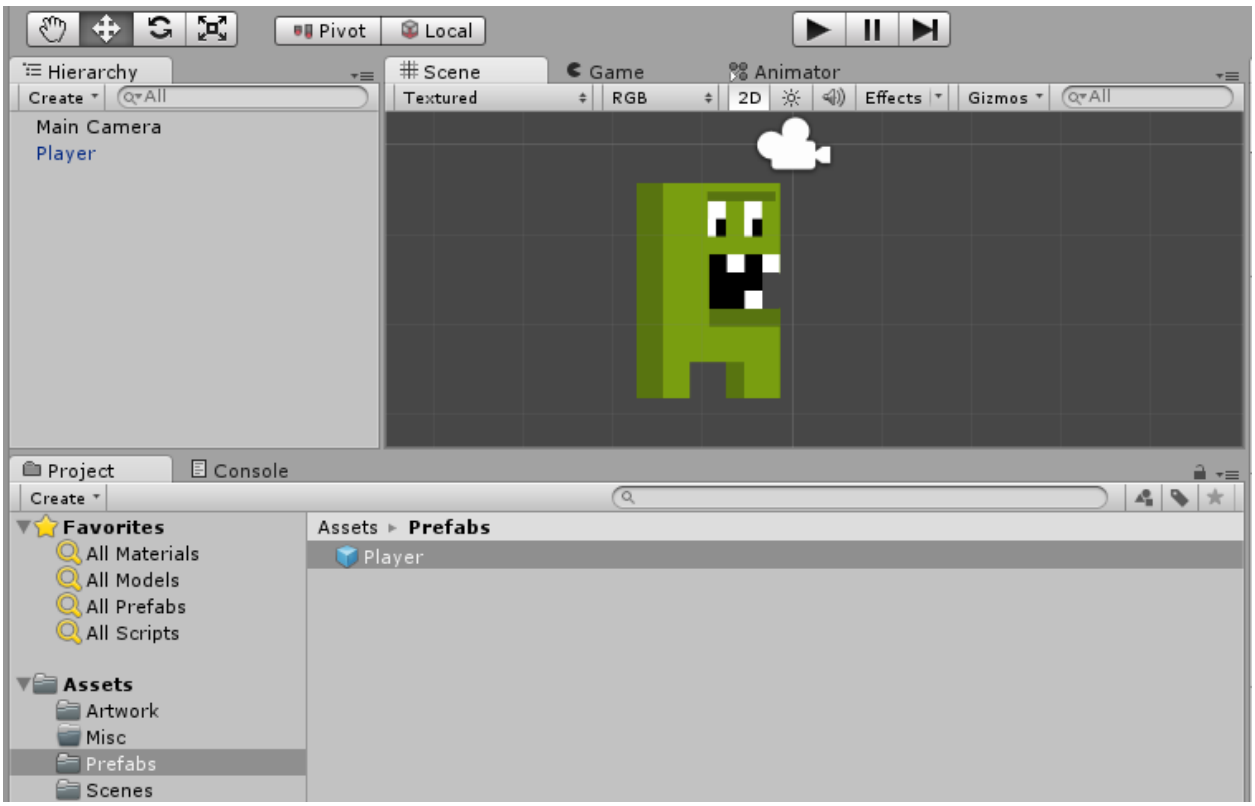
Simply select Save Scene from the File menu.



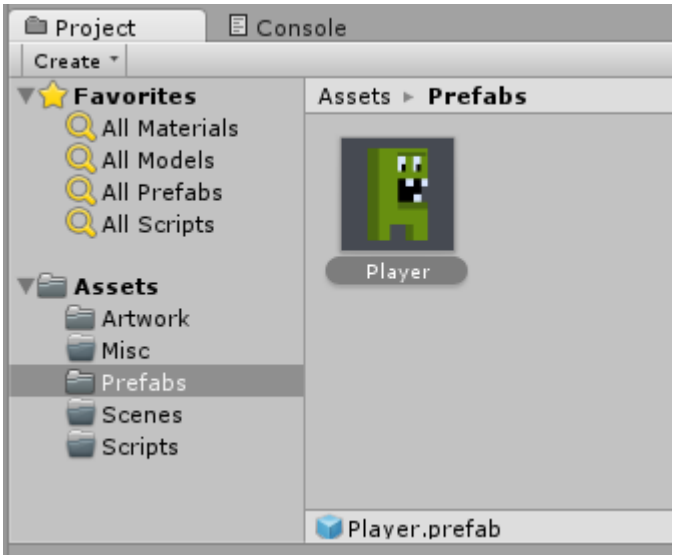
And save it as `Game` in our Scenes folder. It is a good idea to get in the habit of saving often, especially after each of the sections in this book.

Creating Prefabs

At this point, we have an animated sprite on the stage, but not much else. We are going to start creating reusable GameObjects called prefabs in our project. Our first one will be this monster. Before we start, you should give it a more descriptive name in the Hierarchy tab. Let's call him `Player`. Then drag the `Player` instance from the Hierarchy tab to our `Prefabs` folder.



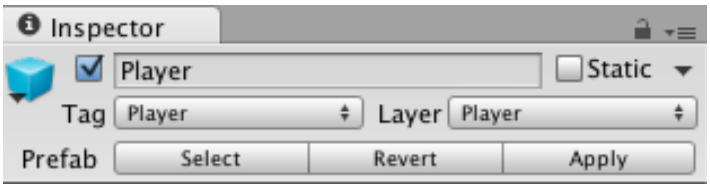
You will now notice that the instance of the `Player`'s name is now blue in the Hierarchy view. We now also have a `Player` prefab in our `Prefabs` folder.



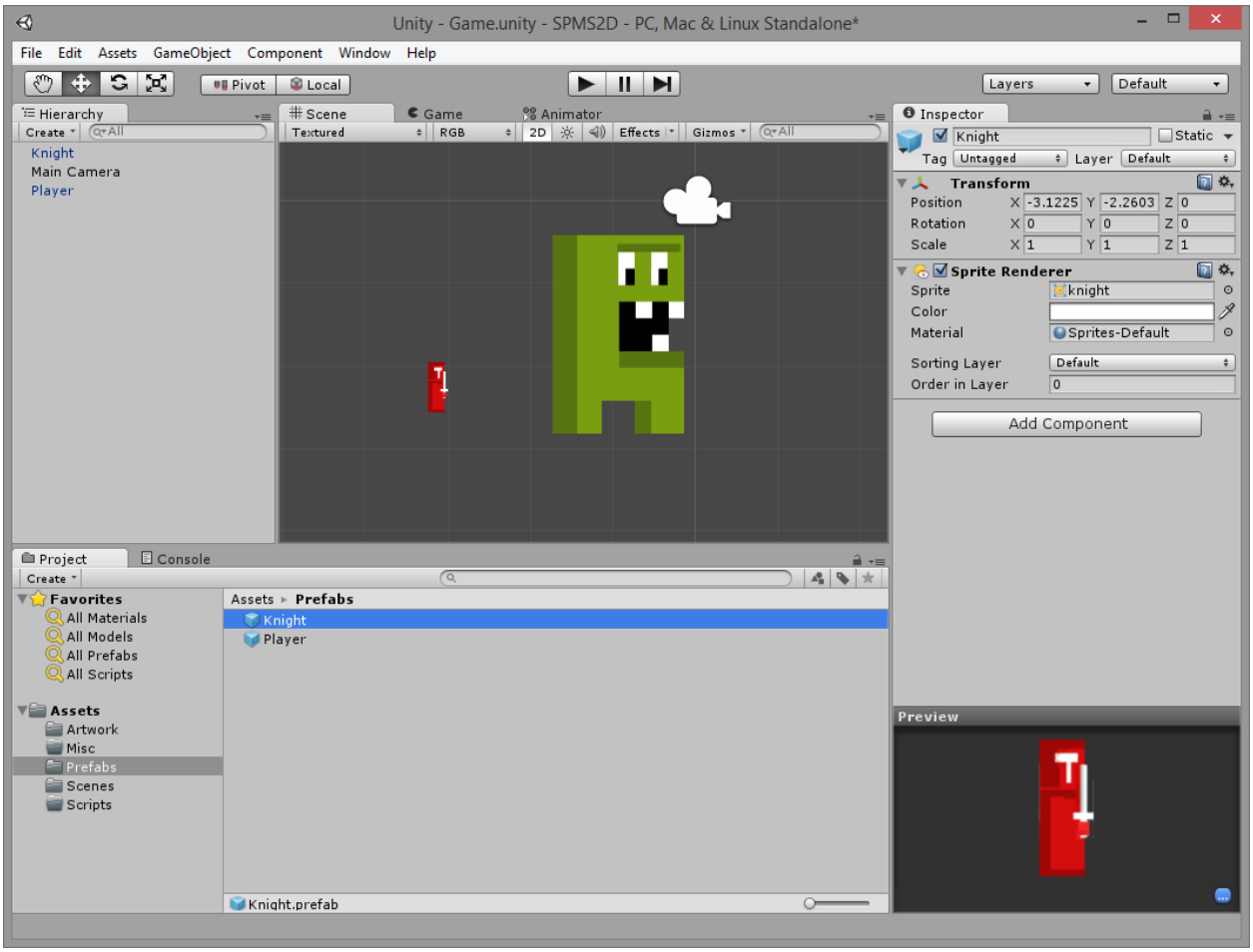
It is important to note that, while you can continue to edit the instance of the `Player` in the Scene, none of those changes will be saved to the actual prefab itself. Likewise, as you continue to modify the prefab, some instance-specific properties, such as position, scale, and more, will not be updated on the instance itself. This disconnect is a little frustrating to deal with at first, but it helps to understand that our `Player` prefab is the base template of all `Players` in our game. Any instance of the `Player` in

the Scene has its own unique values from the base properties, allowing you to customize each individual prefab derivative on a case-by-case basis. While you may be questioning this for the player, it will make more sense when we create the bad guys.

If you are working with an instance of a prefab in the scene and want to update the actual prefab itself, you can use the inspector's Apply button.



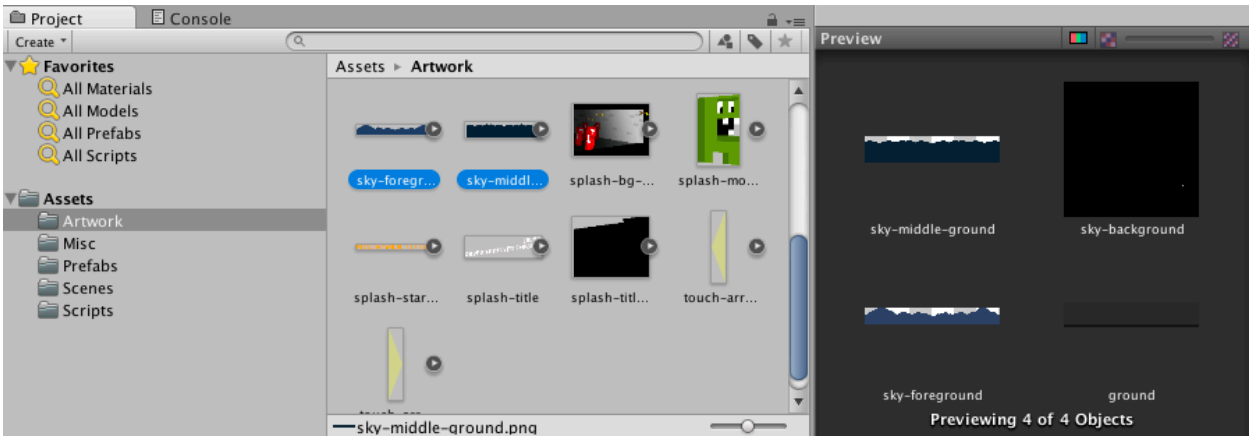
Let's make another prefab for our game. Find the knight sprite in the `Assets` folder, drag it to the Scene, and then rename him `Knight`. Once you have done that, drag the instance in the Hierarchy tab back into the `Prefabs` folder.



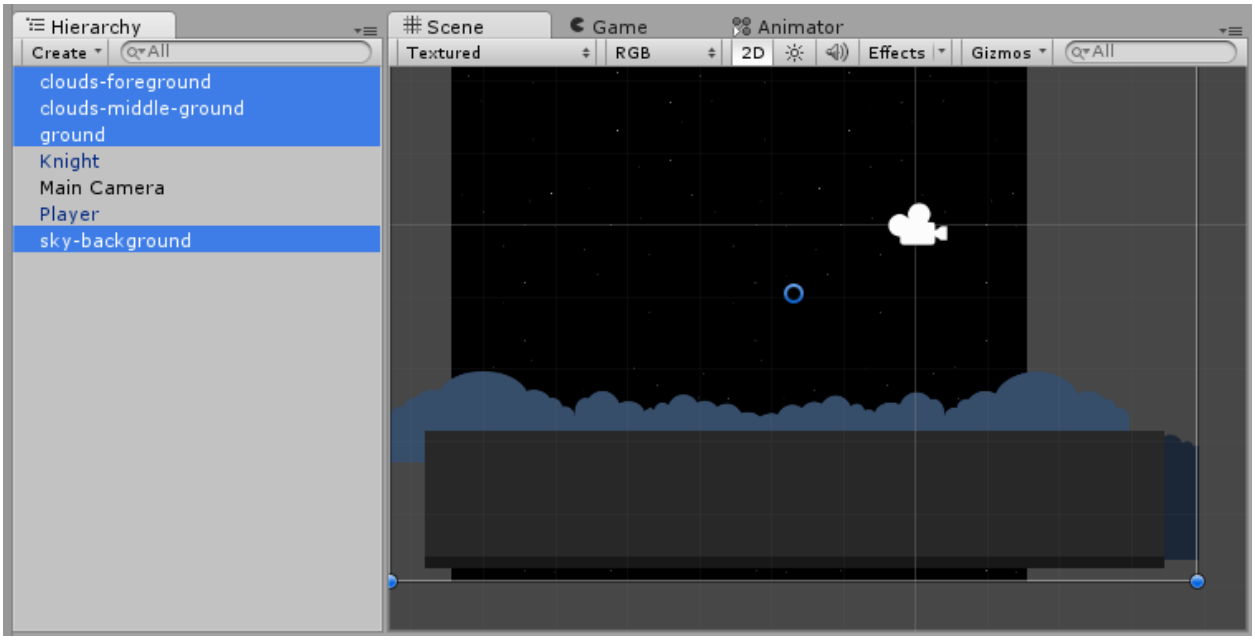
We now have our basic characters for the game. Let's look into making the rest of the level.

Laying Out the Scene

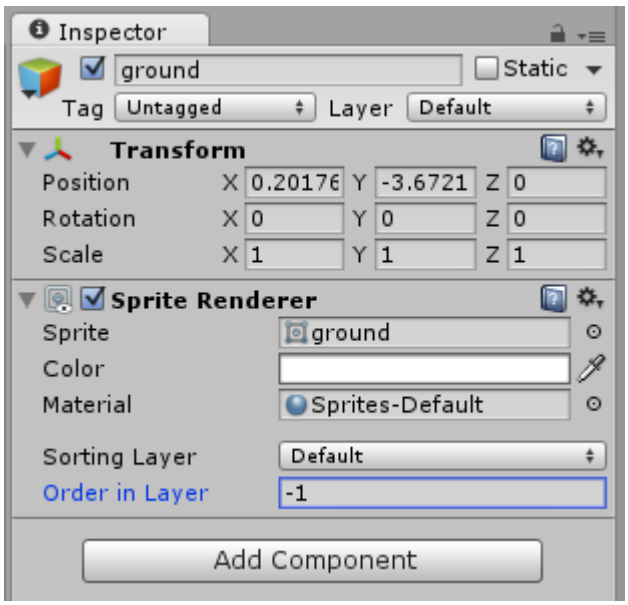
For this introduction, I am not going to get into too many details on building a complicated level layout. Our level will consist of four layers: the ground, the clouds in the foreground, the clouds in the middle ground, and the clouds in the background. You can find them in the Artwork folder.



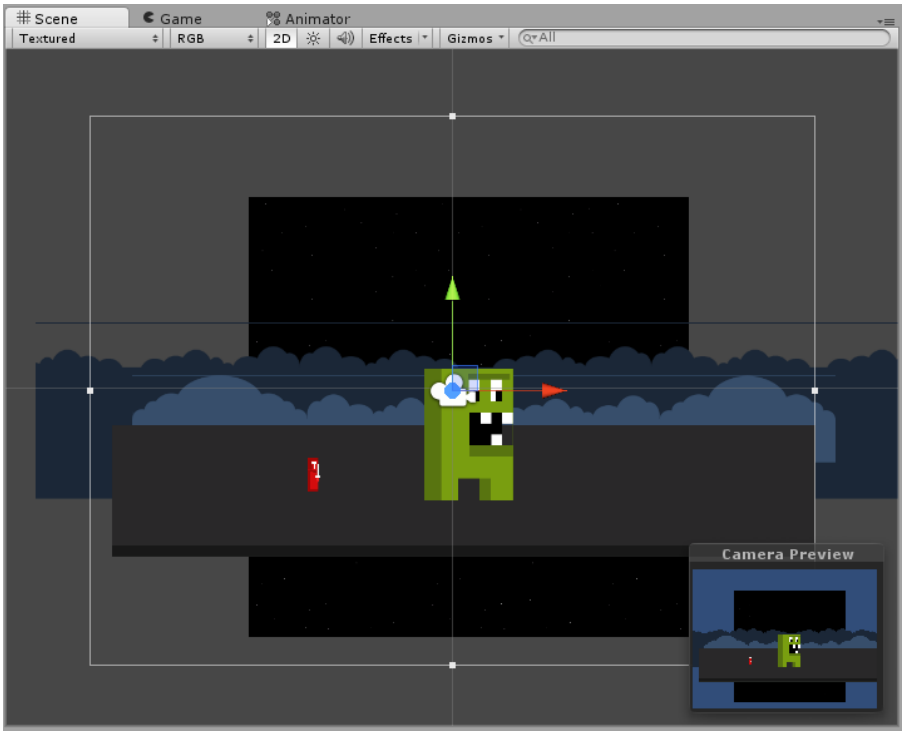
You'll probably notice that, when you try to drag them all to the Scene Unity wants to make an animation out of them. To avoid this, you will need to drag each sprite over one at a time. When you are done, they will probably look like a mess in the scene and will be covering up the `Player` or `Knight` we just created.



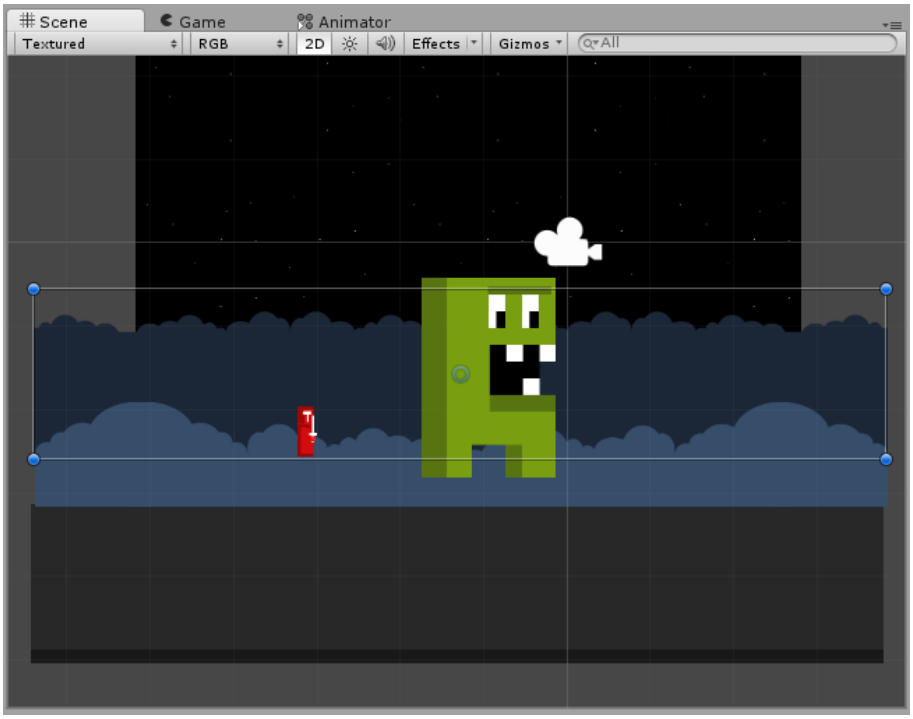
In order to fix this, we will need to change the layer order of the GameObjects. Let's start with the `ground` by selecting it and going into the Inspector. Under the Sprite Renderer component you should see a setting for `Order in Layer`, set that to `-1`.



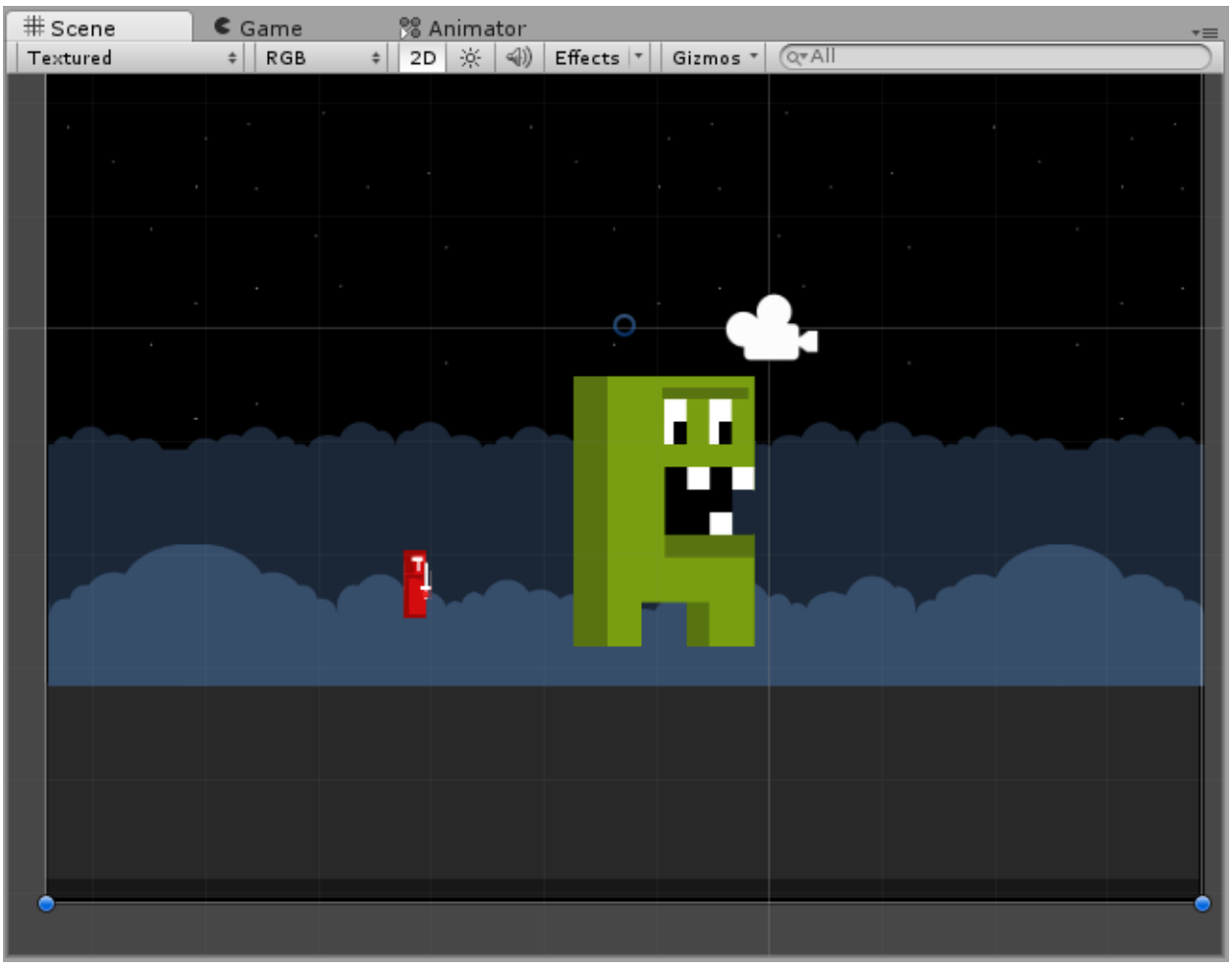
This will place it behind our Player and Knight, since they will remain at the `0` order of the layer. After that, set the `clouds-foreground` to layer order `-2`, so it will sit on top of our `ground`. The `clouds-middle-ground` should get a layer order `-3`, and the `sky-background` will get a layer order of `-4`. Make sure you preview the camera to ensure everything is still in view. You should end up with something like this:



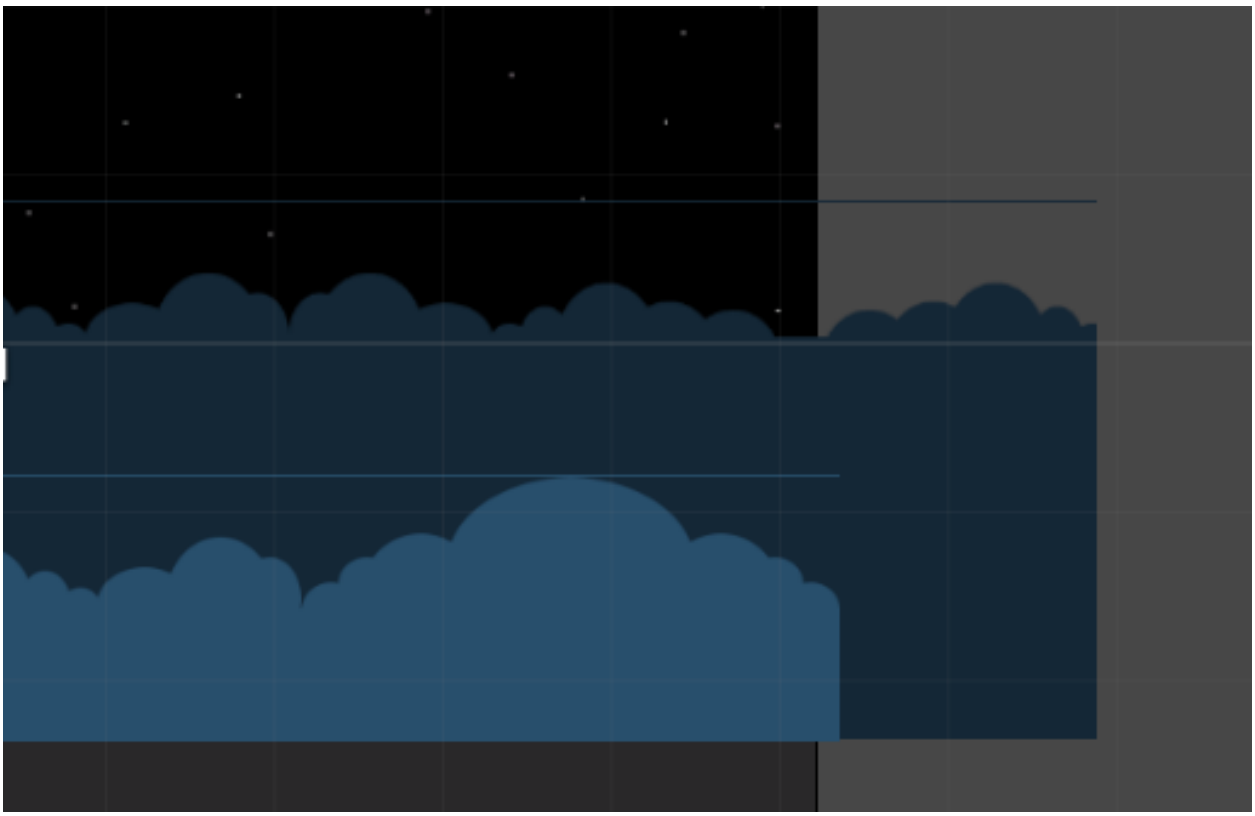
Now we can take advantage of one of Unity's greatest feature, which is its ability to let you build out your game visually. Let's change the layout of the level to look more like this:



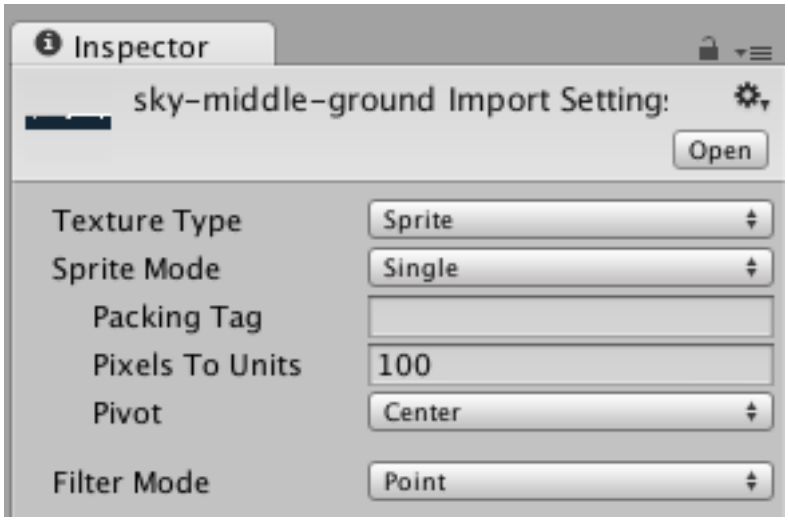
Basically, the way this works is that our foreground clouds cover the top part of the ground. The middle-ground clouds go right up to the lowest dip in the foreground clouds, so you always see dark blue behind them and the sky is in the middle. While you are at it, make the sky a little larger to match the width of the other layers.



As you can see, you are easily able to resize anything in the Scene and move things around, just like in a photo editor. Unfortunately at this point you may notice some strange artifacts showing up in the background images that look something like this:

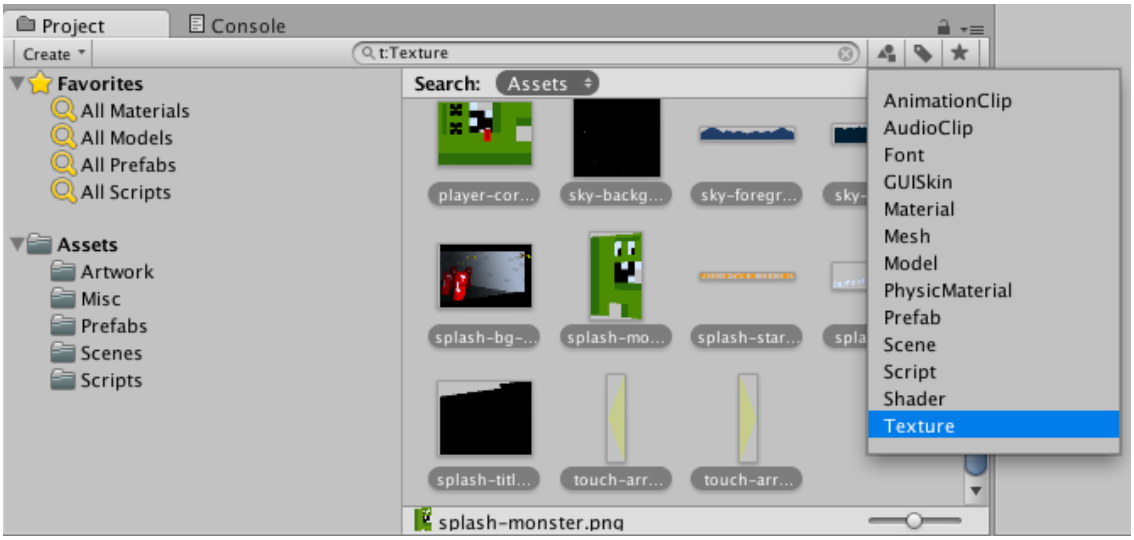


Going into our Artwork folder and selecting one of our sky images to easily fix this. Once you have the sprite selected, go to the `Filter Mode` in the Inspector and change it from `Bilinear` to `Point`.



Make sure you hit apply and you will notice that the artifact will disappear in the Scene inspector.

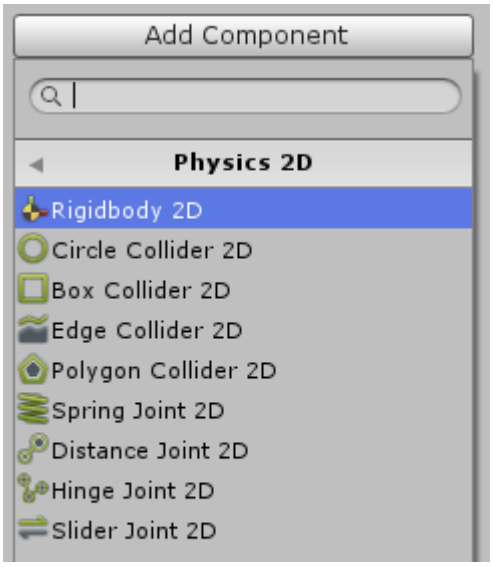
Go ahead and do this for all of the level artwork and the other textures as well. Simply select them all in the artwork folder by clicking on the first file and while holding down Shift, click the last one. You will also need to filter out anything that is not a texture. To do this, select the icon to the right of the search box and select Texture, which will highlight all the Textures in the directory.



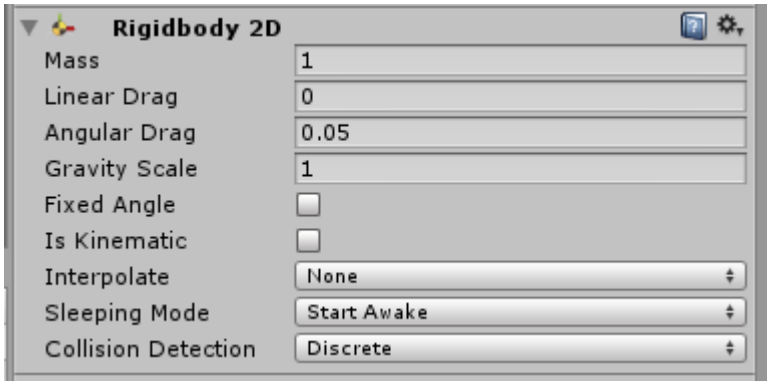
This will allow us to make a group modification. By setting all of our textures to Filter Mode Point we will have clean looking pixel art in Unity. Now we are ready to set up some physics and collisions to give this game some life.

Working with Box2D

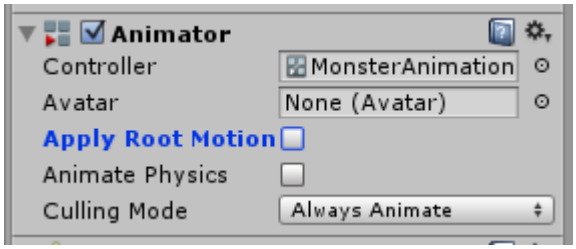
With the addition of the new 2D workflow in Unity, there is now Box2D to handle all of your in-game physics. This is great news because you can take advantage of a common 2D physics engine, as well as avoiding the overhead of the built-in 3D physics engine. To get started, let's add a new component onto our `Player` prefab. Select the `Player` in the `Prefab` folder, not the one in the Hierarchy tab, and in the Inspector panel click Add Component. From there, select Physics 2D to see all of the options.



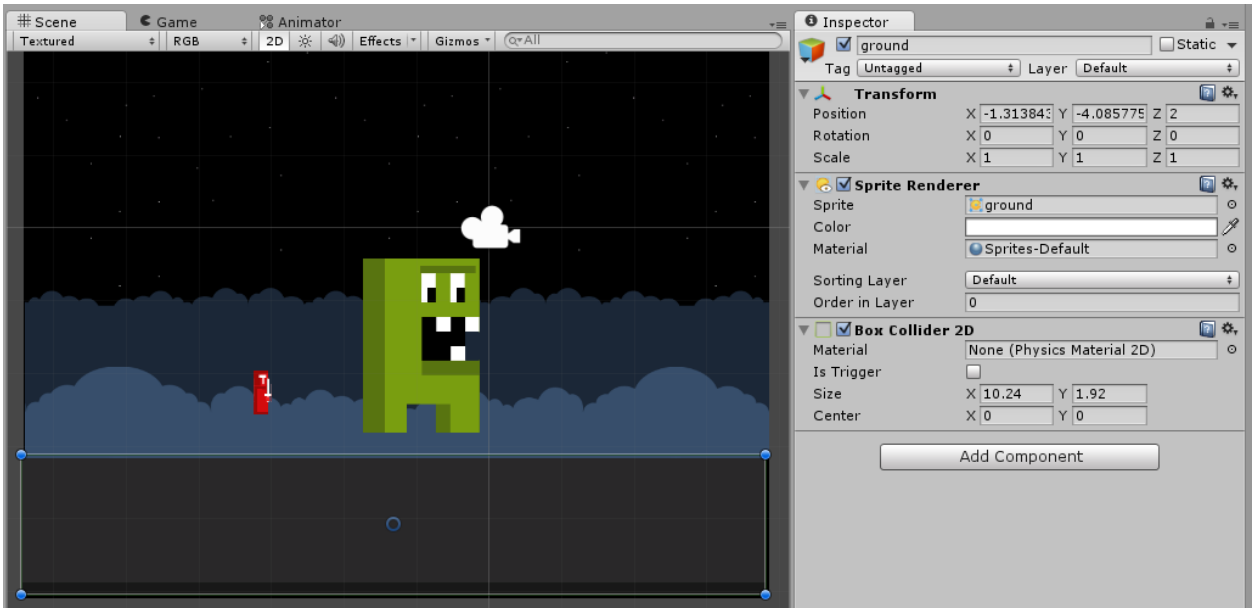
To get started, we are going to add Rigidbody 2D to our Player prefab. By adding a Rigidboxy 2D component we are telling our GameObject that it needs to obey the laws of physics in our game. We will have the ability to manage the GameObject's `Mass`, `Drag`, `Gravity Scale` and a few other important properties that help make up the physics of the Box2D engine. Now that you have the Rigidbody 2D added to our prefab, you should see these new properties.



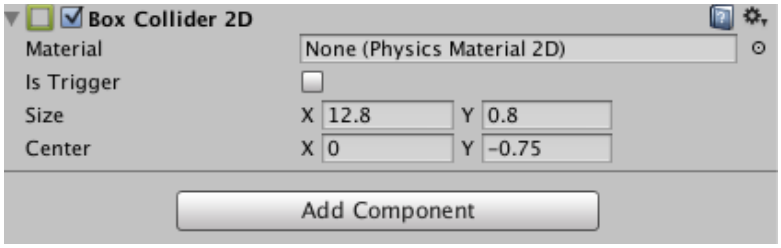
Try to run the game. At this point we would expect the player to fall since it has a `Gravity Scale` of `1` but it just floats in one place. To fix this, make sure you go into the Animation settings on the `Player` prefab and uncheck `Apply Root Motion`, which is selected by default. This is a setting that is normally used in conjunction to Unity's 3D animation system, Mecanim and is set to true by default. We will need to disable this since our sprite animation doesn't have any motion data attached to it.



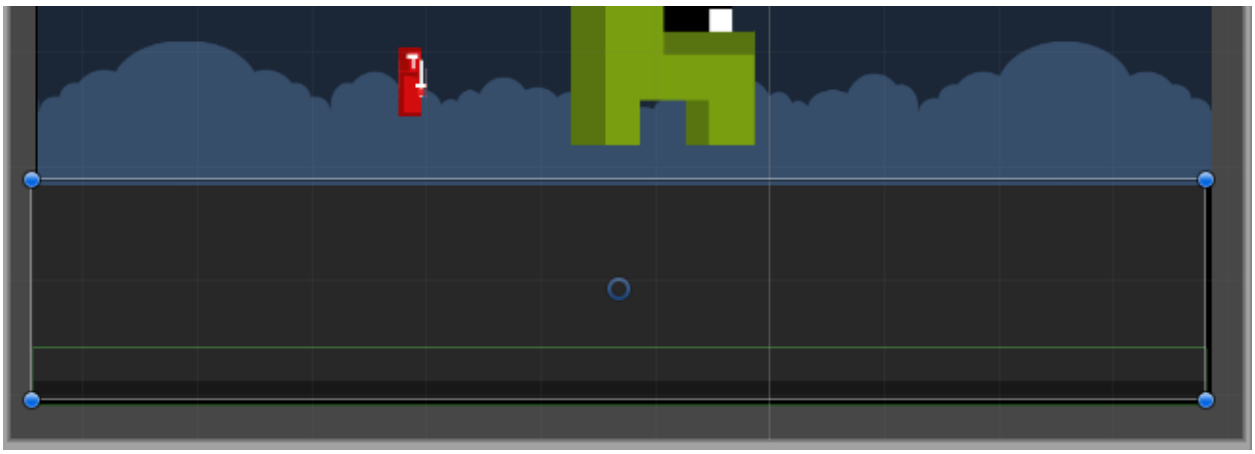
Once this is fixed and you run the game, the Player will fall off the screen since gravity is being applied to it. To counter this, we are going to want to make the ground GameObject solid. Select the `ground` instance in the Hierarchy view, or in the Scene itself, and add a Box Collider 2D component to it.



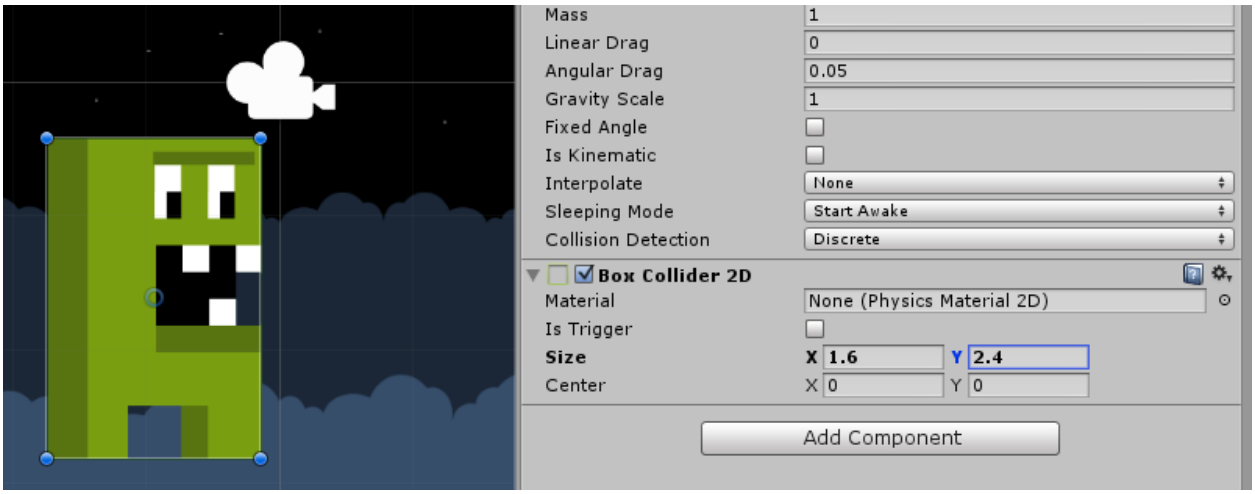
Here you will see we can define the size of the collider and its offset. Before we move on, we are going to want to clean this collider up a little. We want the player to look like he is walking inside of the ground area, not on top of it. To make this happen, we will need to modify the `Size` and `Center` of the collider. The `Size X` value should be `12.8` and the `Y` value is `0.8`. Likewise the `Center X` value will be `0` and the `Y` value is `-0.75` like so:



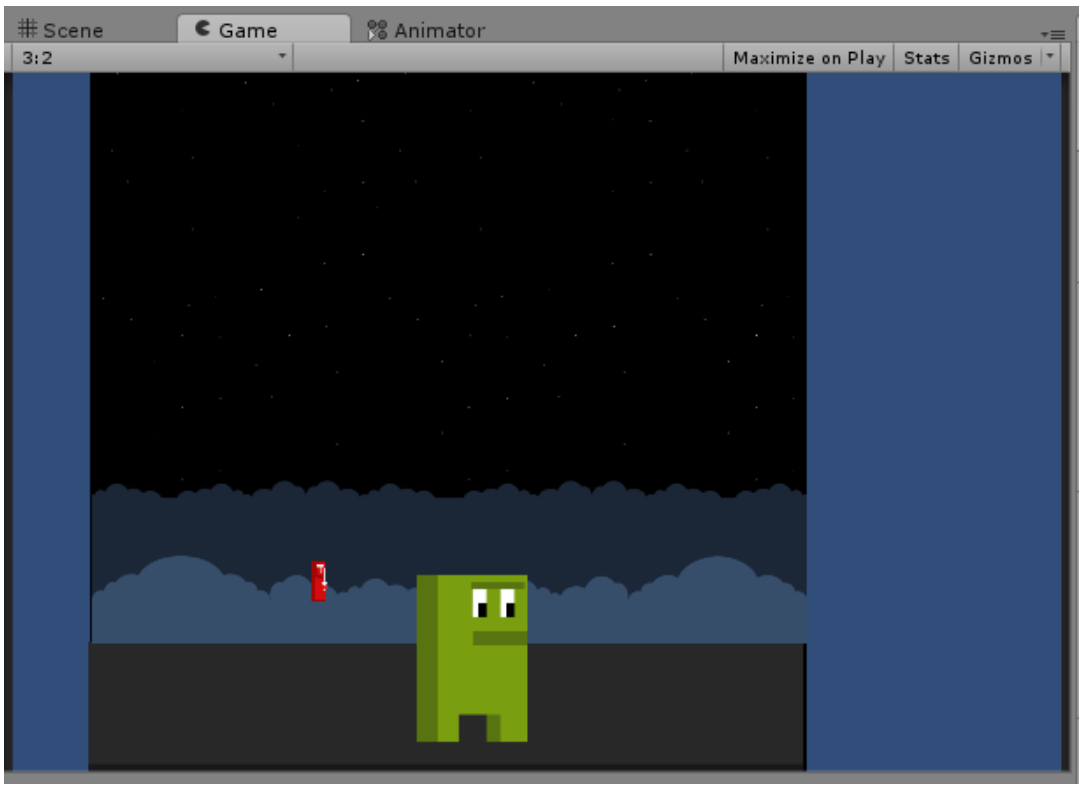
Now, if you look at the ground, you will see that the green box that helps us visualize the collider is now closer to the bottom of the image.



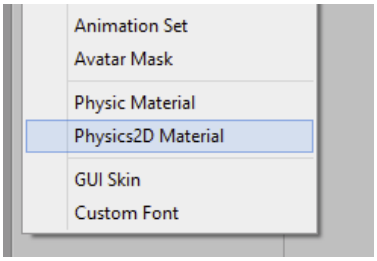
The last thing we need to do is go back into our `Player` prefab and add a Box Collider 2D to it. Once you have added the collider, you should be able to test the game again and see that the `Player` now falls onto the ground and no longer goes through it. We'll just need to make sure the player's collider has the same dimensions as the artwork. If your collision box is not the same size as the sprite, set its `Size X` value to `1.6` and the `Y` value to `2.4` like so:



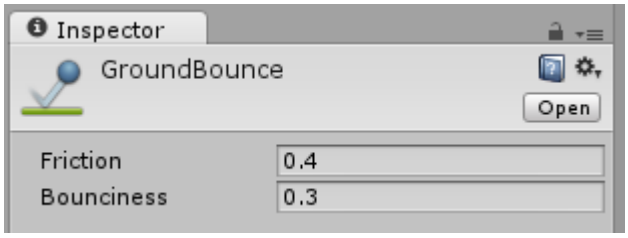
Now try running your game and you will have the Monster fall into the correct position.



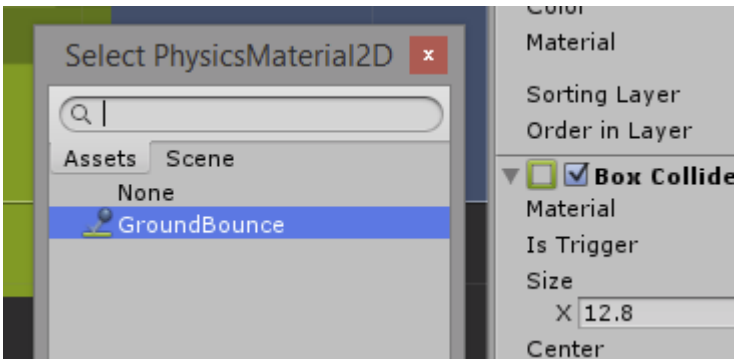
Right now the Player is kind of boring when he drops. We should add a little bounce to him. To do this, we will need to create a special kind of component called a Physics2D Material. To do this, we will need to click on the Create button in the Project tab and add this to our `Misc` folder.



Name it `GroundBounce` and change the `Bounciness` to `.3`.

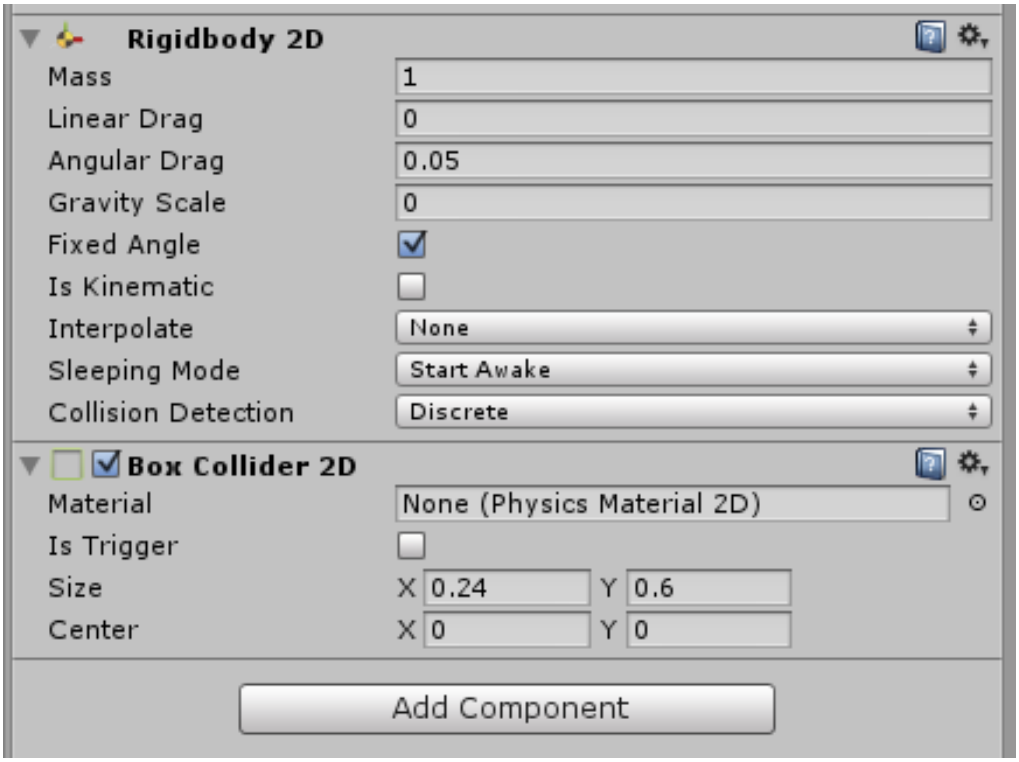


Now we can add this to our ground GameObject's Box Collider 2D component. Select the `ground` from the Hierarchy tab and click on the little circle next to the Material input field on the Box Collider 2D component. You will get the following menu that allows you to pick from any GameObject in the current scene, or from our `Assets` folder.

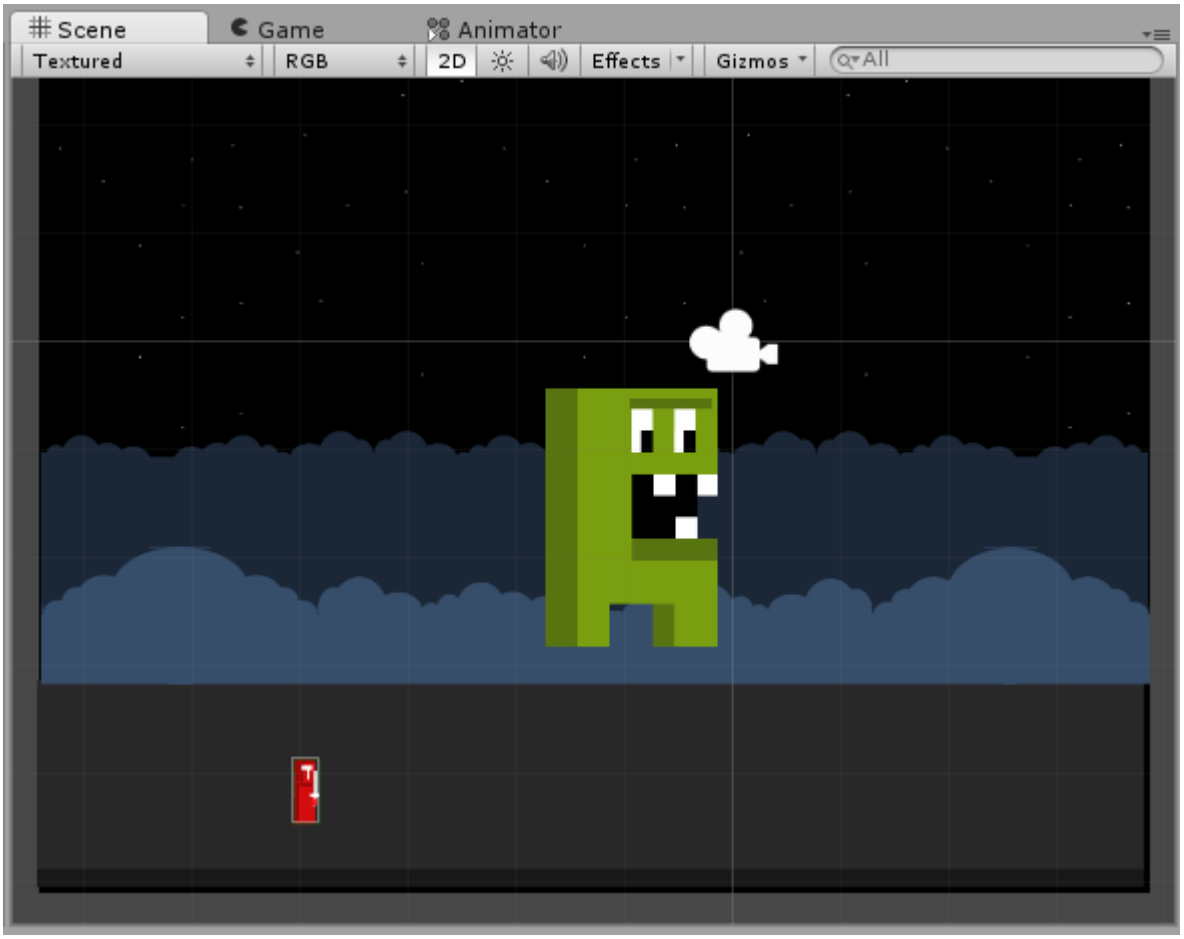


Select `GroundBounce` and then run the game. Your Player should do a nice bounce when he hits the ground.

The last thing we are going to want to do is set up our bad guy to be able to collide with the player later on when we start to move him. Select the `Knight` prefab and let's add some components to it. Start with adding a Rigidbody 2D then setting `Gravity Scale` to `0`, and checking `Fixed Angle`. Also, add to Box Collider 2D a `Size` of `X` to `.24` and `y` to `.6`. In the end, it should look like this:



By setting the `Gravity Scale` value to `0` we insure that it will not fall no matter where we end up spawning him on the screen and checking off `Fixed Angle` insures that he will always stay upright no matter what he collides with. Now let's talk about how to move the player and the bad guys. Before we do that, just put the instance of the `Knight` prefab in the scene closer to the ground so he will be ready to move in the next section.



Making Things Move

At this point, we are ready to start coding. We'll begin with getting our bad guys to walk forward. To do this, we will write a simple script in C#. Let's create this in the `Scripts` folder and call it `MoveForward`.

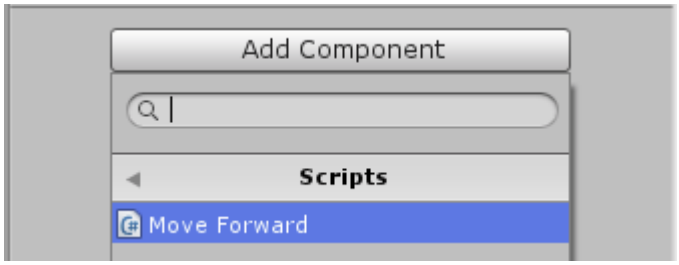
Once you have your new class open in your code editor, add the following property:

```
public float speed = .3f;
```

You can also remove the `Start` method; we won't be using it. Now add the following code to the `Update` method:

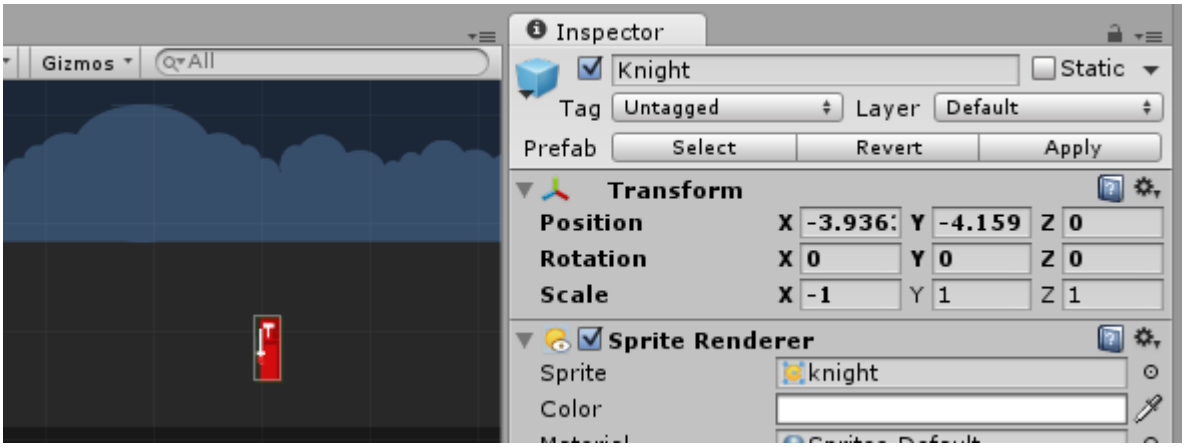
```
rigidbody2D.velocity = new Vector2(this.transform.localScale.x, 0) *
speed;
```

Basically, what this script is doing is setting the velocity of the default Rigidbody 2D component to match the `localScale` multiplied by `speed`. This will move the Knight forward since the default `localScale` `x` value is `1`. So every frame we are basically setting the velocity to .3. I'll show you why multiplying the `speed` by the `localScale` is important after we finish setting it up. All we need to do now is open up the `Knight` prefab and add this script as a component.



Now, if we run the game, you should see your `Knight` move forward. You can always adjust the `speed` value by changing it on the instance itself, the prefab so all newly created instances get the new value, or finally in the code itself. Later on, when we dynamically spawn these `Knights`, you can play around with assigning a new speed to them via code.

There is one more thing I want to go over with the knight, which is how to control what direction he is moving in. You may have noticed that we are taking the `localScale` and multiplying it by the `speed` to set the velocity. If you select the instance of the Knight in the Scene and look at the top of the Inspector panel, you will see the `Transform` component. The last value here `Scale`. A neat trick for making any GameObject face to the left, assuming their default position is to face right, is to change the `Scale X` to `-1`.



Here you will see that the `Knight` is now facing left. If you run the game, he will automatically move to the left now. This is because we set him up to use `Scale` as the speed modifier. If you think about it, positive velocity will move him to the right and negative velocity will move him to the left. Now we can start talking about moving the player.

Let's create a new script that we will use to manage our player's movement. Call the script `Player` and attach it to the `Player` prefab. Add the following properties to the class:

```
public float speed = 200;
public float maxSpeed = 5;
int moving = 0;
```

This movement script is going to be a little different. Not only will we have a speed, but also since we will be applying a force to move the player, we will need to make sure that the player doesn't exceed the maximum speed we want it to move at. In addition, since we don't want to be applying that force

when the player doesn't have a key down, we will need to know when we want to be moving or when we should stop.

Now let's start handling the basic logic to change the direction the player is facing. Delete the `Start` method, since we don't need it, and add the following to the `Update` method:

```
if (Input.GetKey("right"))
{
    moving = 1;
}
else if (Input.GetKey("left"))
{
    moving = -1;
}
else
{
    moving = 0;
}
```

Here you can see we are going to change the value of the moving property based on the key being pressed. In Unity, it is very easy to get the current key being pressed. Simply ask the `Input` class for the key and pass in the key you want. Here we are simply looking for the right key to set moving to 1, the left key sets it to -1, and if neither key is pressed we set it to 0. Now add the following code after the conditional we just added:

```
if (moving != 0)
{
    var velocityX = System.Math.Abs(rigidbody2D.velocity.x);

    if (velocityX < .5)
    {
        rigidbody2D.AddForce(new Vector2(moving, 0) * speed);

        if (this.transform.localScale.x != moving)
            this.transform.localScale = new Vector3(moving, 1, 1);
    }

    if (velocityX > maxSpeed)
        rigidbody2D.velocity = new Vector2(maxSpeed * moving, 0);
}
```

There is a lot going on here, but we'll talk through what is going on. First, we test to see what the value of moving is. If the Player should be moving, we get the absolute value of the Player's current velocity. Remember back to our knight that we use positive for right and negative for left. Using `System.Math.Abs` will always turn a value as a positive number, so it will easier for us to get an accurate idea of the actual velocity and not the direction it is being applied in. Now, with the `velocityX` value, we can test to see if we should apply a force.

Let's talk about velocity for a second here. Up until this point, we have always been applying a constant value to a GameObject's velocity, but if we stopped applying that force, it would gradually ramp down to 0. For our player, we are simply testing to see if the velocity is at half of its value before we are able

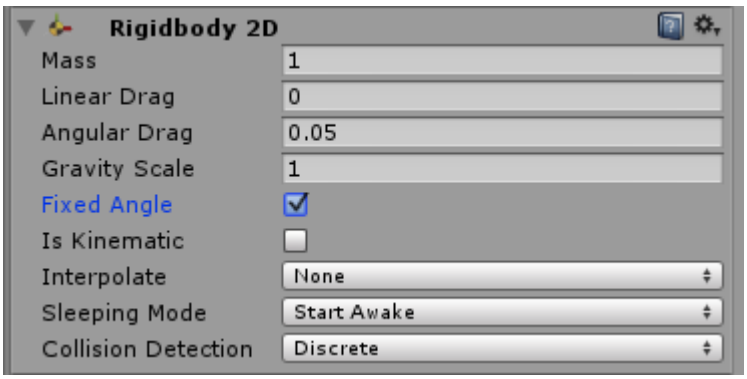
to apply a force to our `Player` instance. In the next line, you will see how we move a Box2D object by simply calling `AddForce` on the `rigidbody2D` property. This method accepts a `Vector2` representing the force's `x` and `y` values. Since we just want to move the player along the x-axis, we will set the `x` value of the `Vector2` to `1` and leave the `y` value at `0`.

Now we do a quick test to see if the Player's `localScale.x` does not equal the moving value. If it doesn't, we simply update the `localScale` to match the direction the `Player` instance is moving in. And, at the end of all of this, we make sure that the Player's velocity is not greater than the `maxSpeed` we allow the Player to move at. If the value is greater, we simply reset the `rigidbody2D`'s velocity to the `maxSpeed`. The end result should give the effect that the player is sliding forward based on the direction of the key that was pressed. This will be a little different then what you may be used to in a game since we want the monster to appear like it is lunging towards the bad guys.

While it may look like a lot of code, it's actually very straightforward, and these are the basics of how movement works in Unity. The best part is that we now have two encapsulated scripts we can reuse in other games if we ever need an enemy to simply move forward or want to move a Player left and right along the x axis.

You may have noticed that the `Player` can be knocked down if you run into the `Knight` too quickly.

Since we are using Box2D physics, objects in the game world are going to react a little differently than you may expect in a normal 2D environment. As you may remember, we had set the Knight to have a `Fixed Angle`. This locks it into place so any other GameObject that collides with it will attempt to move around the smaller GameObject. In this case our `Player` will try to fall over the `Knight`. To help fix this, let's open up our Player prefab and check the `Fixed Angle` toggle option in the Rigidbody 2D component.



Now the `Player` GameObject will not fall over the `Knight`. You may want to do the same thing for the `Knight` prefab as well, if it's not already checked.

Now there is one last thing we should do when it comes to controlling our player. We saw how easy it was to use the keyboard but eventually we will want to publish this game to mobile or maybe the player likes using the mouse. To handle this we can use a few extra lines of code to add basic mouse/touch controls for our game. The idea will be that we will capture the mouse press and detect if it's on the left hand side of the screen or the right hand side of the screen. From there we can detect which direction to move the player in. Let's modify our `Player` script with an extra property:

```
float mouseX = 0;
```

Then at the top of our `Update` method, before where we test for the keyboard input add the following conditional:

```
if (Input.GetMouseButtonDown (0)) {
      mouseX = 90 * ((Input.mousePosition.x - Screen.width / 2) /
(Screen.width / 2));
} else {
      mouseX = 0;
}
```

What this does, which is similar to how we captured the keyboard event, is look for the mouse button to be down. In this case the `0` represents the left mouse button. From there we divide the screen's width in half. This will determine if the value is negative or positive. If the mouse is not down we simply set the `mouseX` to `0`. Now we can simply test the value of `mouseX` and deduce what direction the player should move in. To do this we need to modify the next conditional. We'll need to make the following change from:

```
if (Input.GetKey("right"))
{
    moving = 1;
}
else if (Input.GetKey("left"))
{
    moving = -1;
}
```

to this:

```
if (Input.GetKey("right") || mouseX > 0)
{
    moving = 1;
}
else if (Input.GetKey("left") || mouseX <0)
{
    moving = -1;
}
```

If you run the game now, you should be able to use the mouse to move the player left and right in addition to the keyboard. Now we can talk about the camera and how to make it follow the `Player` GameObject as it moves around the screen.

Modifying the Camera

At this point, we have all the basics in place to run our game. The next thing we are going to want to do is make the camera follow the `Player` GameObject. To begin, select the `Camera` GameObject from the Hierarchy panel in the Scene window. We'll want to start by changing the `Background` color to `black`.

We will also want to tweak the `Size` of the `Camera`. By default, it was set to a `Size` of `5`. Change that to `2` and you should see the camera is now pulled in closer to the action.

If you run the game, the camera is probably not going to capture the action very well. Let's look into making a very basic camera follow script. On the `Main Camera` GameObject, add a new script and call it `CameraFollow`.

In this script we are going to build a very simple update loop that will focus the camera on a GameObject we set as the target. Add the following two properties to the script:

```
public GameObject target;
private Transform targetPos;
```

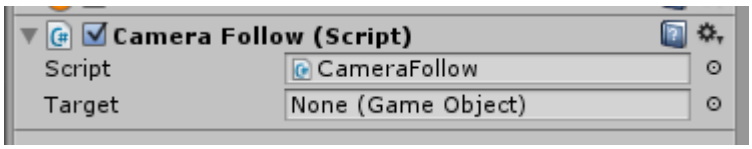
This will help us store the target GameObject and a reference to its `Transform`. Now, in the `Start` method, add the following:

```
targetPos = target.transform;
```

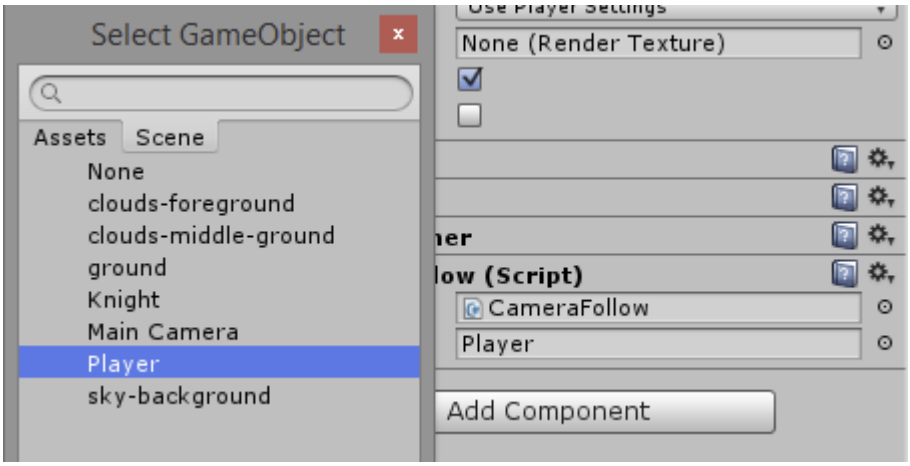
Next we will need to tell the camera to update its position to the target's position. Add this to our `Update` method:

```
if (targetPos)
      transform.position = new Vector3(targetPos.position.x,
targetPos.position.y, transform.position.z);
```

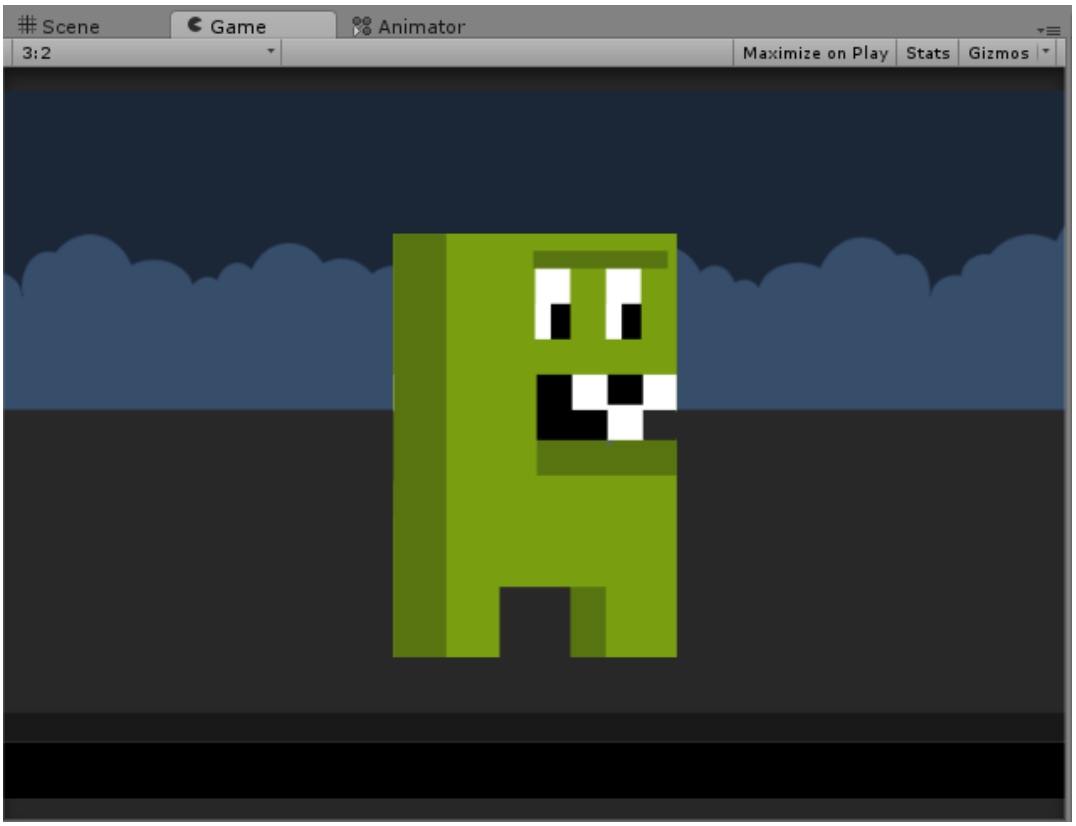
Now we have an incredibly basic camera follow script so add it to our camera GameObject in the Scene and you should now have a `Target` property in the panel.



Click on the little circle to the right of the field and find the reference to the `Player` in the scene.



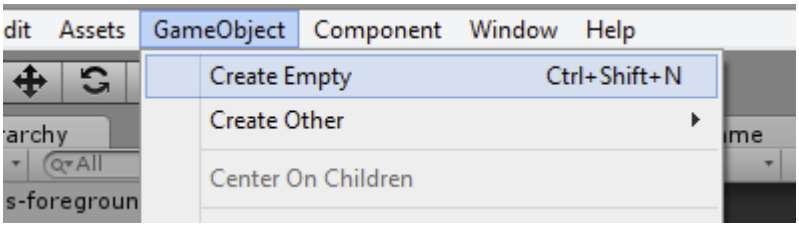
Now our camera will be focused on the Player during the game. Hit Play and make sure everything is working properly.



As you can see, while this script is basic it gets the job done. Now let's talk about bringing the rest of the game to life.

Spawning GameObjects

Our game isn't much fun or challenging right now. We need some more bad guys on the screen. Up until this point, we added new GameObjects manually to the scene. Now it's time for us to actually look at how to dynamically create new GameObjects at runtime. To get started, we need to build a spawner, which is an object that creates other GameObjects. Go to the GameObject menu at the top of the screen and select Create Empty.



This is going to be a simple GameObject that lets us add the code we need to turn it into a spawner. Name the empty GameObject `Spawner` and create a new script on it called `Gizmo`.

A Gizmo in Unity is used for visually debugging GameObjects in your game. A Gizmo has its own draw method that gets called when viewing in the Scene Editor but not during runtime in your game. This will allow us to create a simple box to represent our spawner as we work with it in the Scene Editor, and it will remain invisible when the game runs. Make sure you import generic collections at the top of the class:

```
using System.Collections.Generic;
```

Then add the following properties to the `Gizmo` script:

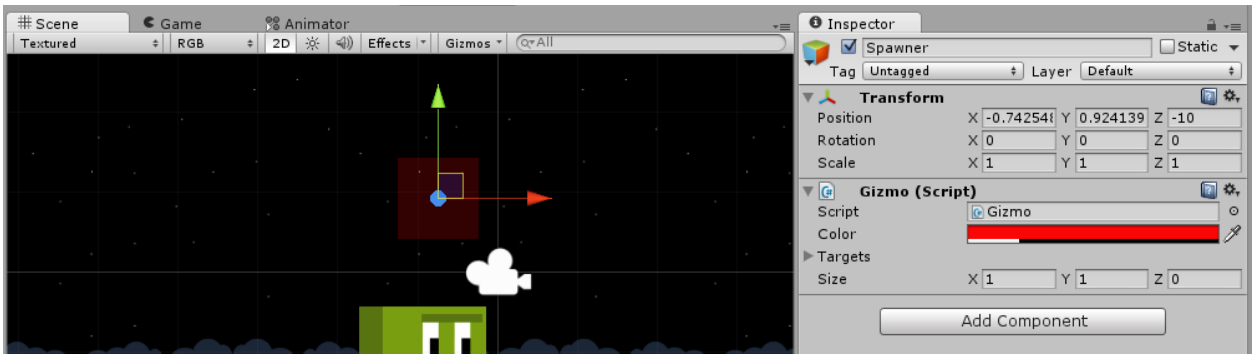
```
public Color color = new Color(0.985f, 0.022f, 0.022f, 0.2f);
public List<GameObject> targets = new List<GameObject>();
public Vector3 size = new Vector3(1, 1, 0);
```

These properties will allow us to change the gizmo's color, its size, and targets which will represent other GameObjects the gizmo is connected to. This will make a little more sense later on when we build out the rest of the spawner logic. For now we need to add the following methods to our gizmo. These next few lines of code will render the gizmo in the Scene Editor, and also help us access the targets via code when we connect other scripts to this script.

```
void OnDrawGizmos()
{
    Gizmos.color = color;
    Gizmos.DrawCube(transform.position, size);

    if (targets.Count > 0)
    {
        for (int i = 0; i < targets.Count; i++)
        {
```

```
            Gizmos.DrawLine(transform.position,
targets[i].transform.position);
        }
    }

}

public GameObject GetRandomTarget()
{
    if (targets.Count == 0)
        return null;

    return targets[Random.Range(0, targets.Count)];
}

public GameObject GetTargetAt(int index)
{
    if (targets.Count == 0)
        return null;

    return targets[index];
}
```

Now let's add the gizmo script to our spawner GameObject. At this point, you should see the spawner rendering as a red box in the Scene tab.



You can change its color by simply clicking on the color dropper next to the `Color` field on the `Gizmo` script, as well as its `Size` below that. Now we are ready to create the spawner logic. Attach a new script to the spawner GameObject and call it `Spawner`. From there you will want to set up the following properties:

```
public GameObject[] enemyPool;
public float delay = 2.0f;
public bool active = true;
private Vector2 direction = new Vector2(1, 1);
private List<GameObject> targets;
private Gizmo parentGizmo;
```

Here we are setting up places to store the enemies we will spawn, the delay between new spawns, whether the spawner is active, as well as some internal values, such as direction to spawn the

GameObjects in, targets for where to spawn at, and a reference to the `Gizmo` script also attached to our spawner. Make sure you import the generic collections again.

```
using System.Collections.Generic;
```

Then in the `Start` method, add the following:

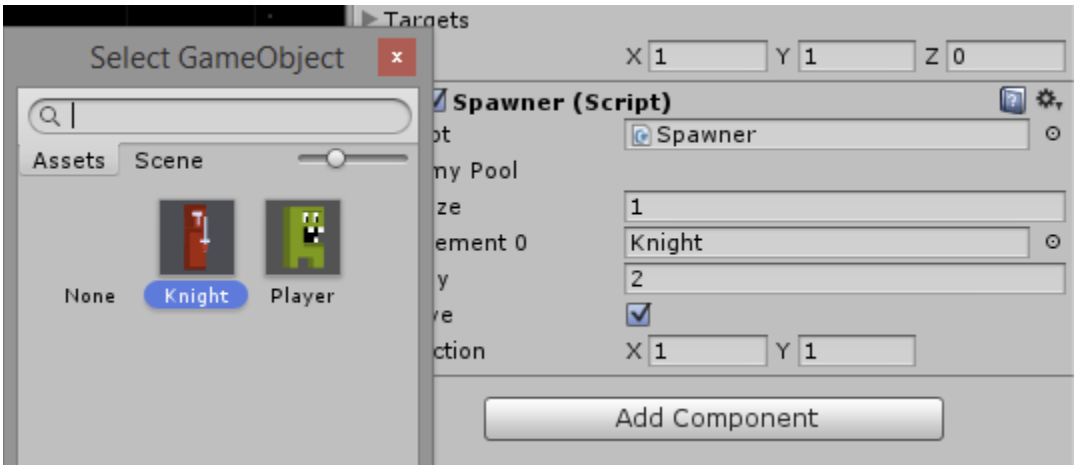
```
parentGizmo = gameObject.GetComponent<Gizmo>();
targets = parentGizmo.targets;
StartCoroutine(EnemyGenerator());
```

The first line is very important. This allows us to actually get a reference to another component script on our GameObject. In this case, we are looking for a reference to the Gizmo script. We will need this reference to get a listing of targets, which we will set up later on the Gizmo script in the Scene editor. The next important part of this method is the `StartCoroutine` call. This allows us to continually call the `EnemyGenerator` method. In that method, we will add a delay to keep it from being called too many times. Here is what that method should look like, just add it below the `Start` one:

```
IEnumerator EnemyGenerator()
{
    if (active)
    {
        var newTransform = transform;

        yield return new WaitForSeconds(delay);

        if (targets.Count > 0)
        {
            var spawnTarget  = targets[Random.Range(0, targets.Count)];
            newTransform = spawnTarget.transform;
            direction = spawnTarget.transform.localScale;
        }

        GameObject clone = Instantiate(enemyPool[Random.Range(0,
enemyPool.Length)], newTransform.position, Quaternion.identity) as
GameObject;
        clone.transform.localScale = direction;
        StartCoroutine(EnemyGenerator());
    }
}
```

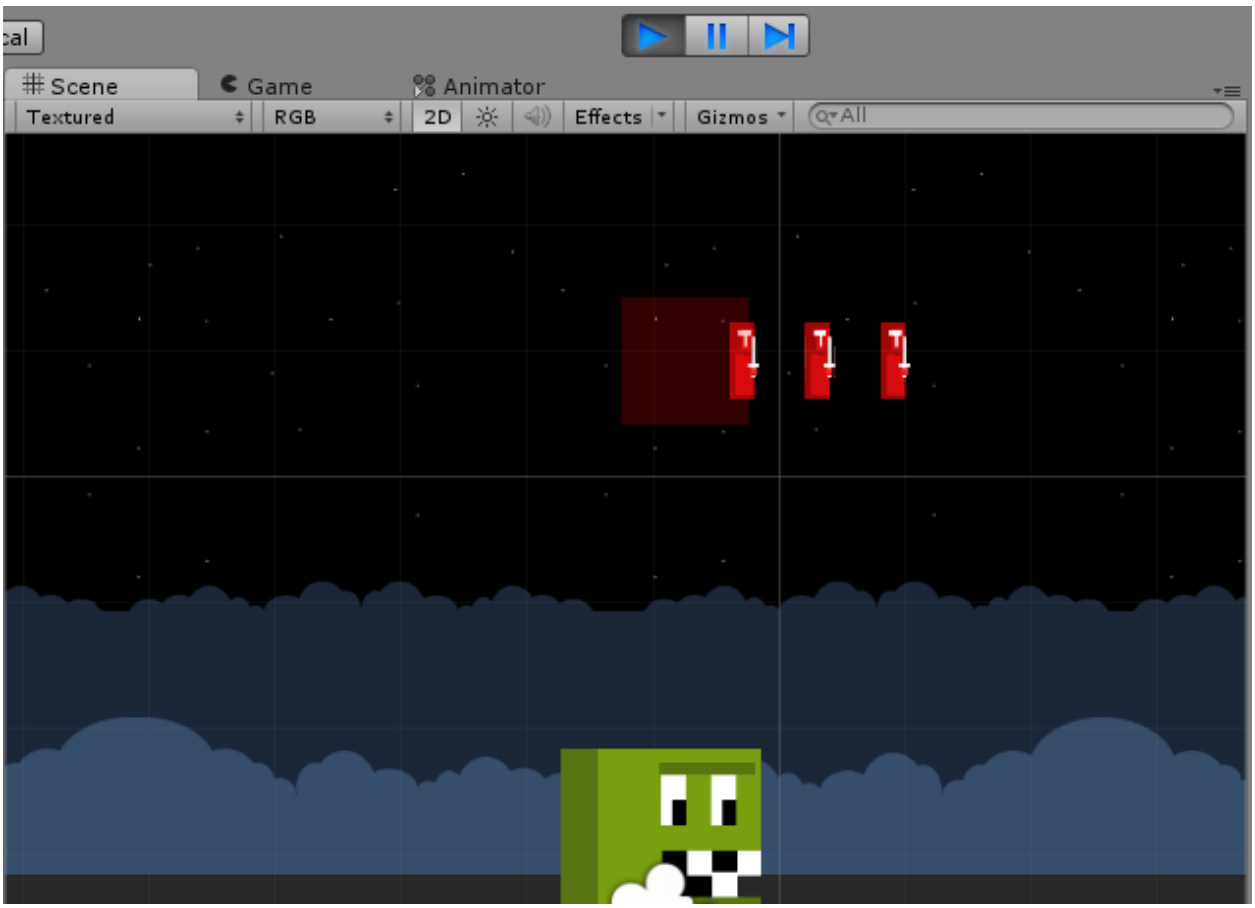
Here you can see we first test if the spawner is active. Next we get a reference to the spawner's own transform. Then we use `WaitForSeconds` while passing in our delay value to decide if the spawner should actually create a new GameObject. You'll notice that this has a return built into it, so if the wait has not happened for long enough, it will exit from the method and not run the rest of the code. If the script is ready to create a new GameObject, it checks for target locations and picks a random one. Then it replaces the spawner's own transform position with the new target. It also sets the direction to match that of the target as well. Finally, it creates a new GameObject randomly from the enemy pool list, modifies its direction to match the direction from either the target or the spawner itself, and then sets a new `Coroutine` to call the `EnemyGenerator` method again.

We now have everything we need to get this working; we just need to set this up in the Scene. Start by populating the Enemy Pool on the component. Set the Size property to 1 and then select the Knight prefab for the first item.

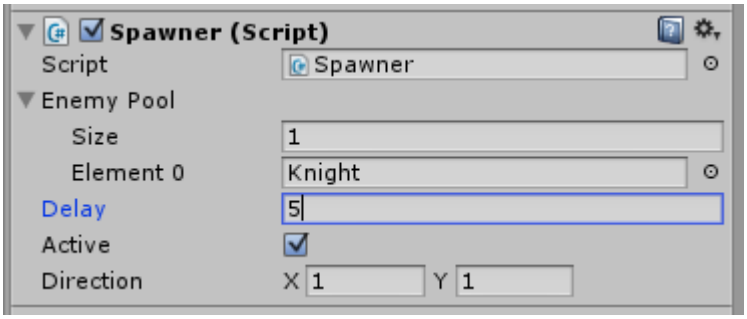


It's important that you use the Knight from the Assets tab and not the Scene tab because we will be removing the Knight GameObject from the Scene tab now that we have the Spawner in place. When you are done setting this up, select the Knight from the Hierarchy tab and delete it from the Scene tab so we don't forget to remove it.

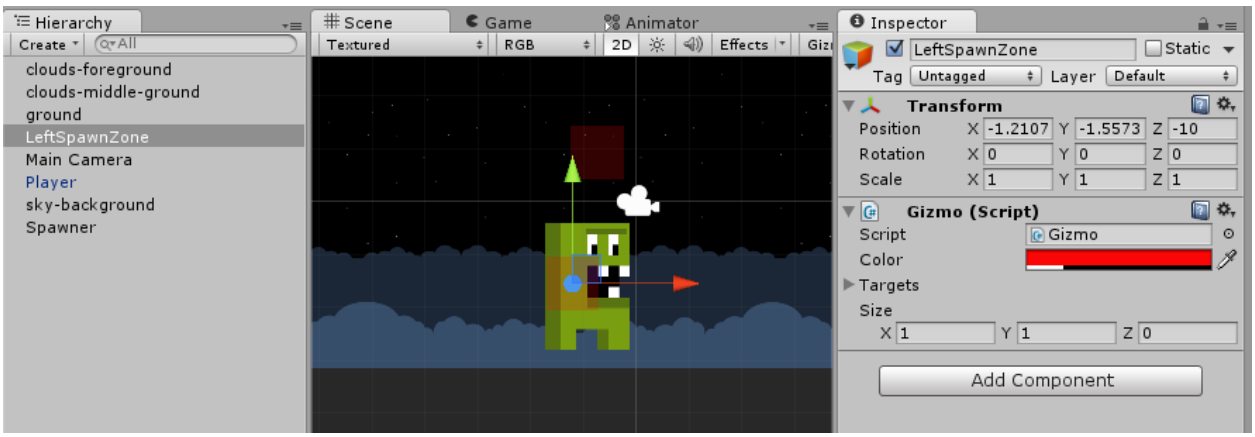
Now, if you run the game and switch over to the Scene tab, you will see Knights getting spawned at the spawner's location, which is in the sky.



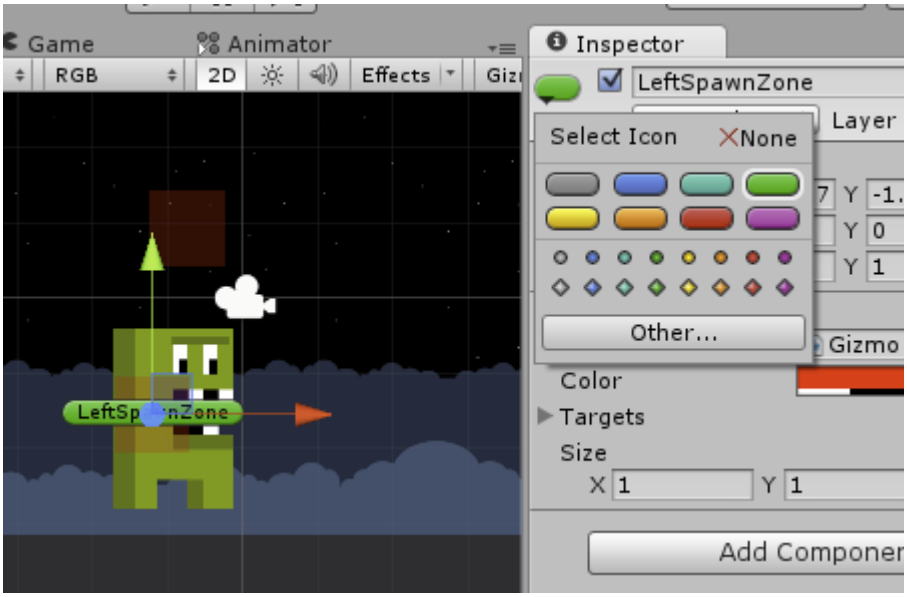
I have moved my spawner into the top area of the background. Also, you may notice that it is spawning a lot of Knights. Simply stop the game, go into the spawner instance, and change the delay value of the spawner component to something bigger, like 5.



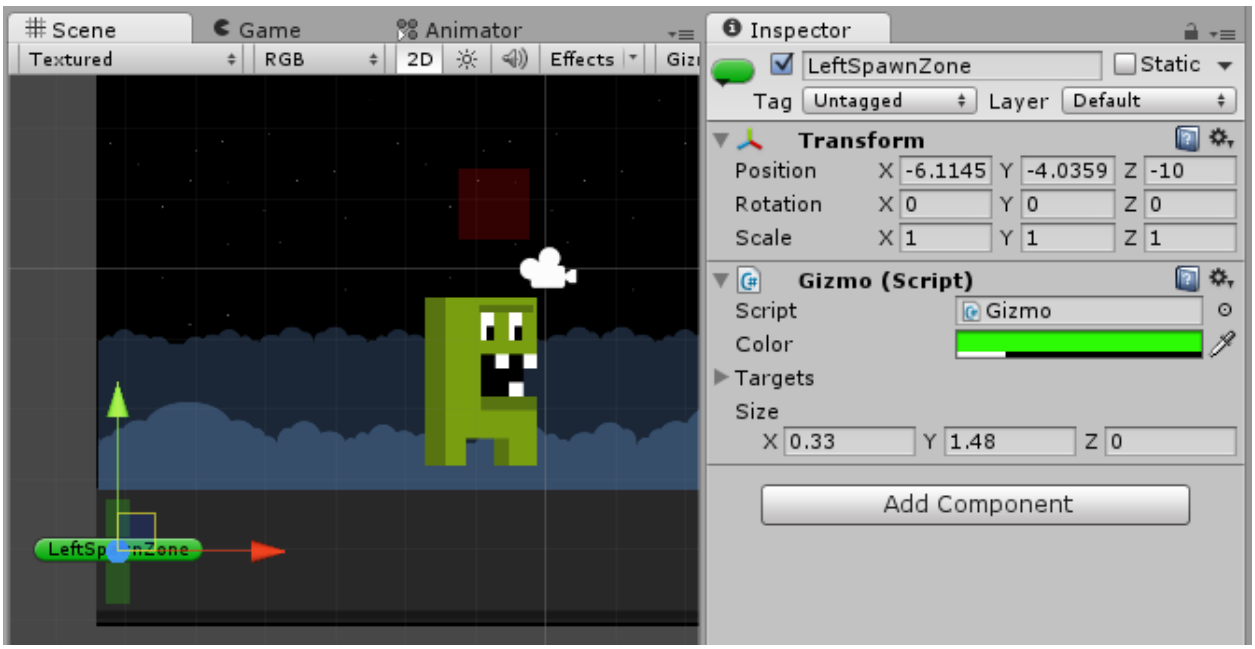
Now you should see fewer `Knights` spawned, but it still doesn't fix that they are floating up in the sky. Next we are going to create some GameObjects to represent spawn points in our scene. Go back to your Scene tab and add a new GameObject; call it `LeftSpawnZone`. Then add the `Gizmo` script to it so we can see it. You should end up with something like this:



As we keep adding Gizmos to the scene, it's going to get hard to tell one from the other. First, we can change the color, but the other thing we should do is turn on a label so we can see the name in the editor. Click on the icon to the left of the GameObject's name in the Inspector panel, and from here you can pick a label or other icon to show in the Scene editor.

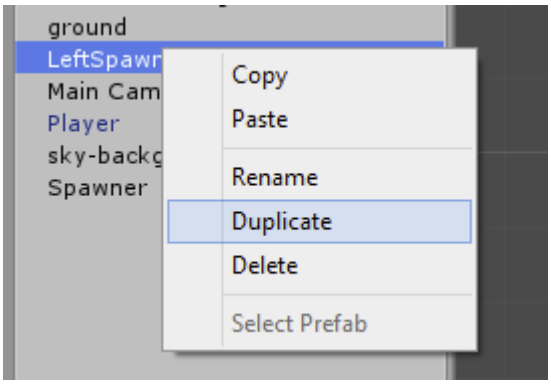


Here you see the label for my `LeftSpawnZone`. Now we just want to customize it a little bit more. Adding the Gizmo doesn't really do much outside of helping us visualize what the spawn zone actually looks like. Here is how I ended up tweaking mine:

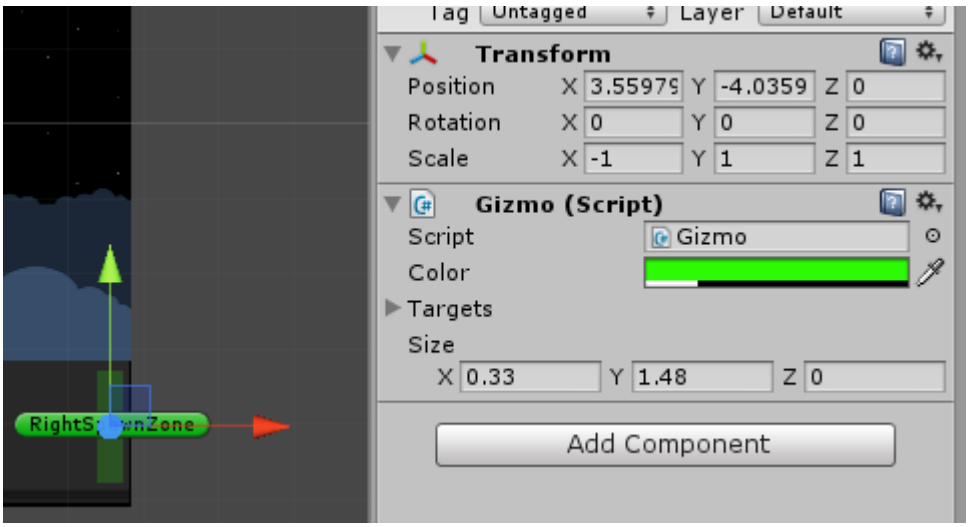


Make sure you put it off to the side of the background. I also changed the size to fit within the ground image.

Now we should do the same thing to create the other spawn zone. We can save ourselves some work by simply duplicating the existing `LeftSpawnZone` and renaming it. Right-click on it in the Hierarchy view and select Duplicate.

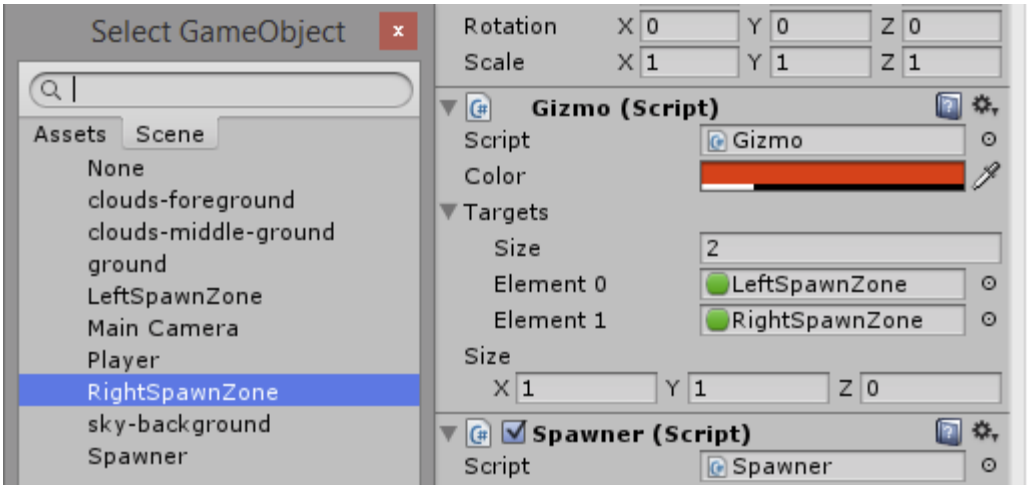


Now rename it to `RightSpawnZone` and move it to the other side of the scene. You'll also want to make one additional modification: change the `Scale X` to `-1`. While this won't make a visual change to the spawn zone, it will play a more important role, which I'll get into shortly.

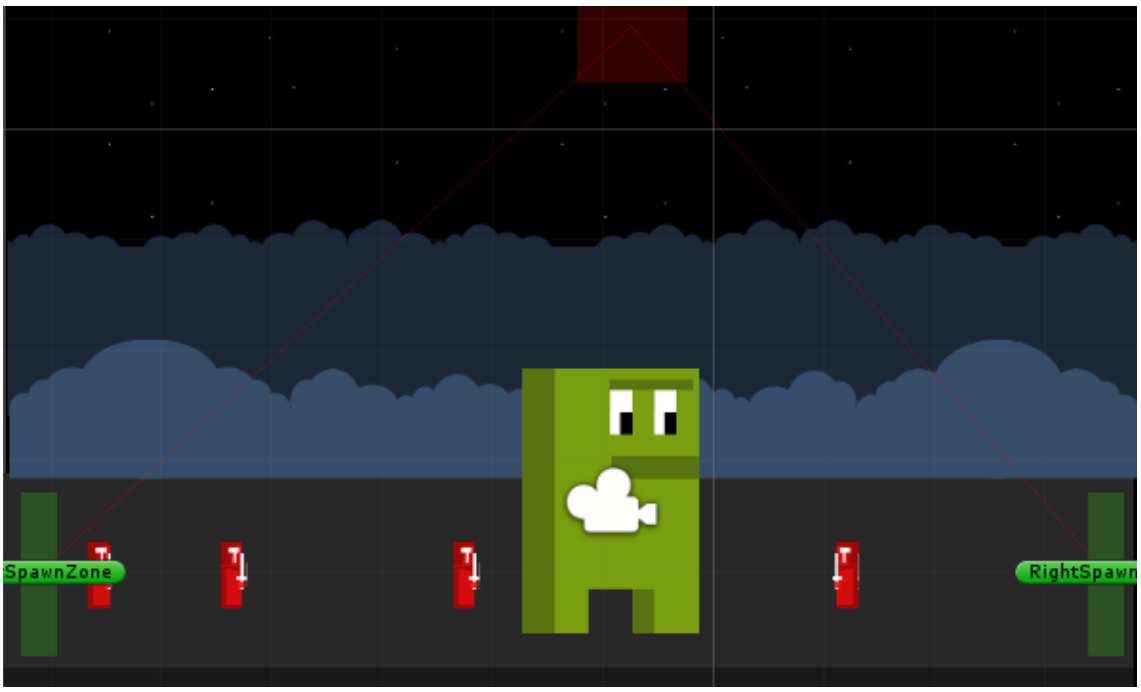


You should also make sure both spawn zones have their `Z` position set to `0`. It's a good habit to get into when working in the 2D tools to make sure GameObjects don't disappear behind another 2D object. So, now with the `Scale X` set to `-1`, any GameObject spawned from this will get that scale factor and not only face to the left but also walk to the left, assuming they have our `MoveForward` script.

At this point, we should have everything we need to connect the `Spawner` to the spawn zones. Select it in the Hierarchy panel, and then in the Inspector panel, add both spawn zones to the Gizmo component's target list.



Once you have done that, you should see two red lines being drawn from the Spawner to the spawn zones to help you visualize the connection. This is part of the Gizmo code we added and is now built into the script. Run the game and you'll see `Knights` being spawned to the new spawn zones, and also walking in the correct direction.

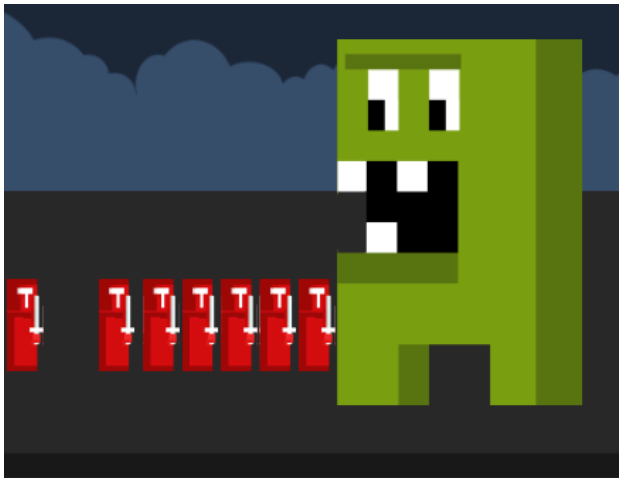


We now have everything connected to spawn bad guys, so now all that is left is a way to kill them.

Collision Detection

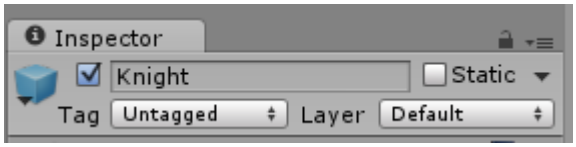
When we talk about collision detection, we really tend to focus on two types: making things solid and detecting when two objects collide. We have already solved the first problem thanks to Unity's built-in support for the Box2D physics engine. As far as we are concerned, everything in the game now knows how to collide with each other. The `Player` doesn't fall through the floor, and it can collide with the `Knights`. The big issue we have now is being able to block the Player from moving off the screen, and what to do when he collides with a bad guy. To get started, let's solve the bad guy issue first, because at this stage of the game, we can't really move around with all the Knights spawning everywhere.

If you collide into too many Knights, you'll see they kind of bunch up and form a line.

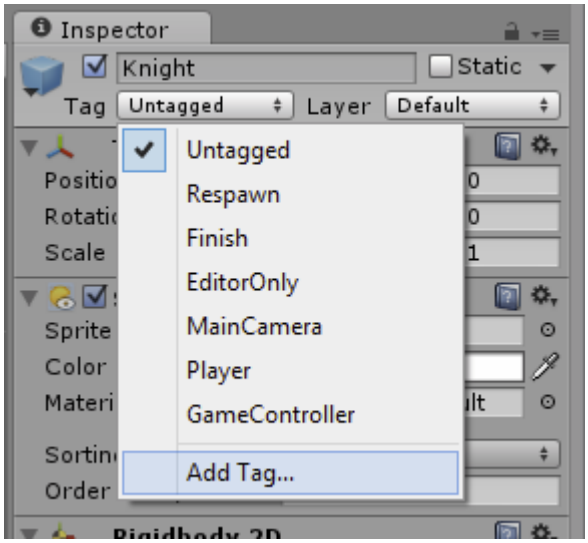


This happens because they are all colliding with each other and pushing the other ones back. We really don't want this to happen; Knights shouldn't collide with each other. We can quickly fix this by taking advantage of Unity's built-in Layers and Tags.

If you open up the Knight prefab, there are two drop-down menus below the name field.



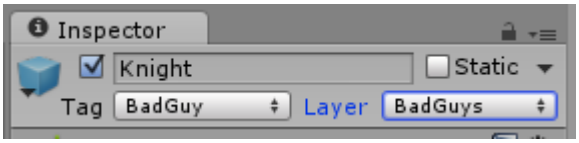
Let's start with Tag. Click on the drop-down menu and select Add Tag.



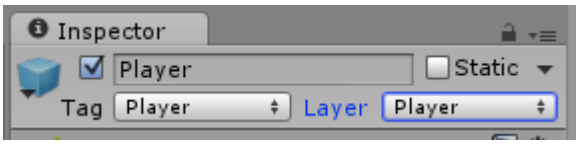
From here we are going to create a few tags for our game. Make one for the `Player` and one for the `BadGuys`.

While we are in here, we should also create the layers we need too.

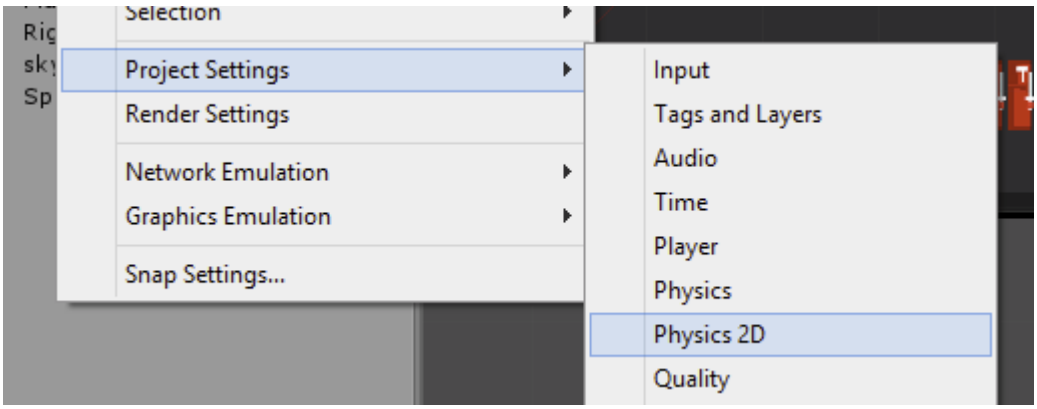
Now reselect the `Knight` prefab and set it to the `BagGuys` tag and layer.



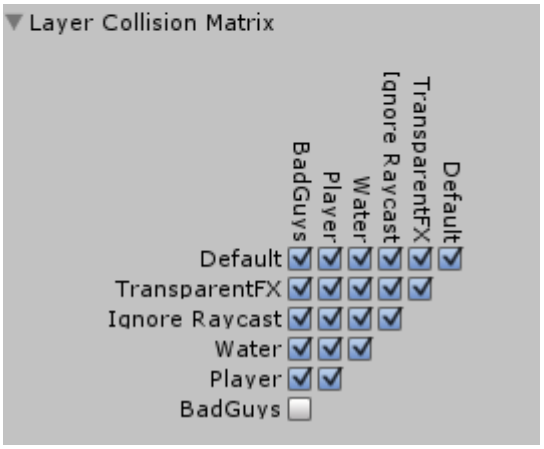
Next we should set the `Player` to the `Player` tag and layer.



Remember that, since we are doing these in the prefab of each GameObject, it will also be set in any new or existing instances of it in our game Scene. Now we need to set up some collision relationships. Open the Physics 2D settings from the Project Settings menu under Edit.



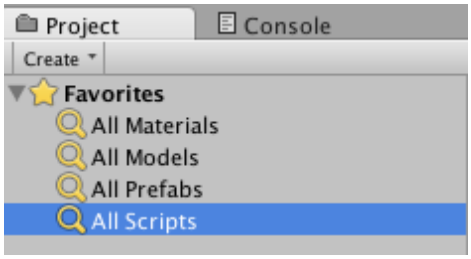
You will now see the Layer Collision Matrix, among other settings, that represent the Physics 2D settings of your game. We are going to want to uncheck the BadGuys layer, which will disable them from colliding with each other.



Take note that the Player layer is still able to collide with the BadGuys layer. This is what will allow us to still test for collision between the Player and the Knights. Now save the project and we are going to start building the foundation for having the Player attack the Knights.

Before we can have each of our main GameObjects attack one other, we need to add the notion of health and what happens when it runs out. To do this we are going to build a simple health script for both of our GameObjects. To get started, go into our Scripts folder in the project window and create a new script called `Health`.

It's also a good time to organize your project. Make sure all of your scripts are in the Scripts folder in our project. You'll notice that, over time when you add scripts to GameObjects in the Inspector panel, they are automatically put in the root of the Assets folder. You can easily find all the scripts in your project by clicking on the All Scripts Favorites filter. Simply drag the ones not in the correct folder over.



In the `Health` script we want to add the following properties:

```
public int maxHealth = 10;
public int health = 10;
```

This allows us to publically define the `maxHealth` of the GameObject. In the `Start` method, add the following:

```
health = maxHealth;
```

Now when the script runs for the first time, it will automatically set the health property to the `maxHealth`. Next we need to add two more methods:

```
public void TakeDamage(int value)
```

```
{
    health -= value;

    if (health <= 0)
    {
        OnKill();
    }
}

void OnKill()
{
    Destroy(gameObject);
}
```

We can now use the new `TakeDamage` method to remove `health` from a GameObject, and when the health reaches `0`, it will automatically be destroyed. Add this script to our `Knight` prefab. Next we are going to create a new script called `Attack`. Add the following properties to it:

```
public int attackValue = 1;
public float attackDelay = 1f;
public string targetTag;
private bool canAttack;
```

These properties should be self-explanatory, but basically we set up the attack value, the delay between attacks, a target tag value (which I'll cover in more detail when I set up the component), and whether the GameObject can attack. Now modify the `Start` method to look like this:

```
if (attackValue <= 0)
    canAttack = false;
else
    StartCoroutine(OnAttack());
```

Basically, we test to see if the GameObject has an attack value. If it does, we start the process of allowing the GameObject to actually attack by calling `OnAttack`. This sets the `canAttack` value to `true`. If the GameObject doesn't have an attack value, there is no point in starting the `OnAttack` coroutine.

Next we want to know when to actually perform an attack. To do this, we can take advantage of a method that is called when a 2D collisions happens called `OnCollision2DStay`. Let's add it to our script, and I'll explain how it works:

```
void OnCollisionStay2D(Collision2D c)
{
    if (c.gameObject.tag == targetTag)
    {
        if (canAttack)
            TestAttack(c.gameObject);
    }
}
```

In Unity, there are several collision calls we can tap into. Here we are using `OnCollisionStay2D`, but you can also take advantage of `OnCollisionEnter2D` to capture the first point of contact or

`OnCollisionExit2D` when two object stop touching each other. Since our game will have the GameObjects right next to each other for continuous attacking, we will use `OnCollisionStay2D`.

So, in this method, we simply check that the colliding object's tag is the same as one set on the `targetTag` value of the component set in the inspector. If there is a match, we test that `canAttack` is set to `true`. Then we test if the attack is valid. Let's add that method next:

```
void TestAttack(GameObject target)
{
    if (transform.localScale.x == 1)
    {
        if (target.transform.position.x > transform.position.x)
            AttackTarget(target);
    }
    else
    {
        if (target.transform.position.x < transform.position.x)
            AttackTarget(target);
    }
    canAttack = false;
}
```

Now, in this method, we are basically testing the direction the GameObject is facing and what side the colliding GameObject is on to make sure we can attack. If it's valid, we call `AttackTarget`. At the end, we reset the `canAttack` flag to `false`. Now let's add the `AttackTarget` method:

```
void AttackTarget(GameObject target)
{
    var healthComponent = target.GetComponent<Health>();
    if (healthComponent)
        healthComponent.TakeDamage(attackValue);
}
```

Here we test to make sure the target we want to attack has a `healthComponent` attached to it. If it does, we call `TakeDamage` and pass in the `attackValue`. Now all we need to do is add the last method, which is `OnAttack`:

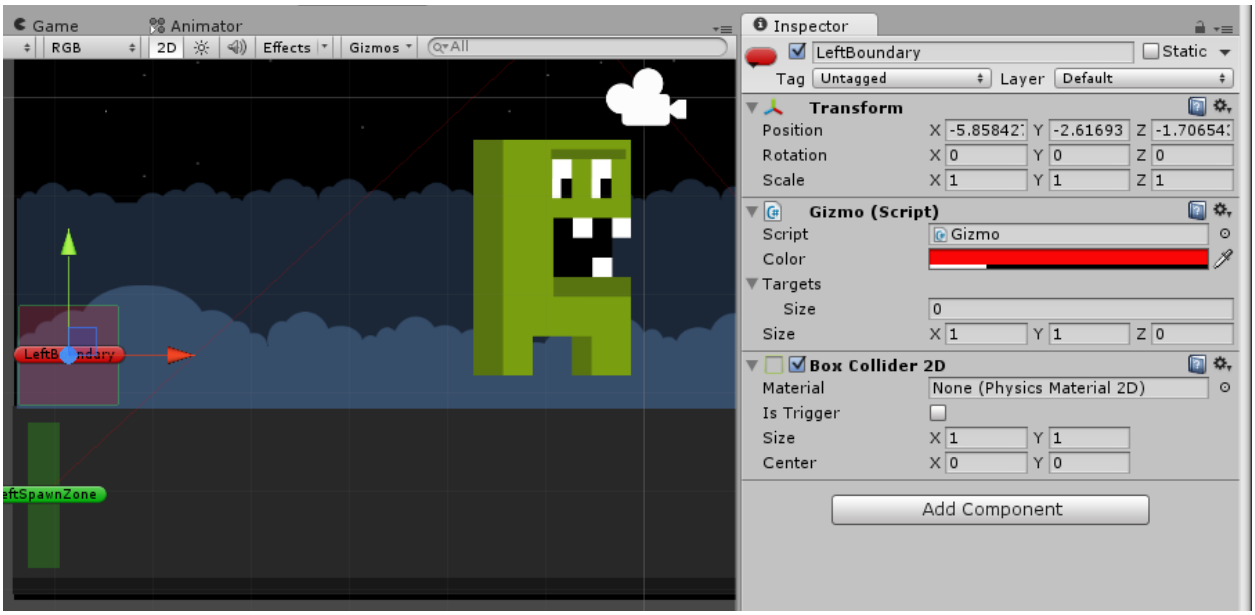
```
IEnumerator OnAttack()
{
    yield return new WaitForSeconds(attackDelay);
    canAttack = true;
    StartCoroutine(OnAttack());
}
```

As you can see, this is what handles our attack delay. We simply use `WaitForSeconds` to determine when the next attack can happen and set the `canAttack` value to `true`. This means that when we have a collision, if the `canAttack` value is `true`, the GameObject will attack. After that, we restart the `Coroutine` again.

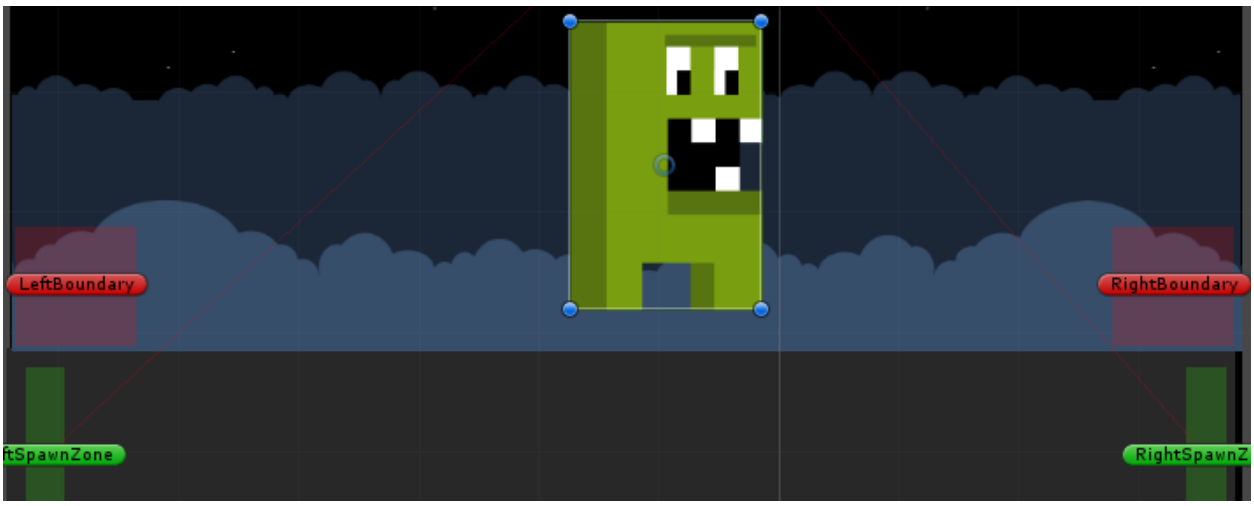
Add this script to our `Player` prefab so he can attack the `Knights`. Before we run the game, we should make one minor modification to the script on the Player; we need to tell it what tag is valid to attack. In the Inspector panel, add `BadGuys` to the `Target Tag` field.

You can also see I made the `Attack Value 5` and the `Attack Delay` to `0.2`, so when I am testing, the `Player` will kill the bad guys quicker. Play the game and try to run into some `Knights` and you will see them disappear after a few hits.

At this point, you may have noticed that it's a lot easier to fall off the level. We can fix that quickly by adding some simple boundaries to keep the Player within the indented area. Create a new empty GameObject and add the `Gizmo` script to it. Call it `LeftBoundary`, give it a label, and add a Box Collider 2D component as well. Here you can see I moved mine to just above the LeftSpawnZone.



This will be just high enough that the Knights won't hit it, but it will stop the Player. Duplicate it, call it `RightBoundary`, and put it just above the `RightSpawnZone`.

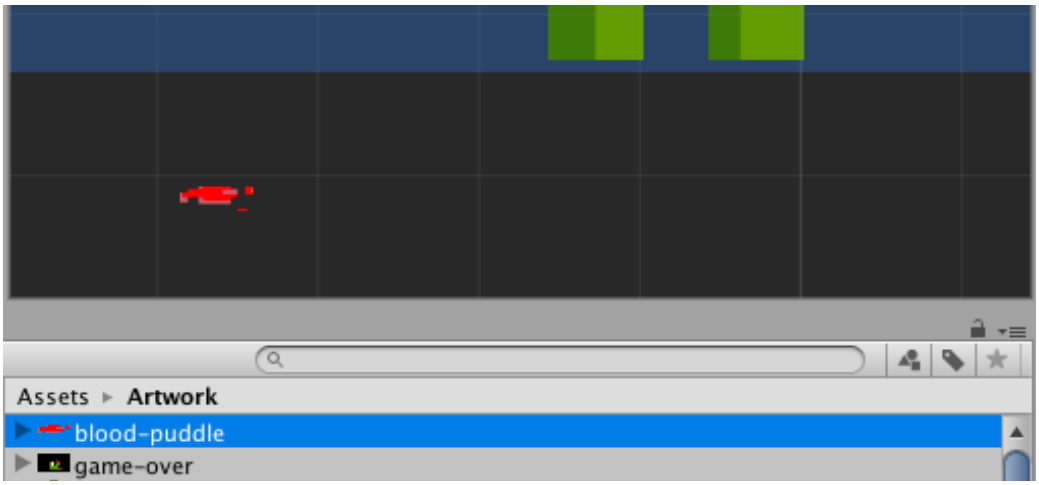


You may notice that the left and right boundaries abruptly stop the player. This doesn't look all that great. You can add the `GroundBounce` Physics Martial 2D we created earlier to the boundary's Box Collider 2D to give it a little more pushback when the player hits the edges of the screen.

Adding Effects

Right now our Knight's deaths are a little anticlimactic. At the very least we should leave some kind of indication that they were killed. To do this we are going to create a simple blood puddle that spawns on their death and fades away after a short delay.

To begin, go into the artwork folder and drag the blood puddle graphic over to the Game Scene.



For right now we will just set it up here to do some testing before fully connecting it up to the knight's death event. Once you have the blood puddle setup, rename it to `BloodPuddle` in the Hierarchy tab, and then drag it over to our Prefab folder. Now we can start configuring it. We'll need to make sure that its `Order in Layer` value is set to `-1`.

This will insure that it renders behind the player and bad guys who are set to the default `Order in Layer` of `0`. Now let's create a new script called `FadeAway`. This will allows use to not only fade a GameObject until it is completely transparent but have it destroy itself when it's no longer visible to cut

down on memory usage. Once you have the script open add the following property and modify the `Start` method like so:

```
public float delay = 2.0f;

void Start () {
	StartCoroutine(FadeTo(0.0f, 1.0f));
}
```

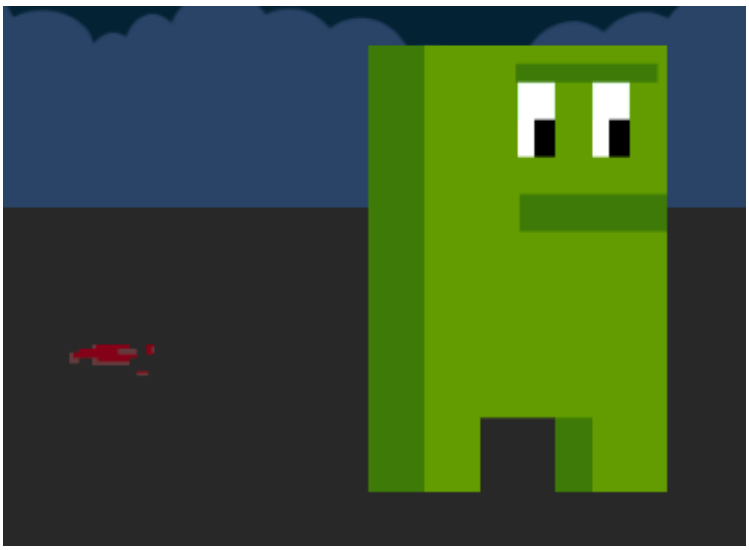
Here you can see we are creating a public delay value, which we will be able to tweak in the inspector panel later on but most importantly we are creating a new coroutine to handle the fading. Let's create that method now:

```
IEnumerator FadeTo(float aValue, float aTime)
{
	yield return new WaitForSeconds(delay);

	float alpha = transform.renderer.material.color.a;
	for (float t = 0.0f; t <= 1.0f; t += Time.deltaTime / aTime)
	{
		Color newColor = new Color(1, 1, 1,
Mathf.Lerp(alpha,aValue,t));
		transform.renderer.material.color = newColor;

		if(newColor.a <= 0.05)
			Destroy(gameObject);
		yield return null;
	}

}
```

This should look familiar since it's similar to how we created our spawner. First we had a delay by calling `WaitForSeconds` and pass in our public `delay` property. From there we get the current material's alpha value. The renderer handles displaying the 2D sprite and has a material with a color property attached to it. Once we have the value of the alpha, we can begin to modify it with a new math utility method called `Lerp`. `Math.Lerp` allows us to interpolate between two values, in this case our current alpha and our target alpha value which is going to eventually be `0` over a set amount of time. From there we simply reassign the new alpha value to the material color and test that it is close to being completely transparent before we call `Destroy`.

You may notice that we don't wait for the alpha value to be completely 0. In fact we could probably be more aggressive with this condition and remove it when the value is 0.1. Due to the way Lerp modifies the value it would never really reach 0, or if it did it would take a very long time. Since this last step is all about optimizing our code, we want to make sure we immediately remove it once the user can no longer see it.

Attach this script to the `BloodPuddle` prefab if you haven't already done so. At this point you can run the game and you should see the blood puddle fade away.



Now we need to connect this up to our bad guys. We'll do this by modifying the `Health` script to accept a new reference to a GameObject to display when its parent is killed. Open up the `Health` script and add the following properties to the top:

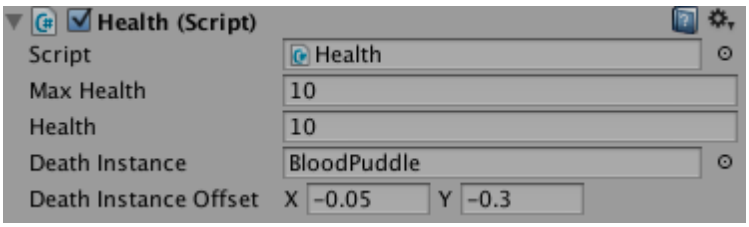
```
public GameObject deathInstance = null;
public Vector2 deathInstanceOffset = new Vector2(0,0);
```

Now we will need to modify the `OnKill` method to look like this:

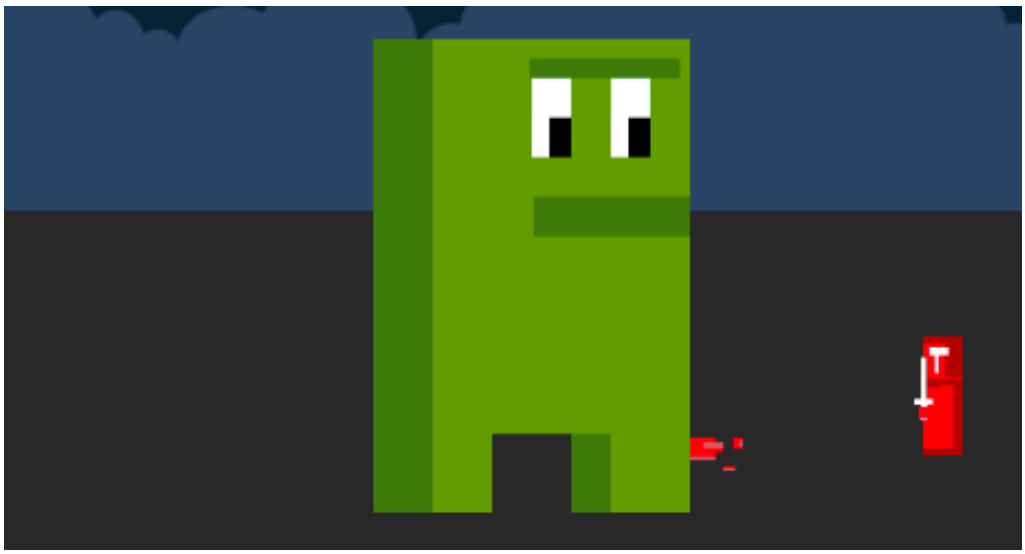
```
void OnKill()
{
      if (deathInstance) {
            var pos = gameObject.transform.position;
            GameObject clone = Instantiate (deathInstance, new
Vector3(pos.x + deathInstanceOffset.x, pos.y + deathInstanceOffset.y,
pos.z), Quaternion.identity) as GameObject;
      }
      Destroy(gameObject);
}
```

Here you can see we test to see if there is a `deathInstance` set on the component. If it is set, we spawn that instance when the GameObject was killed. If we don't set the `deathInstance`, it will simply destroy the GameObject like we did before. We also use `Quaternion.identity` constant to help make sure that the GameObject is aligned with the world's axes.

Now we are ready to remove blood puddle we used for testing from the Scene Editor and go over to the `Knight's` Prefab so we can finish setting this up. You should now see the option to add a `deathInstance` to the Health component on the Knight Prefab. Click on it and select the `BloodPuddle` Prefab. We will also need to add an offset to the `deathInstance` so that it spawns at the base of the `Knight` instead of floating in the air. For the `Knight` we will set the `X` to `-0.05` and the `Y` to `-0.3` like so:



Now let's run the game and kill a knight or two in order to make sure that this works correctly.



At this point we can start adding some extra features to the Player like a health bar and its own death effects.

Adding A Health Bar

Up until this point our player has more or less been invincible. We are going to allow the bad guys to attack him and we'll show the player their status with a health bar at the top of the screen. To get started we are going to need to create a new script called `HealthBar` and add the following properties to it:

```
public Texture backgroundTexture;
public Texture foregroundTexture;
public Texture borderTexture;
public Vector2 offset = new Vector2();
public GameObject target;
private int barWidth;
private int barHeight;
private Health targetHealthComponent;
```

Basically were are going to layer three separate sprites to make up our health bar:

The `backgroundTexture` is what you see as you run out of life. The `foregroundTexture` is the health bar itself and the `borderTexture` is the overlay on top that makes it look nice. To do this we will set three different textures for each layer. We will also need a way to position the health bar on the screen, which is done via the `offset` property. To keep things simple we will align it to the upper left hand corner of the screen since that position is always constant across any screen size. And finally we have a `target`, which when pointed at a GameObject with a `Health` components can give us the values we need to render the health bar correctly.

Now let's set up our `Start` method which will figure out the dimensions of our background texture as well as get a reference to the health component on the target GameObject:

```
void Start(){
      barWidth = borderTexture.width;
      barHeight = borderTexture.height;
      targetHealthComponent = target.GetComponent<Health>();
}
```

In order to render the health bar on the screen we are going to take advantage of a special method called `OnGUI` that is reserve for drawing GUI elements on the top layer of the screen. Let's stub out that method and add in some logic to calculate the percentage of health remaining on the target GameObject:

```
void OnGUI () {
      var percent = ((double)targetHealthComponent.health /
(double)targetHealthComponent.maxHealth);
}
```

As you can see, we are simply using the `Health` script's `health` and `maxHealth` properties to come up with the percentage of health remaining on the target. Now we need to render out the health bar. To do this we are going to use another special utility class called `GUI` and its `DrawTexture` method. Add the following after where we calculate the `percent` in the `OnGUI` method:

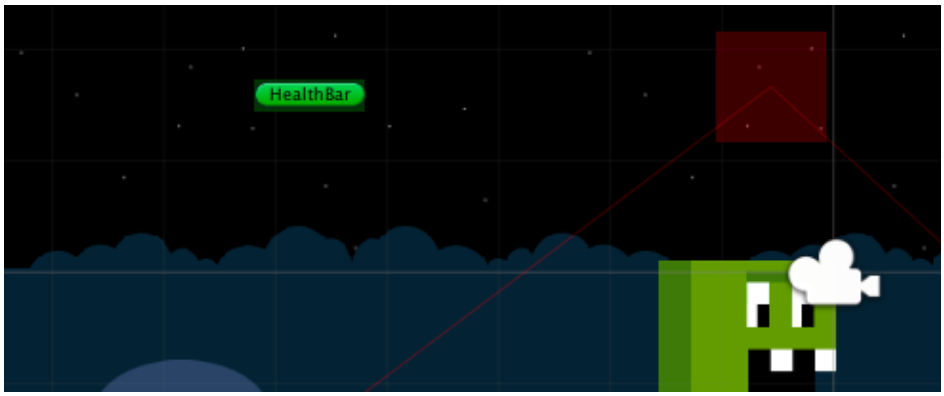
```
GUI.DrawTexture (new Rect (offset.x, offset.y, barWidth, barHeight),
backgroundTexture);
```

As you can see `GUI.DrawTexture` is going to allow us to draw a texture into the GUI layer of the screen, which sits on top of everything else. It requires a Rectangle that represents the final x, y, width and height of what will be drawn to the display as well as a reference to the actual texture itself.

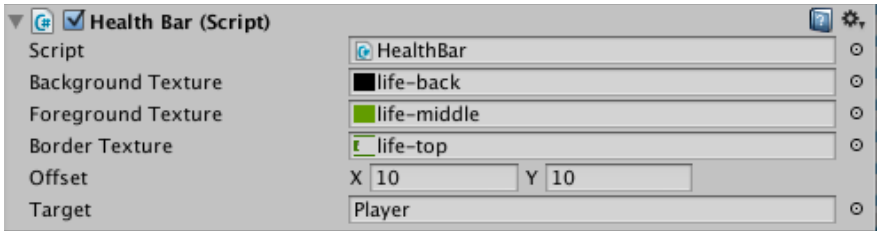
Let's add the next two layers after this one so they render properly:

```
GUI.DrawTexture (new Rect (offset.x, offset.y,
(int)System.Math.Round(barWidth * percent), barHeight),
foregroundTexture);


GUI.DrawTexture (new Rect (offset.x, offset.y, barWidth, barHeight),
borderTexture);
```

I did want to point out that you should notice that we are altering the `width` of the foreground layer by multiplying its `width` value by the `percent` we calculated earlier. This is a standard way of modifying the size of something like a health bar and will allow us to automatically change its `width` on each render based on the target's current health. Now it's time to test this out. To do this we will need to create a new empty GameObject and give it a label Health Bar:

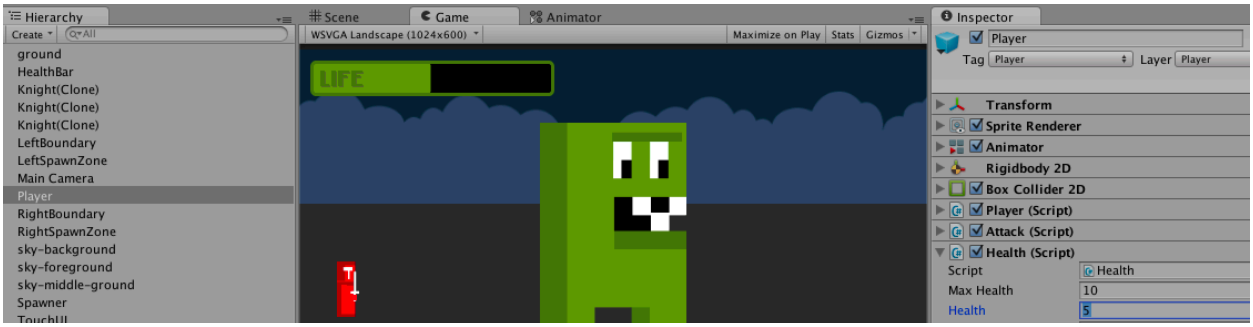


From here add the `HealthBar` script to the new GameObject and rename it to `HealthBar` so we can easily find it the hierarchy view later on if needed. It doesn't matter where you put this GameObject since it will automatically handle drawing the health bar on top of everything ignoring its parent GameObject's location in the scene. Also, make sure to add the `Player` in the scene as the `Target`.



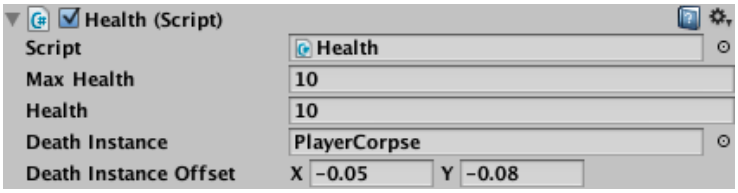
Now if we run the game, we should see a full health bar being rendered out on top of everything:

Right now the player isn't being attacked but we can still test out how the health bar rendering while the game is running. Simply select the `Player` while the playing and you can change the `health` to `5` and watch the health bar update itself:

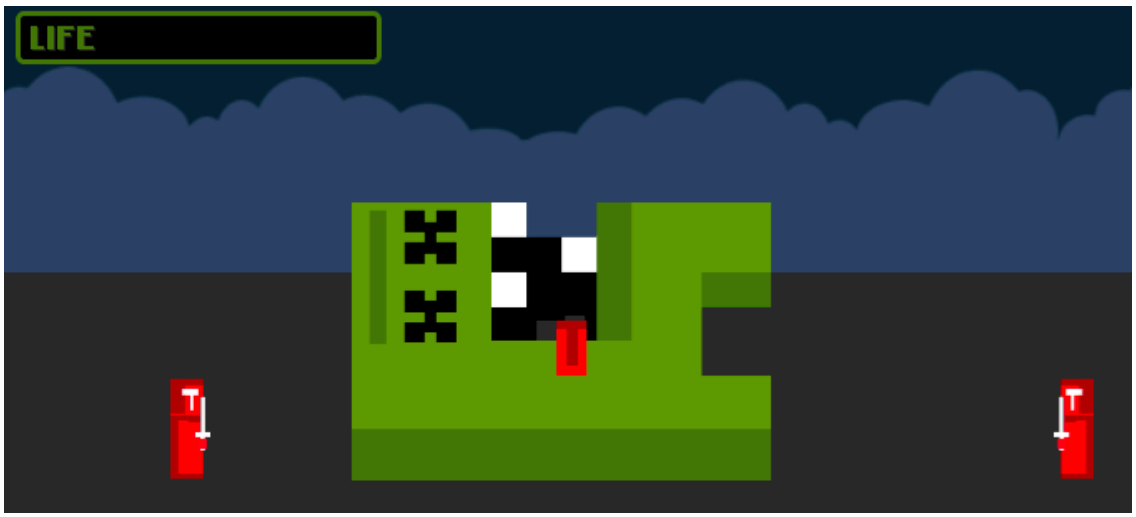


Now we are ready to make the bad guys attack the player. Let's stop the game and select our `Knight` in the Prefab folder. From there we want to add the `Attack` script to it in the inspector panel. We'll configure the `Attack` script to target the `Player` and leave everything else as is:

If we run the game again and don't move, the knights will start attacking the player and you should see the health go down. Then with that working the last thing we are going to want to do is give the player a death graphic similar to how the bad guys have the blood puddles. To do this we will need to create a prefab out of the `player-corpse` sprite in the Artwork folder. Once that is done, go to the `Player` and set the `Corpse` prefab as the `deathInstance` in the `Health` component panel of the Inspector.



You will also need to modify the offset `X` to `-0.05` and `Y` to `-.08` so the corpse apears like it is resting on the floor and not floating in the air when it is created after the player's death.



Everything is looking great and the game is finally coming together. We just need to create one more thing, some visual cue to the player on what the controls for the game are going to be.

Control UI

Our game has some basic UI but without any indication to the player what that may be, it's going to be a little jarring for then when they jump into the game. To help them out we are going to create a very simply overlay to give them an idea that they can move the Player to the left or the right depending on what side of the screen they click on. To get started create a new empty GameObject and call it `ControlsUI`.

It doesn't matter where you put this, we'll be drawing in the GUI layer so it will always show up on top of the game itself and not moving with the Player. Now let's create a new script on this GameObject called `ControlsUI`.

In this new script we are going to add the following properties:

```
public Texture leftArrow;
public Texture rightArrow;
public Vector2 offset = new Vector2(10, 20);
```

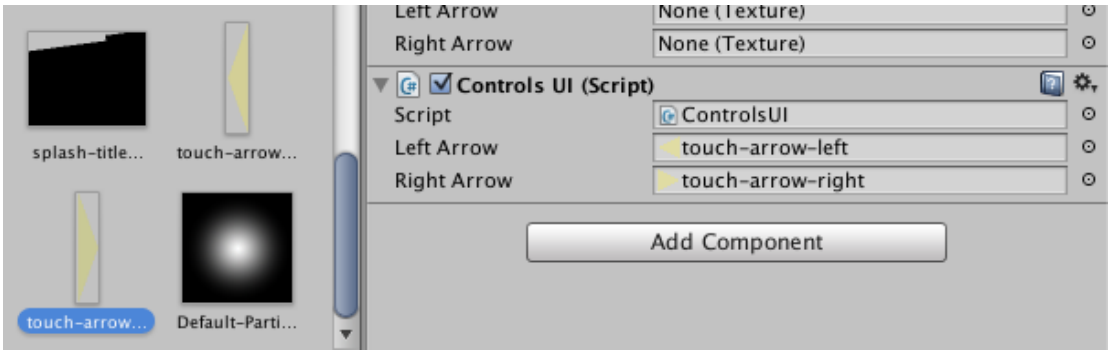
This will be the reference to the textures we want to show in the game and the offset from the edges of the screen we want to display them at. Now add the following method to the script:

```
void OnGUI () {

      if(leftArrow)
            GUI.DrawTexture (new Rect (offset.x, offset.y,
leftArrow.width, leftArrow.height), leftArrow);

      if(rightArrow)
```

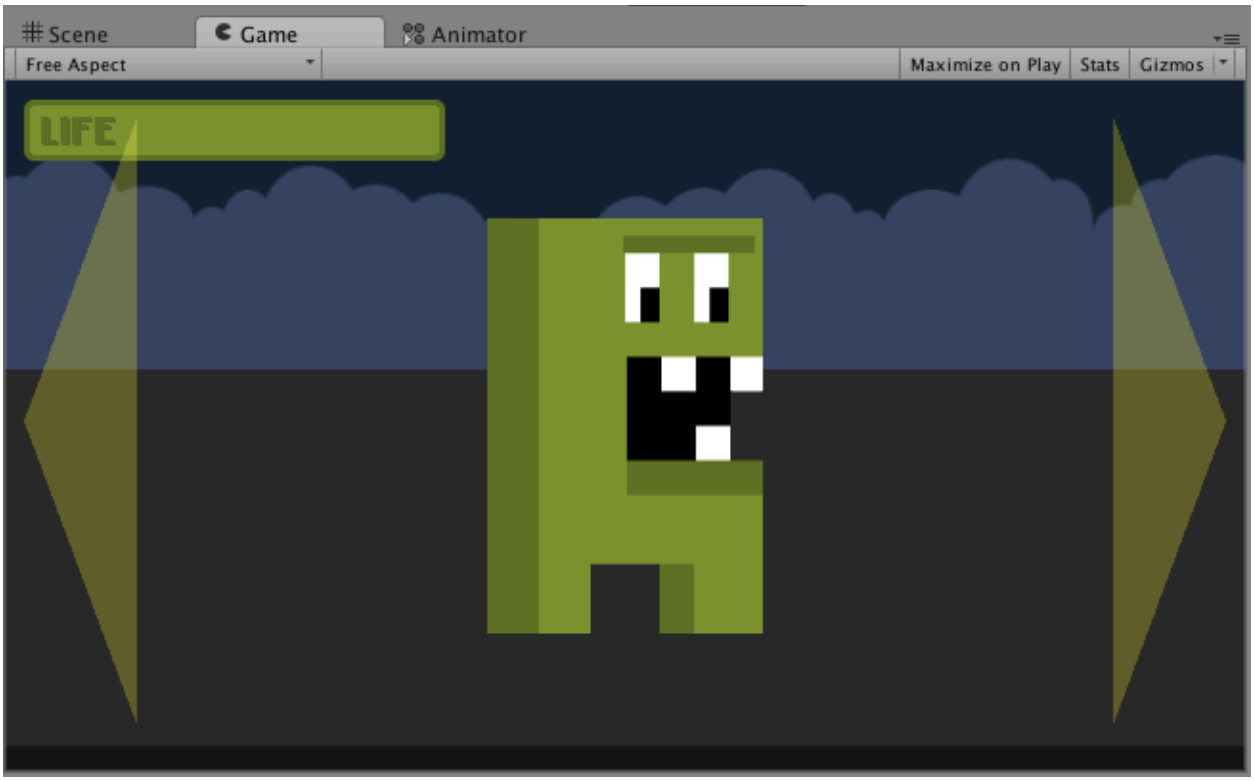
```
            GUI.DrawTexture (new Rect (Screen.width - rightArrow.width
- offset.x, offset.y, rightArrow.width, rightArrow.height),
rightArrow);

}
```

Here you see we are simply making sure that the `leftArrow` and `rightArrow` textures are set, and then we render them to the GUI layer. Take note that while the `leftArrow` is drawn at the offset position, we modify the `rightArrow` to subtract its own `width`, from the width of the screen and the `offset.x` to align it to the right side of the screen no matter what the resolution is.

Now you will need to set the `leftArrow` and `rightArrow` on the `ControlsUI` GameObject in the inspector window. You can find the `touch-arrow-left` and `touch-arrow-right` in the `Artwork` folder.



At this point if you run the game you will see the arrows:



Now we have everything in place to play a full game. We have our bad guys spawning; the player can attack and be attacked as well as the ability to display the health of the player. In the next section we will go over how to create different scenes such as the splash screen and game over screen to tie the entire game together.

Working With Scenes

Creating A Splash Screen

In this section we will cover one of the most important parts of your game, managing scenes. Unity games are broken up into individual scenes. This entire time we have been working in our main `Game` scene but now it's time to create a splash screen and a game over one as well then connect them up to the game.

To get started make sure you save the current scene we are working in the go to he File menu and select New Scene. We are going to call is `Splash` and save it to our `Scenes` folder. Now drag the following sprite textures into the scene:

- splash-background
- splash-monster
- splash-start-text
- splash-title
- splash-title-mask

From here we are going to want to change the coordinates the sprites in the scene by modifying their `X`, `Y` and `Order in Layer` values to the following:

File Name	X	Y	Order in Layer
Splash-background	0	0	-4
Splash-monster	1	0	-3
Splash-start-text	-2	2	1
Splash-title	0.45	-1.6	0
Splash-title-mask	0.4	-3.8	-2

The last thing we want to do is modify the Camera's `Size` property to `2.5` and make sure we set the `Background` to `Black`. Once you do this you will see the following in the Scene Editor.

As you can see, Unity's editor is great for laying out 2D graphics to build basic screens and menus for your game. This a very simple splash screen of course but I wanted to show you how I build these up by layering sprites on top of each other. When you build splash screens like this, it's easier to animate each of these layers as the player moves from scene to scene. If you look at the `splash-title-mask`, it's actually designed to cover the entire background so you can use it to hide everything then animate it down to reveal the splash screen artwork.

We are not going to complicate this with animation so we'll just need to add a little bit of flair to direct the player to what they should do in order to start the game. Let's create a new script called `Blink`. This script is simply going to modify the alpha of a GameObject to give the appearance that it is blinking to catch the player's attention. To get started we'll add the following property to the script:

```
public float delay = .5f;
```

Now we will need to modify the `Start` method:

```
void Start () {
      StartCoroutine(OnBlink());
}
```

Finally we will add our `OnBlink` method below:

```
IEnumerator OnBlink()
{
```

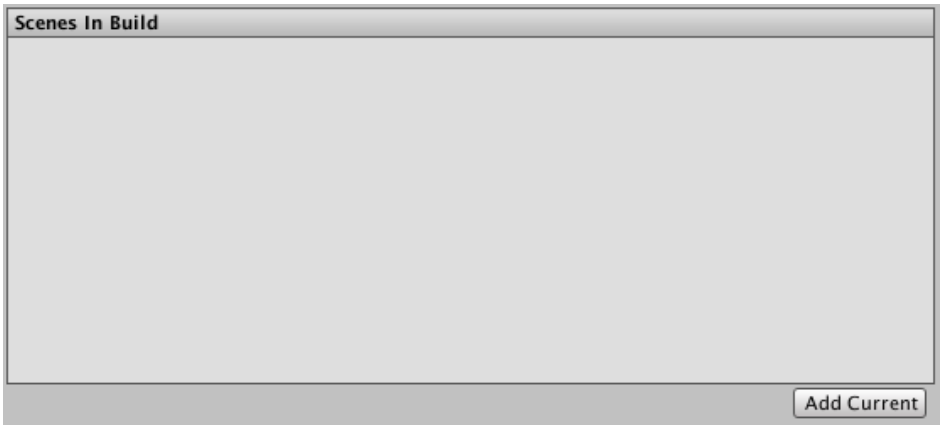
```
        yield return new WaitForSeconds(delay);

        float alpha = transform.renderer.material.color.a;

        var newAlpha = 0f;

        if (alpha != 1) {
              newAlpha = 1f;
        }

        transform.renderer.material.color = new Color(1, 1, 1, newAlpha);

        StartCoroutine(OnBlink());
    }
```

At this point this should be very familiar to you. It's the same concept we used in the `Fade` script and we have been taking advantage of `StartCoroutine` and `WaitFOrSeconds` throughout the book. Now all we need to do is attach this script to our `splash-start-text` GameObject and test it out.

While it's nothing fancy, it is enough to give the splash screen a little bit of life. Now let's talk about how we'll move from one scene to another.

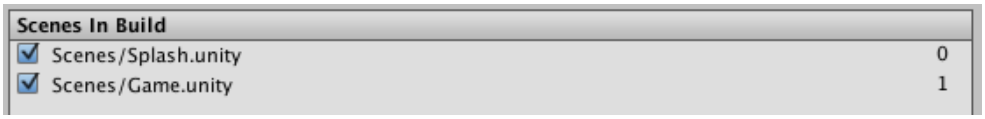
Moving Between Scenes

Moving between scenes in Unity is relatively easy. We can take advantage of a method on the `Application` class called `LoadLevel`. We can simply pass in the name of the scene itself to the `LoadLevel` method. The only thing to keep in mind is that your scene has to be added to the publishing settings of the game or it won't work. To get started, let's run through adding both of our scenes to the publishing tab. Start with the current `Splash` scene. Go to the File menu and select the Build Settings. From here you will see the following:



This is the area where you can manage what scenes are part of your game's build. At the bottom of the area on the right hand side is a button called Add Current. Click that and the current scene you are working in, which should be `Splash`, will be added to the list.

Now close this panel and open up our `Game` scene then add it to the list as well. You should see that the `Splash` Scene is the main one with an `ID` of `0` and that `Game` is now the next one with an `ID` of `1`.



Now that we have both of our scenes added to the project build settings we can work on switching between them. Go back into the `Splash` scene and let's create a new script called `ClickToContinue`. In it we are going to add the following properties:

```
public string scene;
private bool loadLock = false;
```

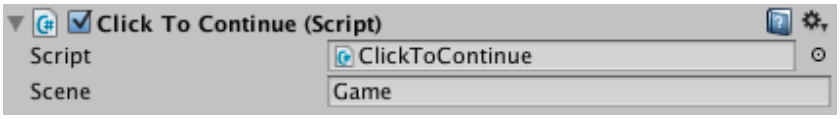
These two properties will allow us to set the name of the scene we'll navigate to and set a lock flag so that we don't keep calling the switch screen logic over and over again if the player keeps clicking. This is very important because if you have some kind of animation that needs to happen before the transition you don't want to that to keep being called by accident. Now let's modify the `Update` method to look like this:

```
void Update () {
	if(Input.GetMouseButtonDown(0) && !loadLock)
	{
		LoadScene();
	}
}
```

Earlier on in the book we used the `Input` class to get the mouse button being pressed. Here you can see we are doing the same thing again but we also test that the `loadLock` flag hasn't been set. Now we need to add our `LoadScene` method:

```
void LoadScene()
{
	if (scene == null)
		return;

	loadLock = true;
	Application.LoadLevel(scene);
}
```

Here we make sure that a scene has been defined, and then we set the `loadLock` to `true` so we can safely call the `Application.LoadLevel` method. Now in order to actually use this we will need to attach it to a GameObject. Click on our `splash-start-text` GameObject in the Hierarchy tab and add the `ClickToContinue` script to it. We'll also need to add the name of the scene to go to, so in the inspector panel, type in `Game` into the `Scene` value.



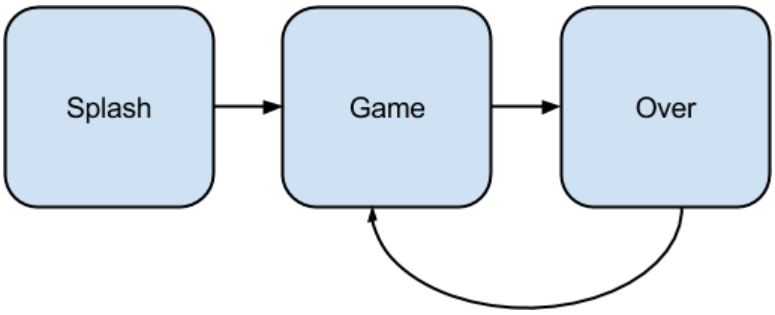
At this point you should be able to run the game and test that you can go from the `Splash` scene to the `Game` scene by clicking anywhere on in the screen. If it doesn't work, chances are good that you don't have the scene added to the build settings. This call will fail silently so make sure you have the scenes correctly added to the Build Settings window and you defined it on the component.

Creating A Game Over Screen

So now that we have a `Splash` scene that can transition into the `Game` scene we are ready to add in a game over screen and allow the player to restart the game. To do this we are going to need to make a new scene called `Over`. Once you have it created, drag the `game-over` sprite onto the scene and position its `X` and `Y` values to `0`,`0`. We'll also want to modify the camera's `Size` to be `2.5` and the `Background` to `Black`:

The last thing we are gong to want to do here is drag over the `splash-start-text` sprite, and attach the `Blink` and `ClickToContinue` scripts to it. Position the splash-start at `X` value of `0` and `Y` value of `-2.3` then set the `Scene` property to `Game` for the `ClickToContinue` component. Here is what the scene should look like when you run it:

You'll notice that we don't set this to go back to the `Splash` scene. We want our player to jump right back into the action so no need to make them click twice to restart the game. The `Splash` scene is really only for the first time you enter the game, after that the loop should be like this:



We are all done here, but before we can see the `Over` scene we are going to need to add a way of getting to it when the player dies. Luckily we have an easy way to do this without writing any new code in our game. Open up the `Game` scene and I'll show you how we can reuse some of the same code we already wrote to add a simple death animation for the player and link it to the `Over` scene.

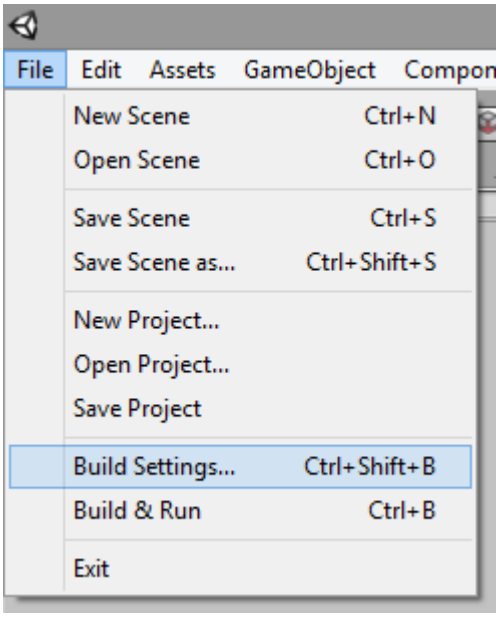
Once you are in the `Game` scene go to our `PlayerCorpse` prefab and attach the `ClickToContinue` script to it. Make sure that you also set component's `Scene` value to `Over`. Now when the player dies it will spawn a corpse and if the player clicks anywhere, it will go to the `Over` scene. Take a second to test out that it works.

Once the player dies and spawns the corpse you will be taken to the `Over` screen and from there it's just another click for the player to start to a new game. See how easy that was? Sure, it's not perfect and in an ideal world we would want to show the player some text but you could easily create a new script that simply renders the `splash-start-text` or some kind of click to continue text to render on the GUI layer. Or even better, you can add a timer to make the player automatically transition into the game over screen after a short delay.

At this point we have created a fully working game. And while it's not going to win any awards in its current form you have learned enough to move onto the final stage of Unity development which is publishing.

Publishing

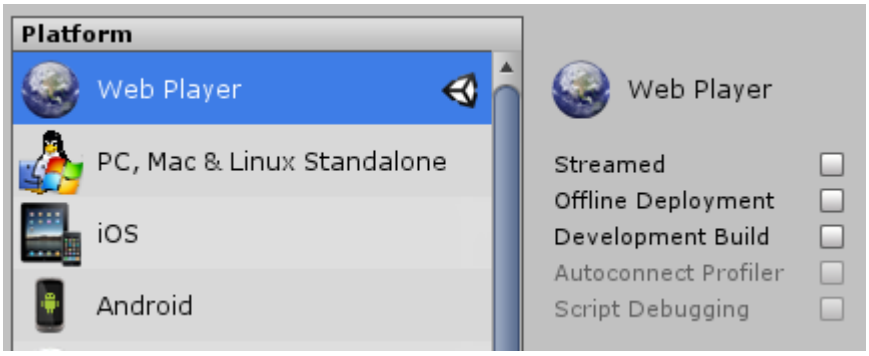
One of the biggest advantages of Unity is its ability to publish to multiple platforms. In this section we are going to do a quick overview of how to publish from Unity on different platforms. There are lots of resources out there that cover this process in more detail but this high level overview will give you the basic working knowledge of where to gets started. To access the output options, open the Build Settings from the file menu.



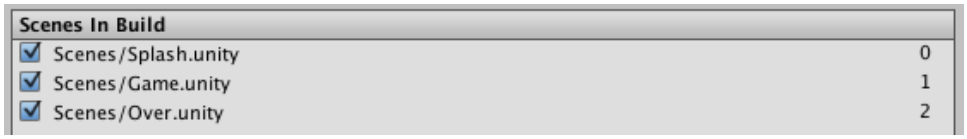
From there you will be presented with all the options that I will go over next.

Web

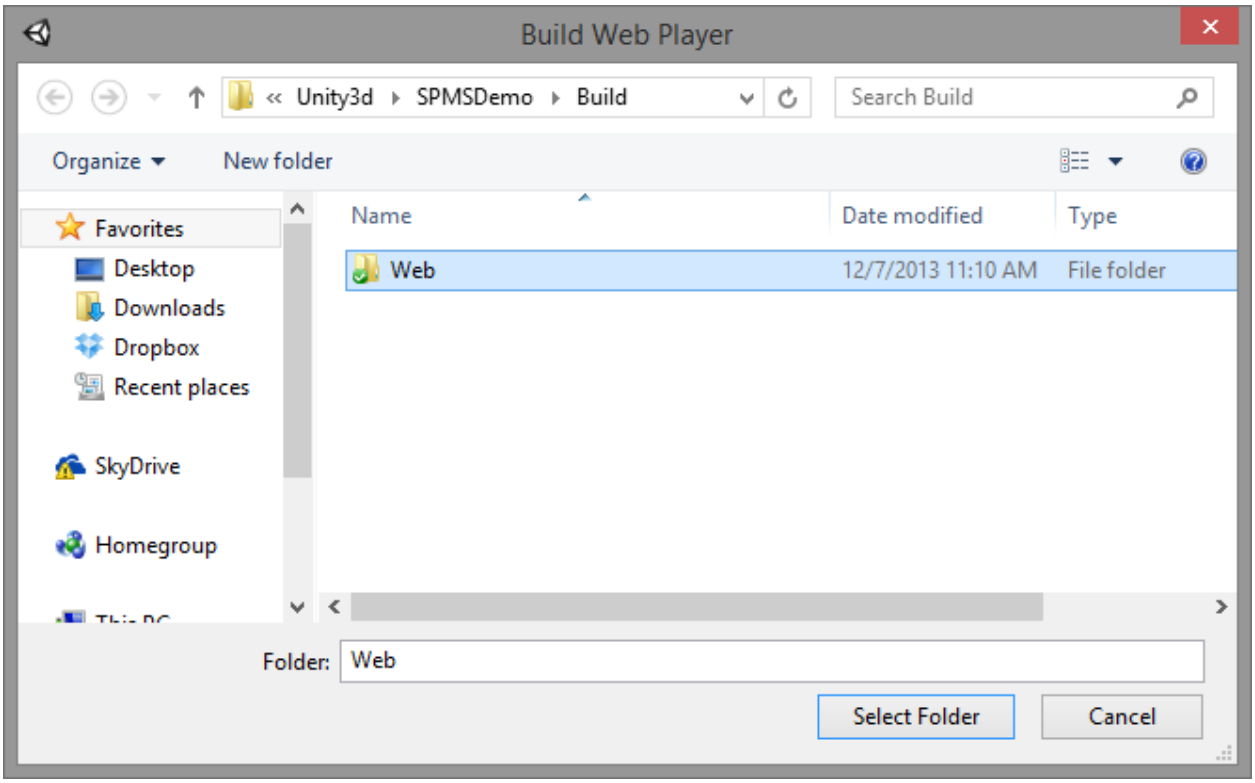
Out of all the options, Web Player is the easiest of the Unity output options. Simply select Web Player from the Platform menu:



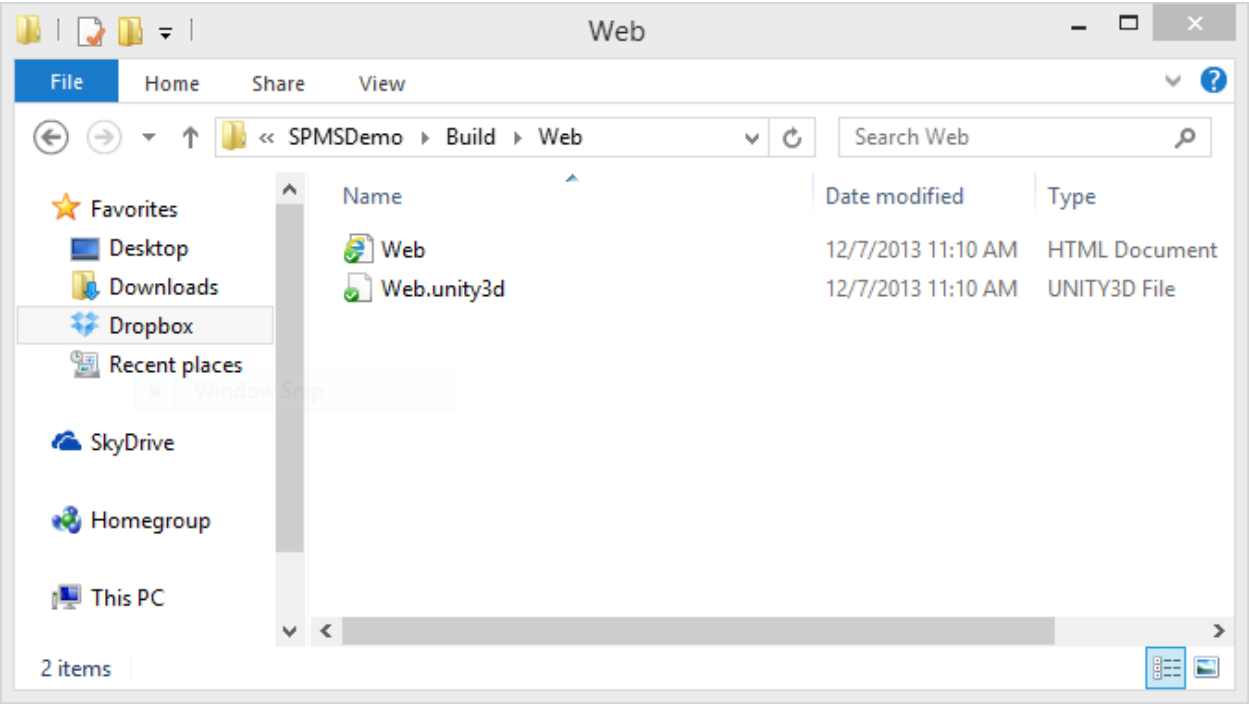
You will also want to make sure you include the current game scene so that it becomes the default scene in your outputted project. At this point you should have all three scenes in this menu:



From here you can click on the Build button, which will give you an option to save out the project. I usually create a `Build` folder in my project and organize them by target platform; in this case I save out to a `Web` folder:

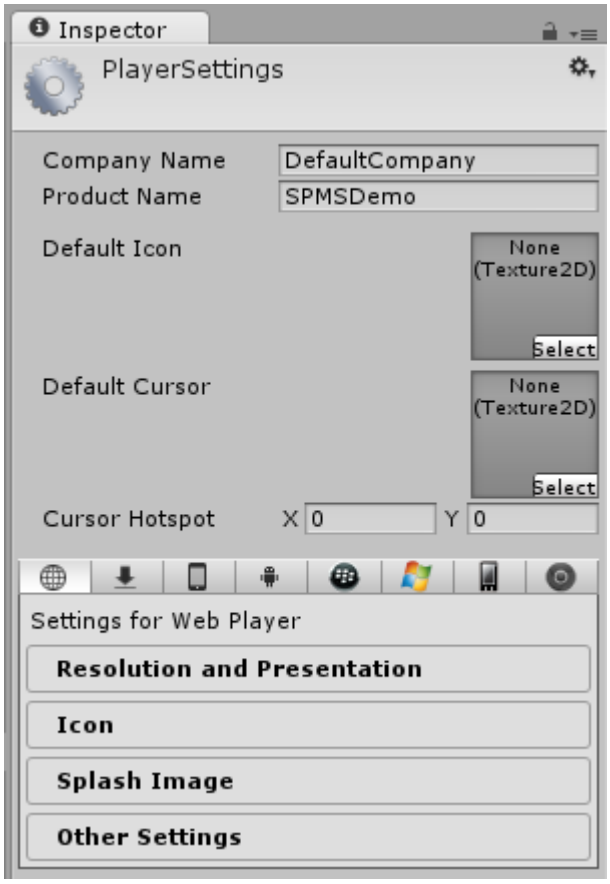


Once Unity creates the build you will see a standard `Web.html` file and the project file itself:

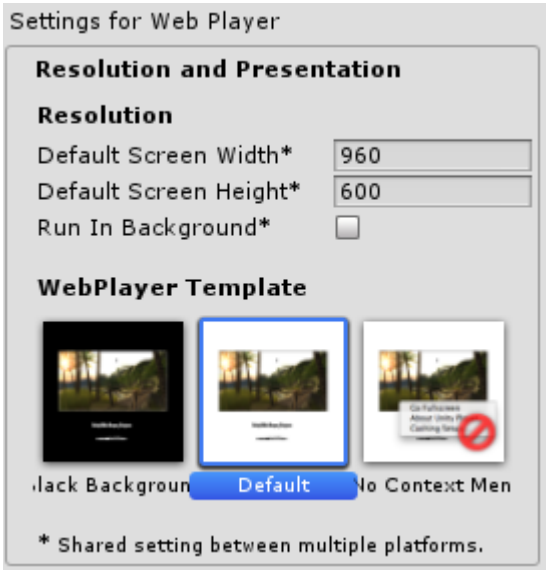


From here if you open the `Web.html` file in your browser, you should be able to play the game, assuming you have the plugin installed. From there you can simply upload this to a server or game portal that supports Unity and let others play your game online.

The last thing I wanted to point out was that if you click on the Player Settings button, which works for any platform you output to, the inspector panel will open up and provide additional export options for you:

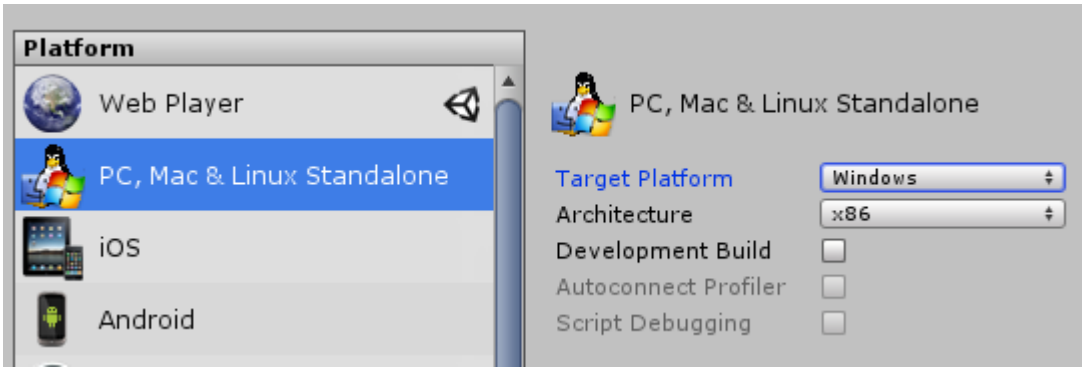


You may want to dig deeper into the Resolution and Presentation tab where you can configure the game's embed resolution, if it runs in the background and what the page template is set to.

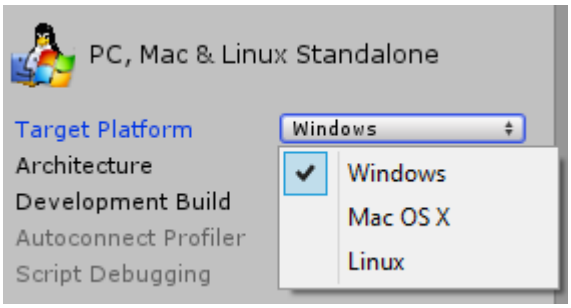


Desktop

Desktop is the next platform that is incredibly easy to build for. Switch over to the desktop option in the Platform menu.

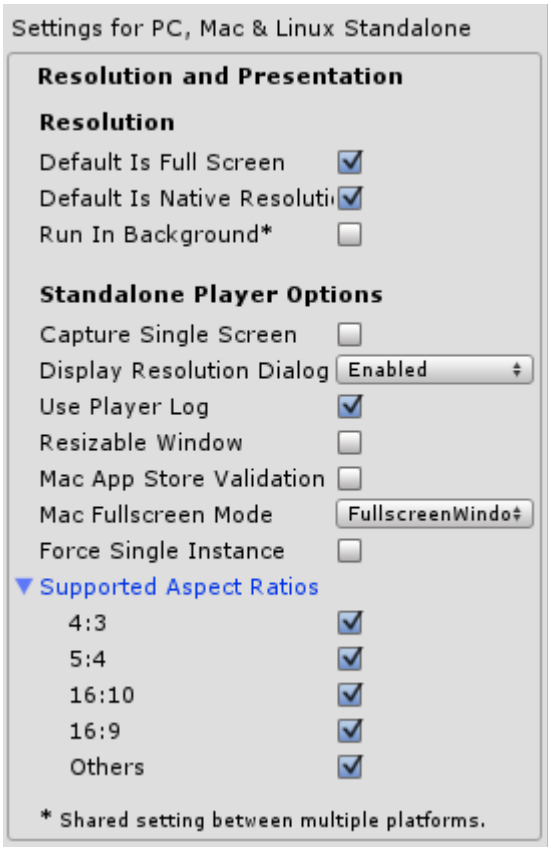


As you can see, you now have the ability to choose your platform from the drop down which includes Windows, Mac and Linux:



Similar to how the Web output worked, once you hit Build you will be asked to save out the game and again I create a new folder called `Desktop` in my `Build` directory. From there Unity will generate out an executable based on your targeted platform.

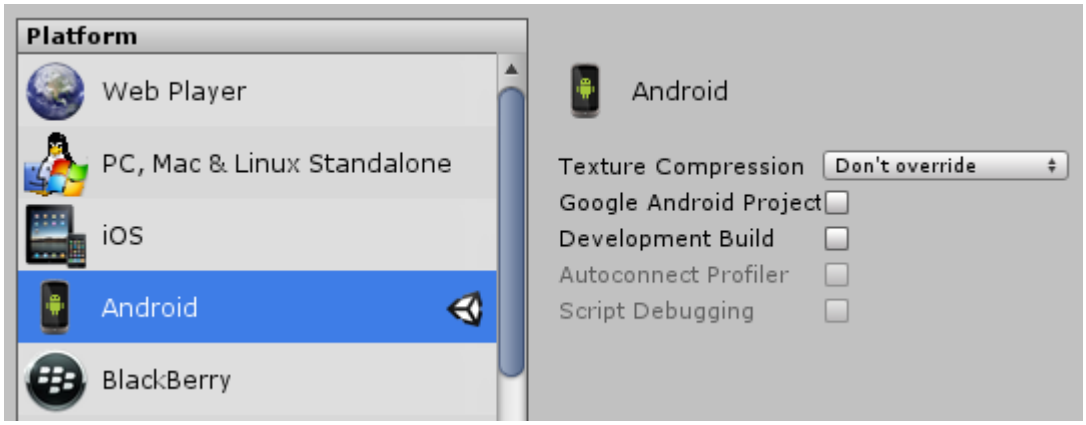
As you can imagine there are even more option for configuring desktop builds. Again the resolution tab is going to be the most important one to pay attention to:



When it comes to desktop and mobile, supporting different aspect ratios is going to be critical to the way your games looks in the finished product. At the end of the publishing section I will briefly talk about aspect ratio and how you test out each aspect ratio when playing your game back in the Unity IDE.

Android

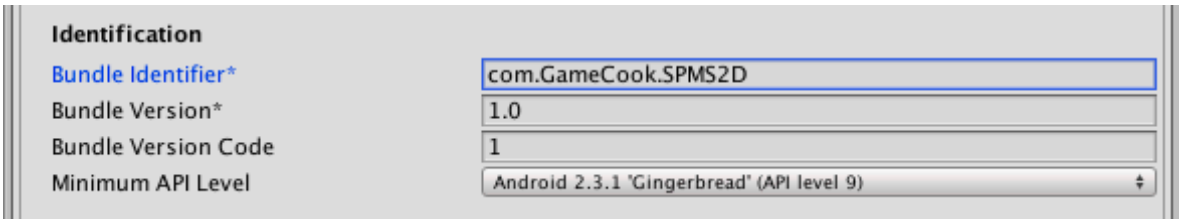
Out of all the mobile platform build options, Android is going to be the easiest one to build for.



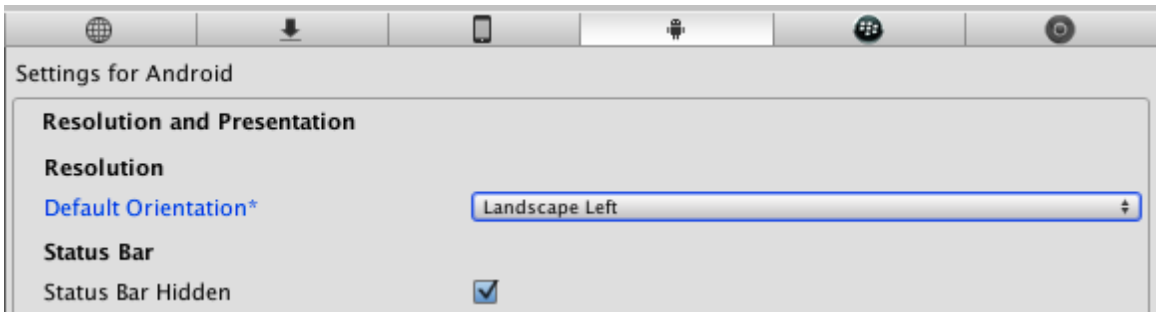
You will need to do a few additional things from what we saw when doing builds for Web or Desktop. Like with the other two builds, you will be asked for a location to save your game's APK. I created an `Android` folder in my `Build` directory just like in the previous two examples. Once you save the path

to the APK you will be asked for the path to the Android SDK. You can download this from the Android Developer Site at https://developer.android.com/sdk. Note that you will also need to have the JDK installed on Windows as well if you have not configured your computer for Android development already. On the Mac this is a little easier to setup. Once you have that all installed and ready to do an Android build, simply point Unity to the root of the Android SDK you downloaded to your computer.

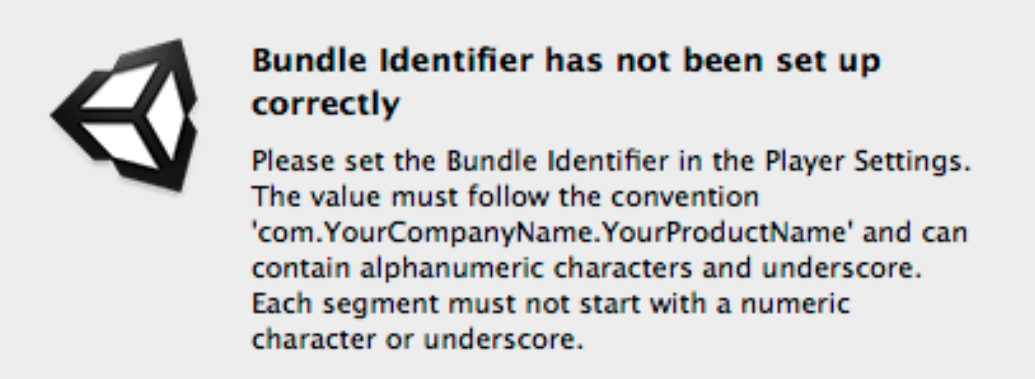
Unity also requires you to make modifications to the Android build settings, most important being updating the `Bundle Identifier`.



Since we are also building for mobile it may be best to set the `Default Orientation` to `Landscape` and make sure to check `Status Bar Hidden`.



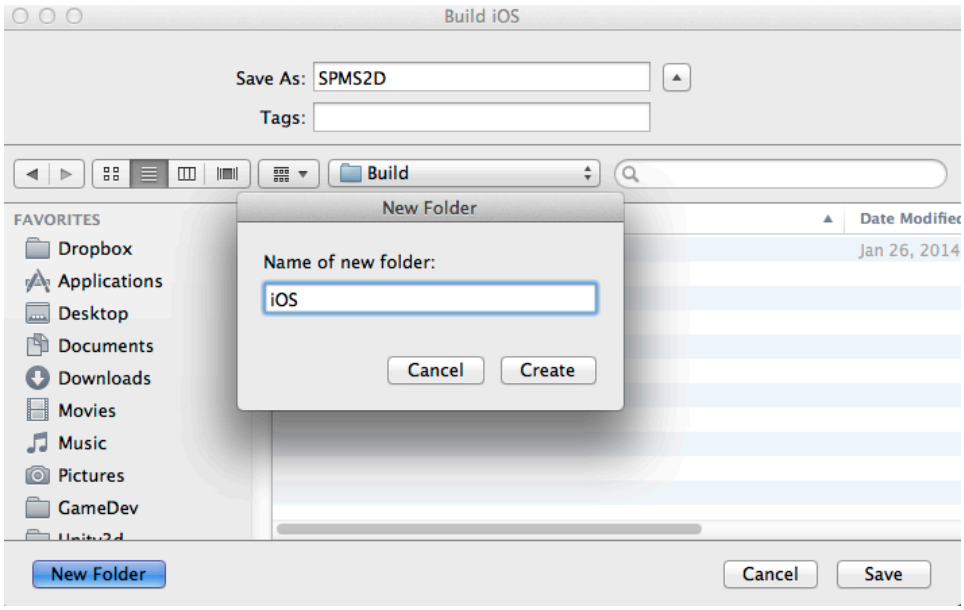
If you start a build and forget to change the Bundle Identifier you will get a build error like this.



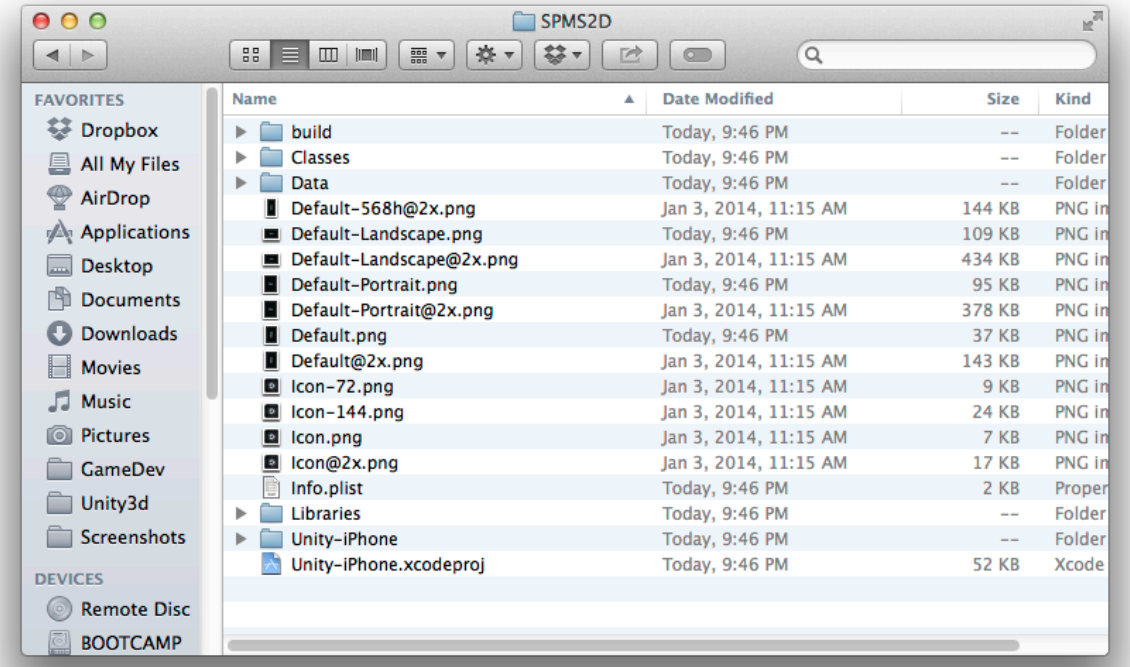
Outside of that you should have everything setup now to build to Android. You can chose to manually install the generated APK yourself or connect an Android device via USB and have Unity install and run it for you which is incredibly convenient.

iOS

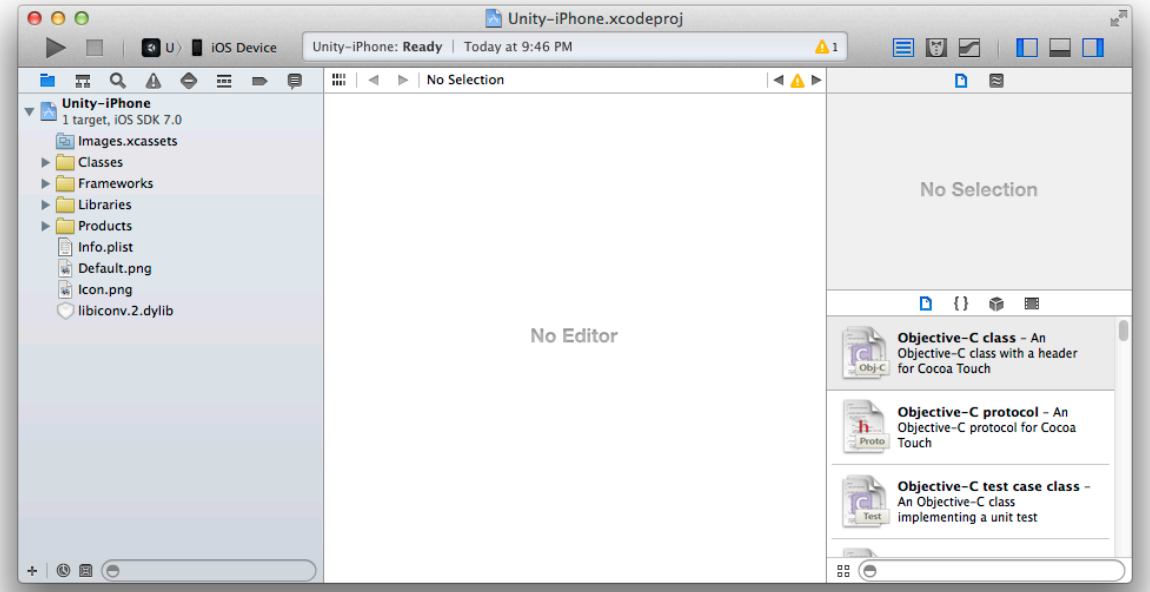
iOS perhaps one of the more difficult platforms to get up and running on. If you have done any work on iOS before you will be familiar with setting up provisioning and testing apps on devices. At a high level, when you do an iOS build you will be asked for a place to save your final game. Here you can see I set up an iOS folder just like we did for all of our other platforms.



It is critical that you create a separate folder for your iOS build because Unity doesn't just create a single file for your game; it actually generates a full Xcode project. Once the build is done you can open up the directory to see the project:



From here you can open up the `Unity-iPhone.xcodeproj` file and begin working on setting it up to run in Xcode:

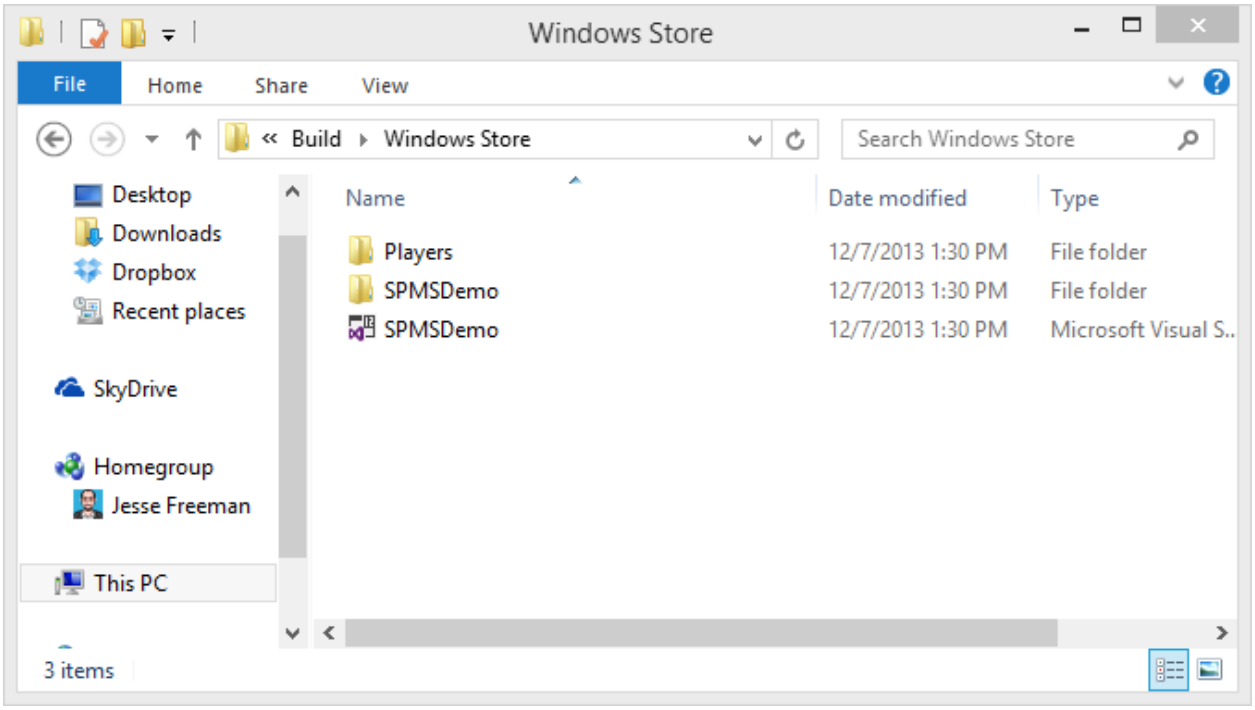


Unity has an excellent getting started guide on working with iOS, which you can check out on their site at http://docs.unity3d.com/Documentation/Manual/iphone-GettingStarted.html. There area steps involved in order to get an iOS device configured to test with including setting up your developer provisioning and readying a device for developement. Just keep in mind that you will need to have a physical iOS device to test your game out with since you can't rely on the emulator.

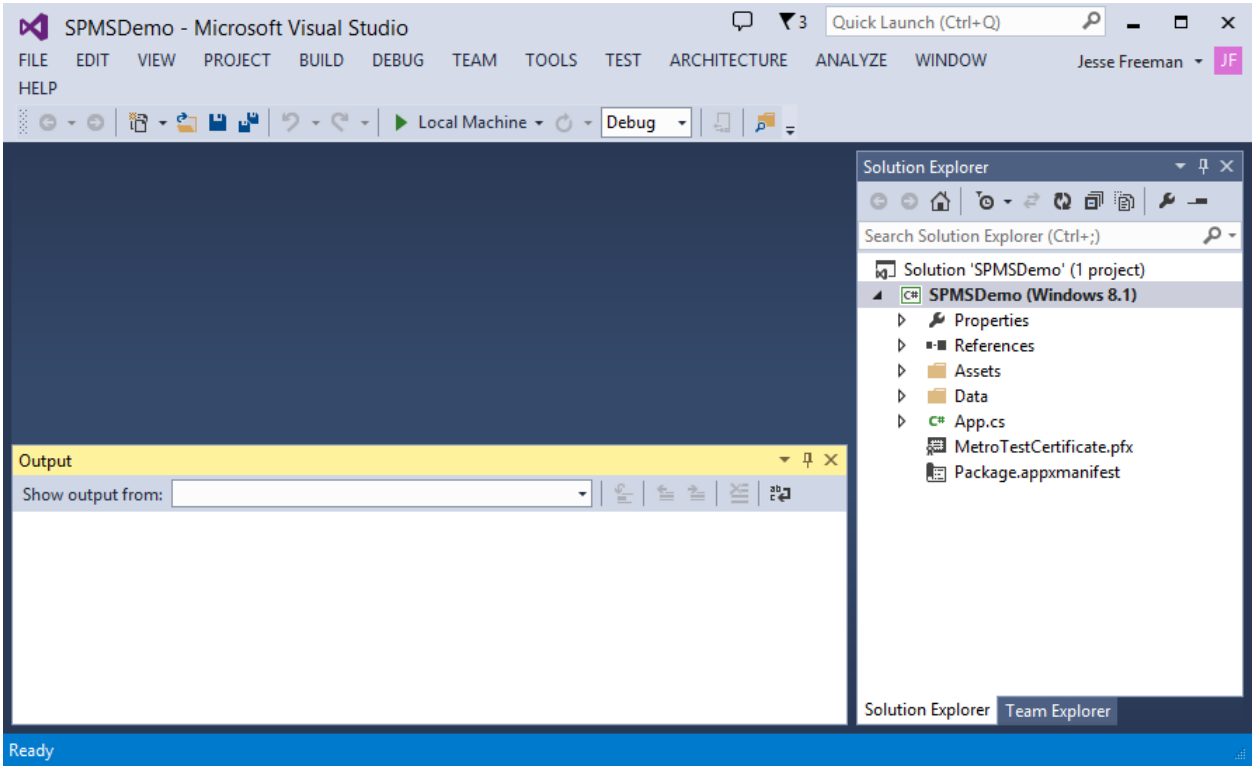
Windows Store

While there may be a little confusion over the difference between the Windows Store Apps and Windows Desktop, this option is to target Windows 8. This means that while you can still create desktop builds of your games that run on Window 8 via the PC build option, if you want to distribute them in the Window Store you will have to select it specifically as a build target.

There are two targets for Windows 8, 8.0 and 8.1. It is also important to mention that in order to build to Windows Store you need to be on Windows 8 or 8.1. When you run a Windows Store build it will generate a Visual Studio Project similar to what we saw on iOS. I save my build out to its own `Windows Store` folder in my `Build` directory.



Simply click on the solution file to open it up in Visual Studio.



From here you can click on the compile button at the top of the screen with the green arrow that says Local Machine. This will build the game and install it as a Modern App for testing.

Just like the other platforms, you will want to walk through the configuration options and make sure you support the additional artwork such as icons, splash screen and more for the Windows Store.

Windows Phone

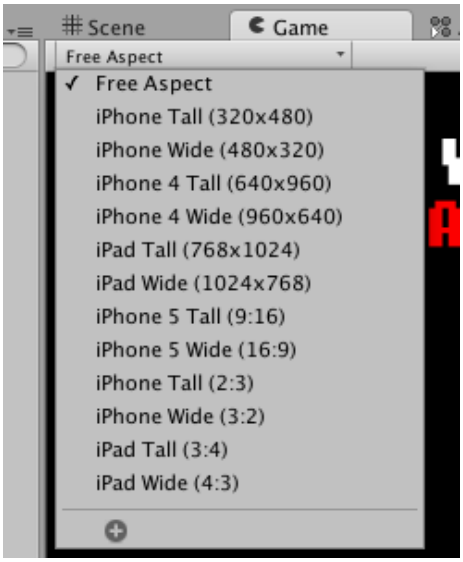
Windows Phone is similar to building for Window Store. Unity will output a Visual Studio project for Windows Phone and you'll follow a similar process to how we created the Windows 8 Visual Studio Project. Simply follow the steps you have gone through above in the Windows 8 build to create a Windows Phone build in Visual Studio. You will need a physical Windows Phone for testing.

Touch Controls

We cheated a little in our game by only relying on a single point of input, the mouse. The good news is that you should get touch support built in when you run it on touch enabled devices like Android, iOS, Windows 8 and Windows Phone. Unity supports a multi-touch API but we will not be covering it here. I just wanted to bring this up as you create more complex games that you will need to consider what input looks like across multiple devices. In my games, as you can see I always try to keep the input to a single source so it works seamlessly across multiple devices well while offering keyboard or controller support on platforms that can support it. Make sure to check out Unity's input documentation at http://docs.unity3d.com/Documentation/Manual/Input.html.

Aspect Ratios

The final thing I want to cover is how Unity allows you to preview different aspect ratios. As you know, trying to support all of these different devices is challenging. When you move from platform to platform in the build settings, Unity offers up Aspect Ratio options in the IDE:



At the end of the day, resolution comes down to aspect ratio, is your game 4:3 or 16:9 or even both? Good games can easily support either resolution; here is an illustration from the original game I based this book on:

Here you can see I am supporting two different aspect ratios and three different resolutions. The key is to have a viewport that can expand or contract to adapt to the different resolutions. For an example, at 800x400 pixels you see the minimum screen real estate needed to play the game. On higher resolutions/aspect ratios your field of view is expanded to fill in the extra space. Also, UI should always be "liquid" meaning it can be pinned to different corners of the screen and moves into place based on the current resolution.

Here is what the game is designed to look like at 800 x 480 or 4:3:

You'll notice that the game elements take up most of the real estate of the screen. Compare this to the 16:9 version:

Here you see a lot more of the action. The UI has expanded out more and you can get a large area of the background as well as what is to the side of the player but everything is still centered nicely.

I have found that mocking out a single game screen with resolution guides on it helps me better visualize how my game will scale across all of the different devices that it will play on. Unity does a great job of helping you figure out what the game is going to look like by taking advantage of the aspect ratio tab in the game tab so make sure you check that from time to time to make sure everything is display as you expect it to.

Conclusion

At this point you should know everything you need to build simple 2D games in Unity 4.3. We covered a lot of the critical things you would need to actually make a full game. I have written over 30 games in 5 different languages and every time I learn a new language or framework I stop to ask myself the following questions

- How do I display graphics?
- How do I make things move?
- How do I control the camera?
- How do I dynamically create game entities?
- How do I detect collisions?
- How do I create UI?
- How do I move between scenes?

By using this basic set of questions you can quickly learn any game framework or language to help you build the games you want to make. If you were paying attention I structured the entire book to follow this flow so you lean the basic building blocks of coding a game from scratch and can apply it to your next project or framework you want to learn.

At this point the game is ready for you to customize, add to, or scrap and start from scratch. I encourage you to continue to play around with this basic game and add things like text for score, more enemies, animation for each scene or anything else you can think of. There are lots of great tutorials and guides out there on working with Unity so this project should be your sandbox for experimenting and learning more about how 2D in Unity works.