**Chapter 3. Side-scrolling Platformer** **At this point, we have a chance to work on a full game, but we've only used 2D so far. In this chapter, we will explore how we can use the concepts that we learned in 2D and use them with a 3D game with 2D gameplay.** **As long as we have played games, there has been one particular genre that has stayed with us almost from the beginning, the platformer. Starting with *Donkey Kong*with the familiar content that we know, refined in *Super Mario Brothers*, given more action with *Mega Man*, taken faster with *Sonic the Hedgehog*, and used even today with games such as *Terraria*, *Super Meat Boy*, and *Child of Light*, there is something that draws us to this specific type of game, especially within the indie game community.** **A platform game (known commonly as a platformer) consists of a player controlling**

**a character that can move around a game environment with extensive jumping be- tween platforms, hence the name.**

**Project overview** **Over the course of this chapter, we will create a complete side-scrolling platformer project. We will learn the similarities between working in 2D and 3D and the differ- ences, in particular when it comes to Physics.**

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

**•Tile-based level creation**

**•Adding player functionality**

**•Adding collectables/power ups**

**•Designing the level layout and background**

**Prerequisites** **As in Chapter 1, *2D Twin-stick Shooter*, you will need Unity installed on your com- puter, but we will start a new project from scratch.** **This chapter uses no graphical assets; however, the completed project and source files can be downloaded from the example code provided for this book on Packt's website (**[**http://www.packtpub.com**](http://www.packtpub.com)**).**

**Setting up the project** **At this point, I assume that you have a fresh install of Unity and have started it up:**

**1.With Unity started, go to File | New Project. Select a Project Location of your**

**choice somewhere on your hard drive, and ensure that you have Setup de- faults for set to 3D. Once completed, select Create. At this point, we will not need to import any packages, as we'll make everything from scratch. From there, if you see the Welcome to Unity pop up, feel free to close it out, as we won't be using it.**

**2.Create the following folders just as we described in the previous chapters:**

**•Prefabs**

**•Scenes**

**•Scripts**

**Tile-based level creation** **While our previous game worked by only needing to spawn enemies in the world, for most games with content, you'll typically have levels, each with its own envi- ronment. When building levels in games, there are some advantages to placing everything by hand, but if you're creating a game with many levels, that work will decrease your productivity. It's also important to note that the more assets you cre- ate for your game, the higher the cost.** **With that in mind, it's a much better idea to create parts that can be reused to cre- ate games. If you've played older 2D games in the past, such as an adventure, RPG, or platforming games, you may have realized that there were a lot of places in the worlds that looked similar to each other, such as the trees, a wall, a chest, door, and so forth.**

**The reason they looked similar is due to the fact that they were using the same sprites. This is because they were truly tile-based games. A tile-based game is where the playing area consists of small rectangular, square, or hexagonal graphic images, referred to as tiles. Imagine a grid of blocks where every block is given a number or ID. Based on the ID, the game will determine how that grid is drawn and behaves when a player interacts with it.** **An important thing to mention is that tile-based games are not a distinct genre; rather, the term refers to the technology a game engine uses for its visual represen- tation. For example, most of the *Pokémon*series of games are top-down, role- playing video games, and the traditional *Mario*series of games are side-scrolling platformers, but both use a tile-based system for graphics. Tile-based engines allow developers to create large levels quickly with relatively few art assets, which is great as a programmer.**

**To show how easy it is to build, we will code a tile-based system for this project:**

**1.The first thing that we're going to want to do is actually create the blocks**

**we'll be placing for the world. Let's first create a cube by selecting GameOb- ject | Create Other | Cube.**

**2.We want this cube to have a collision so that our player can collide against it,**

**but this time we will use a box collider. Check Inspector to confirm that it is there. If not, add this component by selecting Component | Physics | Box Collider.** **The box collider is the 3D equivalent of the box collider 2D component for 3D space.**

**3.Rename the cube to Wall by selecting the top bar in Inspector, renaming it,**

**and pressing *Enter*. Have a look at the following screenshot:**

**4.In the Project tab, go to the Prefabs folder, and drag-and-drop the Wall object**

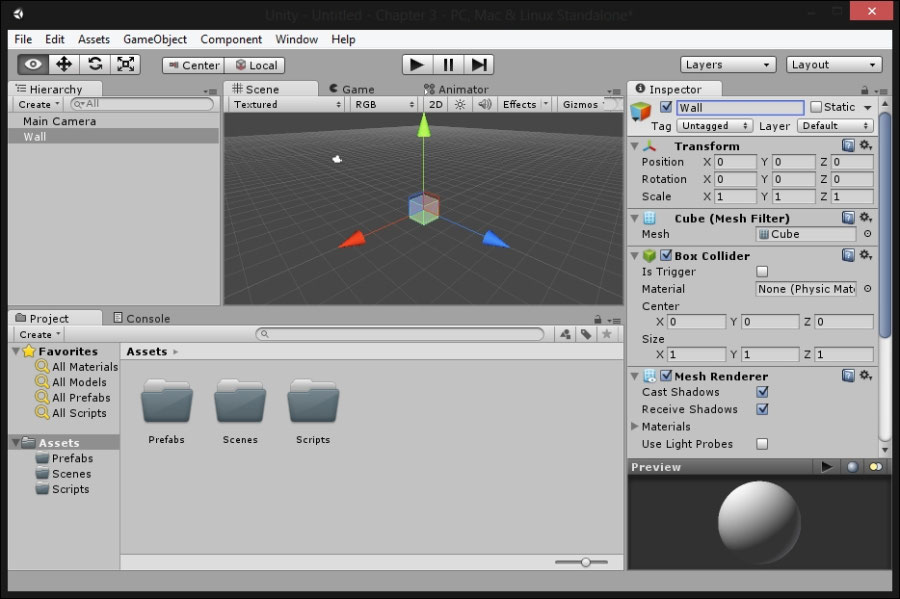
**To show how easy it is to build, we will code a tile-based system for this project:**

**The first thing that we're going to want to do is actually create the blocks we'll be placing for the world. Let's first create a cube by selecting GameObject | Create Other | Cube.**

**We want this cube to have a collision so that our player can collide against it, but this time we will use a box collider. Check Inspector to confirm that it is there. If not, add this component by selecting Component | Physics | Box Collider.**

**The box collider is the 3D equivalent of the box collider 2D component for 3D space.**

**Rename the cube to Wall by selecting the top bar in Inspector, renaming it, and pressing *Enter*. Have a look at the following screenshot:**



**from Hierarchy into it. Once that is finished, select Wall in Hierachy, and then delete it by pressing the *Delete*key.**

**5.Now, we will be spawning a large amount of objects into our world. It would**

**be a good idea to have a parent object to store all of these objects to avoid cluttering our Hierarchy. To do this, let's create an empty game object by going to GameObject | Create Empty. From there, with the object selected, go to Inspector, set its name to DynamicObjects, and optionally, for neatness sake, set its Position to (0, 0, 0).**

**6.The next thing we will do is create a game controller to hold the behavior to**

**create our blocks. Create an empty game object with the name \_GameController, and reset its Position to (0, 0, 0).**  **Note** **I put the \_ before the actual name of the object in my projects, so it's**

**always at the top of my hierarchy, and hence I have easy access to it. If this does not work, you can also drag-and-drop objects to change their place in the hierarchy.**

**7.Underneath the name, you'll see the Tag property. Change it from Untagged**

**to GameController.**

**8.Next, with the game controller selected, go to the Inspector tab and then se-**

**lect Add Component | New Script. Once brought to the next menu, change the language to C#, and set the name of the script to GameController.**

**9.Select the newly created script, and move it to the Assets\Scripts folder. Go**

**to MonoDevelop by double-clicking on the script file.**

**10.Inside the newly created code, we will first need to add two new variables**

**for us to use: level, which will contain the data needed to create our level and wall, which will contain the block we want to spawn:**

**private int[][] level = new int[][] {**

**new int[]{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},**

**new int[]{1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},**

**new int[]{1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},**

**new int[]{1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1},**

**new int[]{1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1},**

**new int[]{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1} };**

**public Transform wall;**

**The wall variable looks similar to things we've created before, but the level variable looks a bit different.**

**Working with arrays** **The level variable is an array. We could create an integer for each place inside of our level, detailing what type is there; however, this is quite tedious, and we would have to remember each element's identifier. An array is a holder of multiple elements of the same type. To access an individual element of the array, we simply need to specify an index of where it is placed in between square brackets (the [ and ] char- acters). Arrays are played sequentially in memory, which means it's really easy to move between elements of them, and it's a very fast operation to access an indi- vidual element.** **The level variable is actually a multidirectional array, which can be thought of as an array of an array of integers. We will use a multidirectional array, because it allows us to draw with numbers like a grid to place each of the elements in our level.**

**That being said, now we actually need to build the level. To do that, perform the fol- lowing steps:**

**1.Let's create a function called BuildLevel:**

**void BuildLevel() {**

**// Get the DynamicObjects object that we created already in the**

**// scene so we can make it our newly created objects' parent**

**GameObject dynamicParent = GameObject.Find ("DynamicObjects");**

**//Go through each element inside our level variable**

**for (int yPos = 0; yPos < level.Length; yPos++)   {**

**for (int xPos = 0; xPos < (level[yPos]).Length; xPos++)     {**

**// Do nothing if the value is 0**

**// If the value is 1, we want a wall**

**if (level[yPos][xPos] == 1)       {**

**// Create the wall**

**Transform newObject = Instantiate (wall, new Vector3(xPos, (level.Length - yPos), 0), Quaternion.identity) as Transform;**

**// Set the object's parent to the DynamicObjects**

**// variable so it doesn't clutter our Hierachy**

**newObject.parent = dynamicParent.transform;       }**

**}   } }**

**Note** **The quaternion class is what is used for rotations inside of Unity. In this instance, Quaternion.identity stands for a matrix on rotation. For more information on quaternions, please check out http://docs.unity3d.com/ScriptReference/Quaternion.html.**  **As you can see, we access each of the arrays stored in the level by using array[index], and for an index inside of that array, we use array[index1][index2].**

**2.Next, we need to actually call this function. Do so in your Start function:**

**void Start () {   BuildLevel (); }**

**3.Save the script, and exit out to the Unity editor. When you get back, you**

**should see under the GameController script that is there in the wall variable, which still needs a value for its variable. In order to assign the prefab we cre- ated previously, we'll need to go to the folder, and then drag-and-drop it into the box for the variable and then release the mouse. Have a look at the fol- lowing screenshot:**

**4.After this, click on the Start button to see the code execute:**

**You may see a bit of the result in the Game screen that pops up, but if you click on the Scene tab, you'll see the level has been built for us!**  **Note** **You can drag-and-drop the Scene tab to share space with the Game tab if you want, as you can see in the preceding screenshot.** **There are a number of other ways that you can use to modify your layout as well. Some of them are provided and will help your workflow.** **To view them, you can either go to the Window | Layouts menu or select the right-most drop-down menu on the toolbar.**

**I personally use the Default layout for this book, but when I have two monitors, I like to spread things out with the Game tab on one monitor and everything else.** **One of our technical editors prefers the 2 by 3 layout with a one-column Project tab (right-click the Project tab, and then select One Column Layout.**

**Creating our player** **Having the basis of our world is great, but if we don't have a player, it doesn't mat- ter how nice the level looks. In this section, we will create the actual player that will walk around and move in the world:**

**1.Let's first create a capsule by selecting GameObject | Create Other | Capsule.**

**Have a look at the following screenshot:**

**Tip** **If you are using Unity 4.6 or above, use the GameObject | 3D Object | Cap- sule**