Community Experience Distilled

Unity 5.x By Example

An example based practical guide to get you up and running with Unity 5.x

Alan Thorn

[PACKT] PUBLISHING open source* community experience distilled

Table of Contents

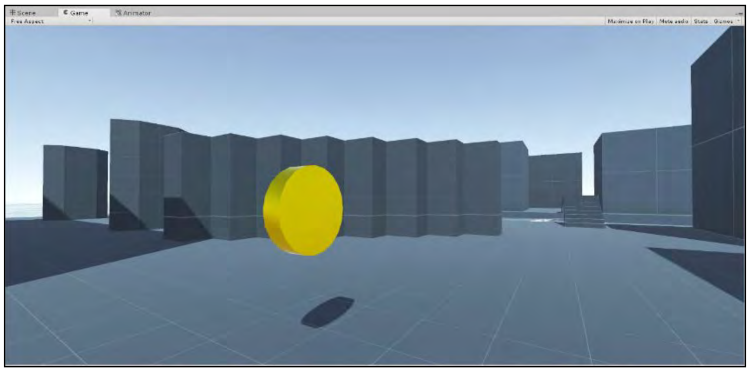
1 The Coin Collection Game: Part 1

This chapter starts the first project on our list, which will be a fun collection game. And remember, it doesn't matter if you've never used Unity before. We'll go through everything necessary step by step. By the end of this chapter, you'll have pieced together a simple, but complete and functional game. This is an important thing to achieve, because you'll get familiar with a start to end game development workflow. This chapter demonstrates:

1. Game Design
2. Projects and Folders
3. Asset Importing and Configuration
4. Level Design
5. Game Objects
6. Hierarchies

Game Design

Let's make a coin collection game. Here, the player should control a character in first-person mode, and they must wander the level, collecting all coins before a time-limit runs out. If the timer runs out, the game is lost. And, if all coins are collected before the timer expires, the game is won. The first person controls will use the default WASD keyboard setup, where *W* moves forward, *A* and *S* move left and right, and *D* walks backwards. Head movement is controlled using the mouse, and coins are collected by simply walking into them. See Figure 1.1, featuring the coin collection game in action inside the Unity editor. The great benefit in making this game is that it demonstrates all core Unity features together, and we don't need to rely on any external software for making assets, like textures, meshes and materials.

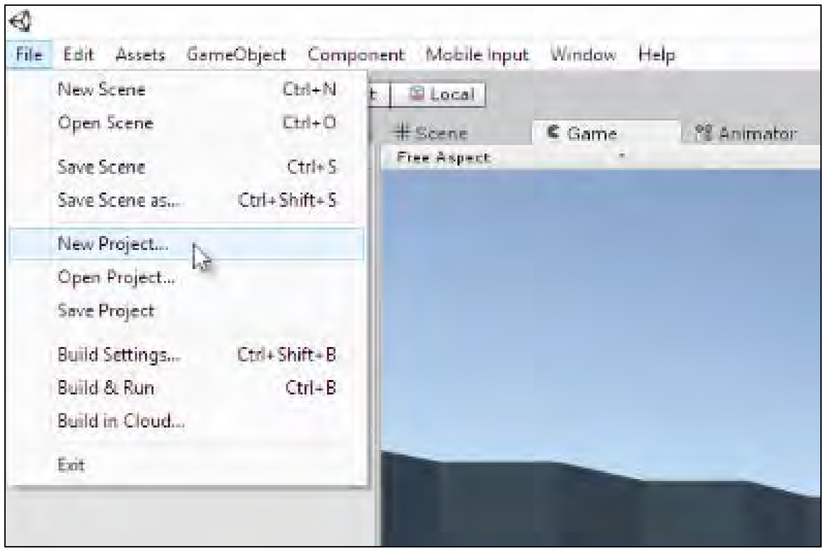


Preparing for a coin collection game

The completed 'Collection Game' project, as discussed in this chapter and the next, can be found in the book companion files, inside the *Chapter01/CollectionGame* folder.

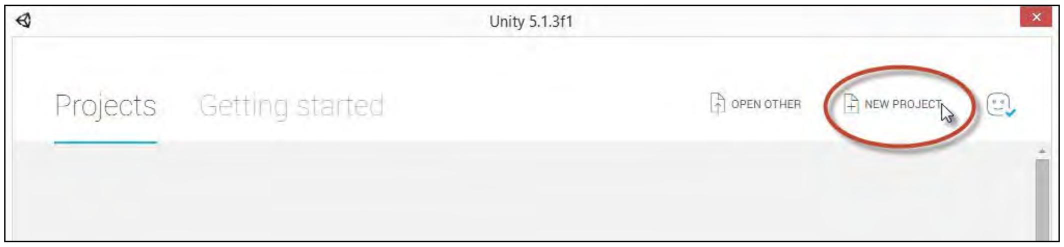
Getting Started – Unity and Projects

Every time you want to make a new Unity game, including coin collection games, you'll need to create a new **Project**. Generally speaking, Unity uses the term 'Project' to mean a 'Game'. There are two main ways to make a new project, and it really doesn't matter which one you choose, because both end up in the same place. If you're already inside the Unity interface, looking at an existing scene or level, you can select *File* > *New Project* from the application menu. See Figure 1.2. It may ask if you want to save changes to the currently opened project, and you should choose either Yes or No, depending on what you need. After selecting the *New Project* option, Unity leads you to the project creation wizard.



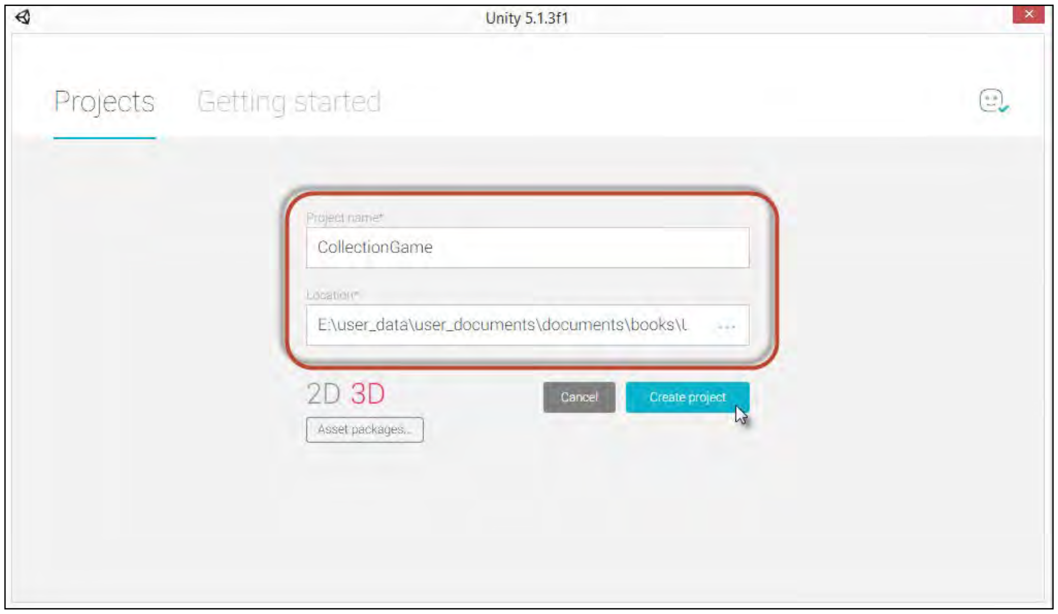
Creating a new Project via the Main Menu

Alternatively, if you've just started Unity for the first time, you'll probably begin at the Welcome dialog. See Figure 1.3. From here, you can access the New Project creation wizard by choosing the *New Project* button.



The Unity Welcome Screen

On reaching the New Project creation wizard, Unity can generate a new project for you on the basis of some basic settings. Simply fill in the name of your project (Such as *CollectionGame*), and select a folder on your computer to contain the project files that will be generated automatically. Finally, click the *3D* Button to indicate that we're going to create a 3D game, as opposed to 2D, and then finally click the *Create Project* button to complete the project generation process. See Figure 1.4.



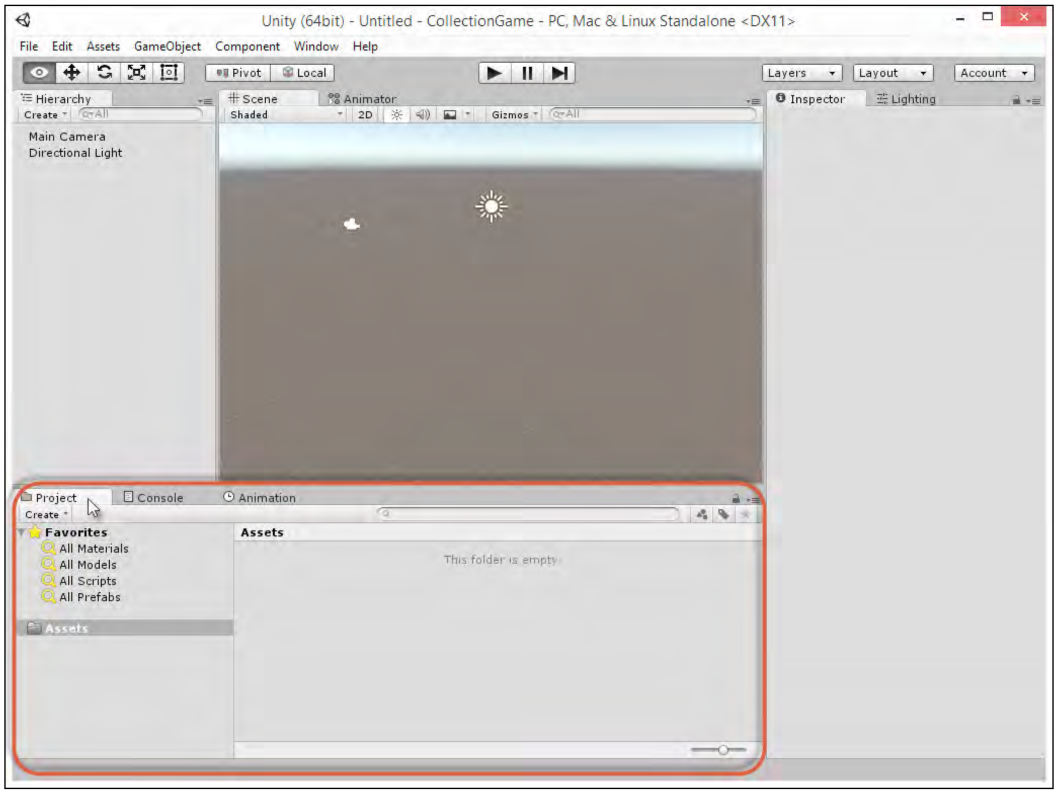
Project name*
CollectionGame
Location*
E:\user_data\user_documents\documents\books\L

Creating a new Project...

Projects and Project Folders

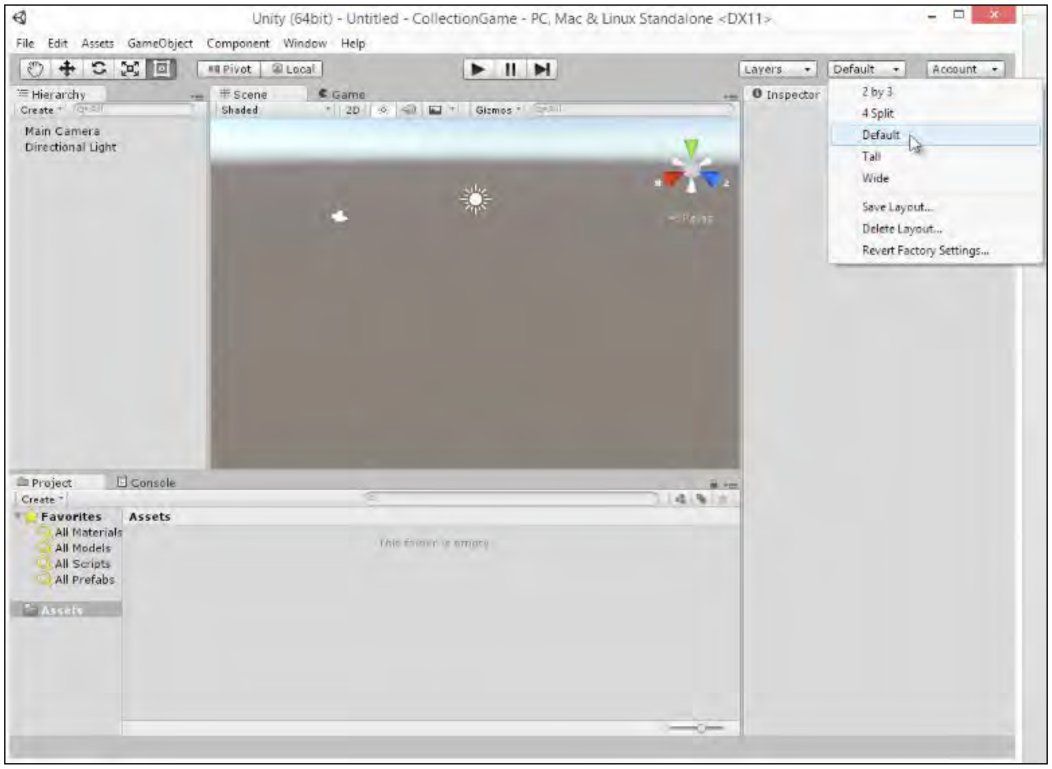
Unity has now created a blank, new and empty project. This represents the starting point for any game development project and is the place where development begins. The newly created project contains nothing initially: no meshes, textures or any other assets. You can confirm this by simply checking the **Project Panel** area at the bottom of the editor interface. This panel displays the complete contents of the Project Folder, which corresponds to an actual folder on your local drive, created earlier by the project wizard. This folder should be empty.

See Figure 1.5. This panel will later be populated with more items, all of which we can use to build a game.



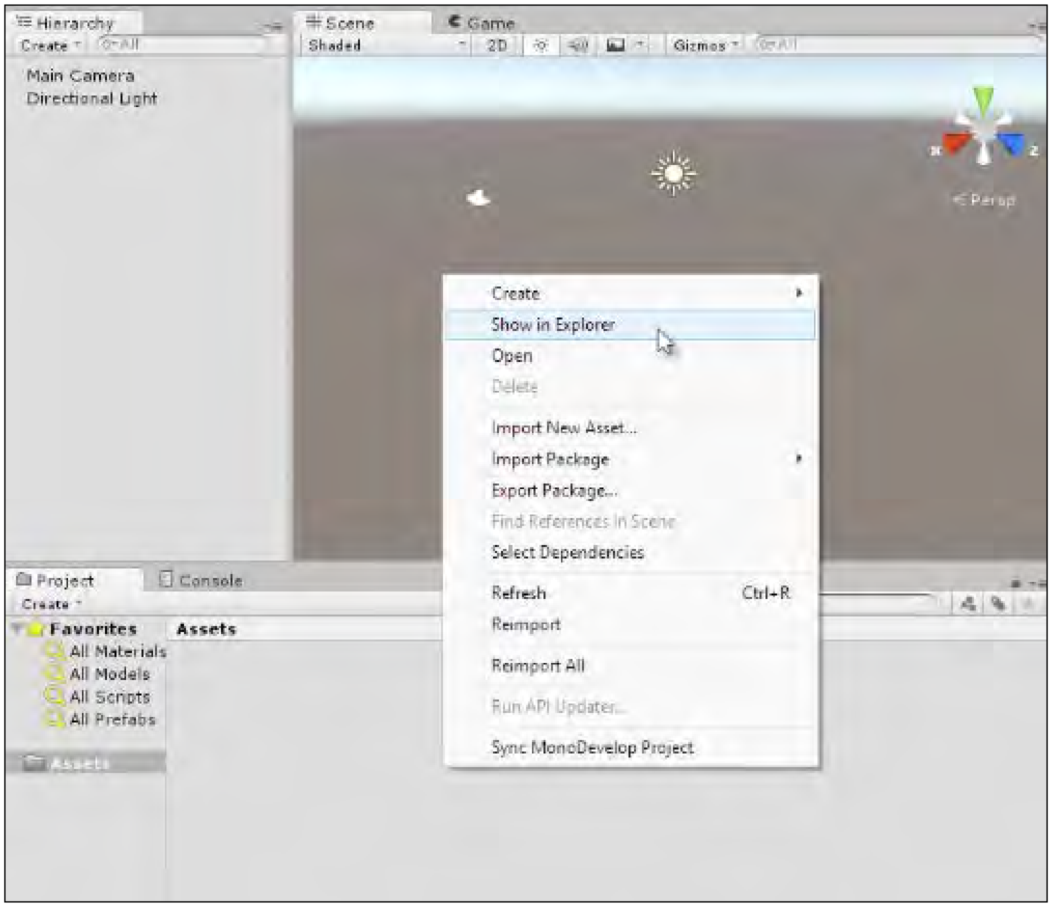
The Unity Project Panel docked at the bottom of the interface...

If your interface looks radically different from Figure 1.5, in terms of its layout and arrangement, then you can reset the UI layout to its defaults. To do this, click the *Layout* drop-down menu from the top-right corner of the editor interface, and choose *Default*. See figure 1.6.



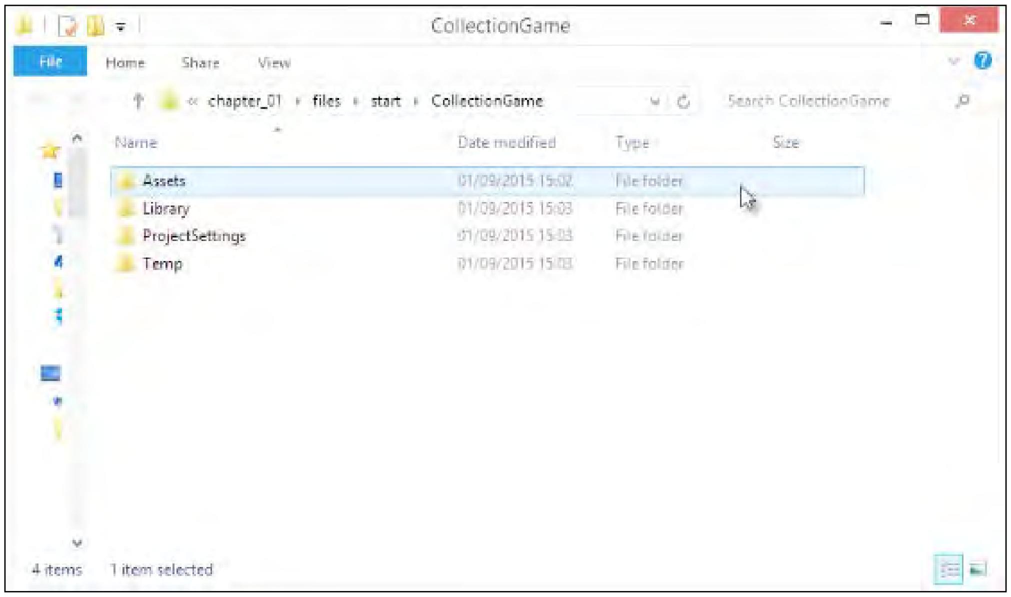
Switching to the Default interface layout

You can view the contents of your project folder directly via either Windows Explorer or Mac Finder, by right-clicking the mouse inside the Project Panel from the Unity editor to reveal a context menu, and from there choose the option *Show in Explorer* (Windows) or *Reveal in Finder* (Mac). See Figure 1.7.



Displaying the Project Folder via the Project Panel

Clicking *Show in Explorer* displays the folder contents inside the default system file browser. See Figure 1.8. This view is useful for inspecting files, counting them, or backing up files. However, you shouldn't change the folder contents manually this way. Don't move or delete files from here, because doing so can corrupt your Unity project irretrievably. You should instead delete and move files, where needed, from within the Project Panel inside the Unity editor. This way, Unity updates its meta-data as appropriate, ensuring your project continues to work properly.



Name	Date modified	Type	Size
Assets	01/09/2015 15:02	File folder	
Library	01/09/2015 15:03	File folder	
ProjectSettings	01/09/2015 15:03	File folder	
Temp	01/09/2015 15:03	File folder	

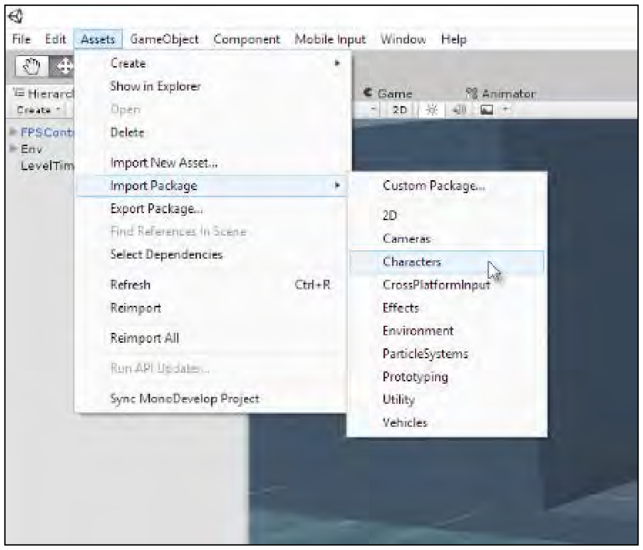
Viewing the Project Panel from the OS file browser

> Viewing the Project Folder inside the OS file browser will display additional files and folders not visible inside the Project Panel: such as *Library* and *ProjectSettings*, and maybe a *Temp* folder. Together, these are known as the Project **Meta-data**. This is not directly a part of your project per se, but contains additional settings and preferences that Unity needs to work properly. These folders and their files should not be edited or changed.

Importing Assets

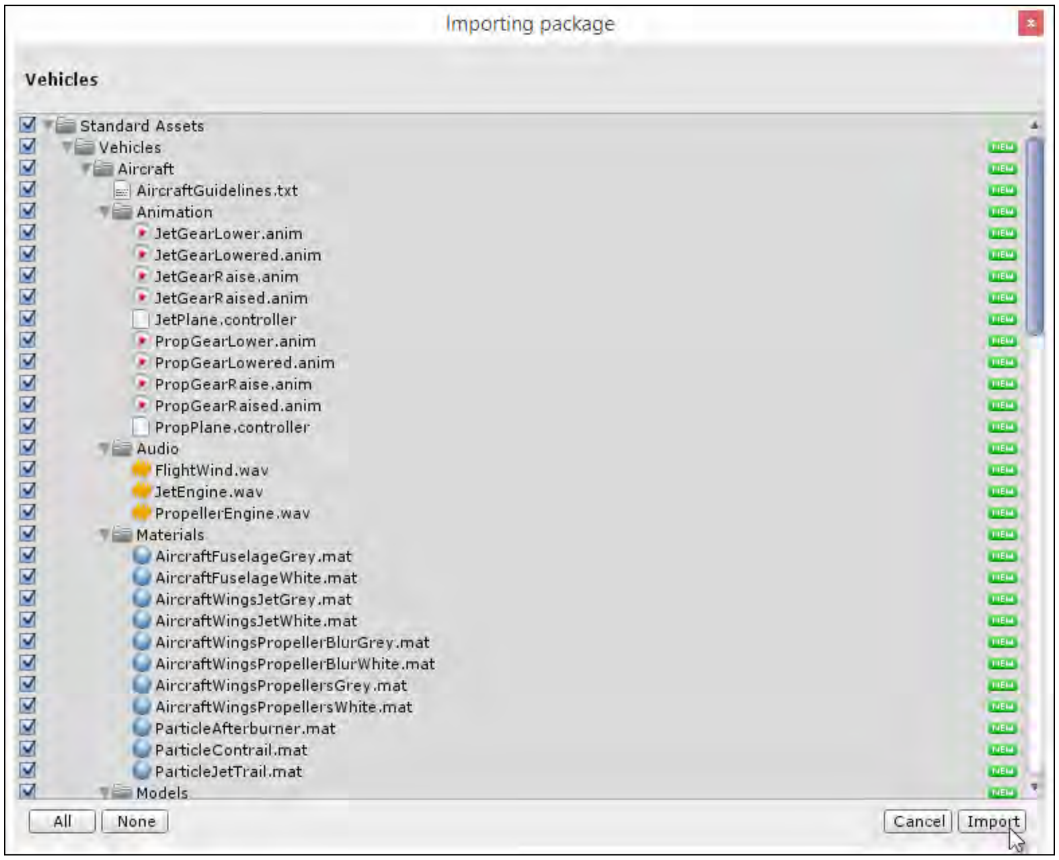
Assets are the raw materials for games; the building blocks from which they're made. Assets include: **Meshes** (or 3D models) like characters, props, trees, houses and more; **Textures**, which are image files like JPEG sand PNGs. These determine how the surface of a Mesh should look; and **Music** and **Sound** effects to enhance the realism and atmosphere of your game. And finally **Scenes**, which are 3D spaces or worlds where meshes, textures, sounds and music live, exist and work together holistically as part of a single system. Thus, games cannot exist without assets- they would otherwise look completely empty and lifeless. For this reason, we'll need assets to make the coin collection game we're working towards. After all, we'll need an environment to walk around and coins to collect!

Unity, however, is a 'game engine' and not an 'asset creation' program. That means assets, like characters and props, are typically made first by artists in external, third-party software. From here, they are exported and transferred ready-made into Unity, and Unity is responsible only for bringing those assets to life into a coherent game that can be played. Third party asset creation programs include: Blender, Maya, or 3DS Max for making 3D models; Photoshop or GIMP for creating textures, and Audacity for generating audio. There're plenty of other options too. The details of these programs is beyond the scope of this book. In any case, Unity *assumes* you *already* have assets ready to go, to import for building a game. For the coin collection game, we'll use assets that ship with Unity. So let's import these into our project. To do this, select *Assets* > *Import Package* from the application menu. Then select *Characters*, *ParticleSystems*, *Environment*, and *Prototyping*. See Figure 1.9.



Importing assets via the Import Package menu

Each time you import a package from the menu, you'll be presented with an Import dialog. Simply leave all settings at their defaults, and click *Import*. See Figure 1.10.

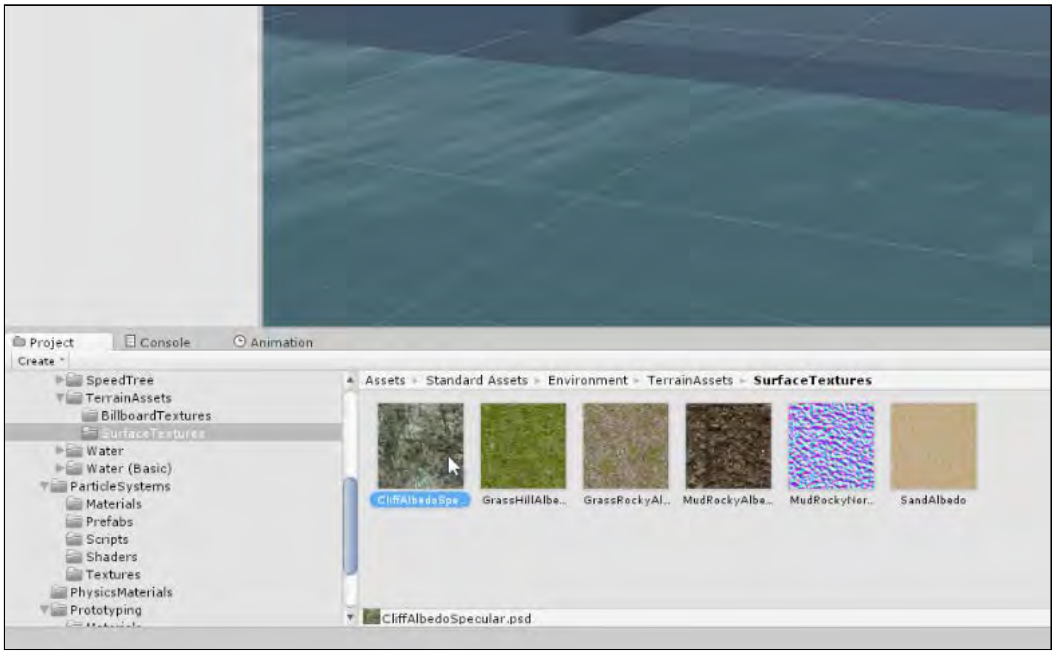


Vehicles

```
Standard Assets
  Vehicles
    Aircraft
      AircraftGuidelines.txt
      Animation
        JetGearLower.anim
        JetGearLowered.anim
        JetGearRaise.anim
        JetGearRaised.anim
        JetPlane.controller
        PropGearLower.anim
        PropGearLowered.anim
        PropGearRaise.anim
        PropGearRaised.anim
        PropPlane.controller
      Audio
        FlightWind.wav
        JetEngine.wav
        PropellerEngine.wav
      Materials
        AircraftFuselageGrey.mat
        AircraftFuselageWhite.mat
        AircraftWingsJetGrey.mat
        AircraftWingsJetWhite.mat
        AircraftWingsPropellerBlurGrey.mat
        AircraftWingsPropellerBlurWhite.mat
        AircraftWingsPropellersGrey.mat
        AircraftWingsPropellersWhite.mat
        ParticleAfterburner.mat
        ParticleContrail.mat
        ParticleJetTrail.mat
      Models
```

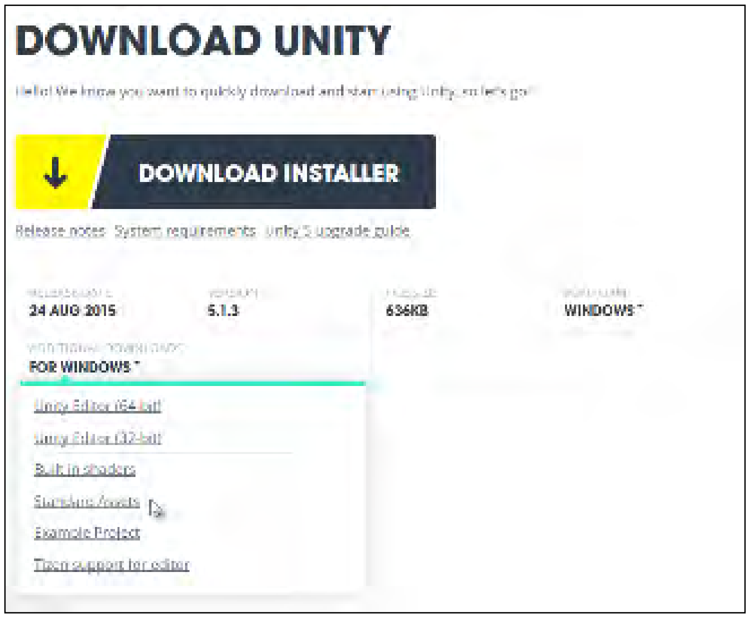
Importing assets via the Import Package menu

By default, Unity decompresses all files from the **Package** (a library of assets) into the current project. After import, lots of different assets and data will have been added to the Project, ready for use. These files are *copies* of the originals. So any changes made to the imported files will not affect or invalidate the originals, which Unity maintains internally. The files include models, sounds, textures and more. These are listed inside the Unity Editor from the Project Panel. See the following screenshot.



Browsing imported assets from the Project Panel

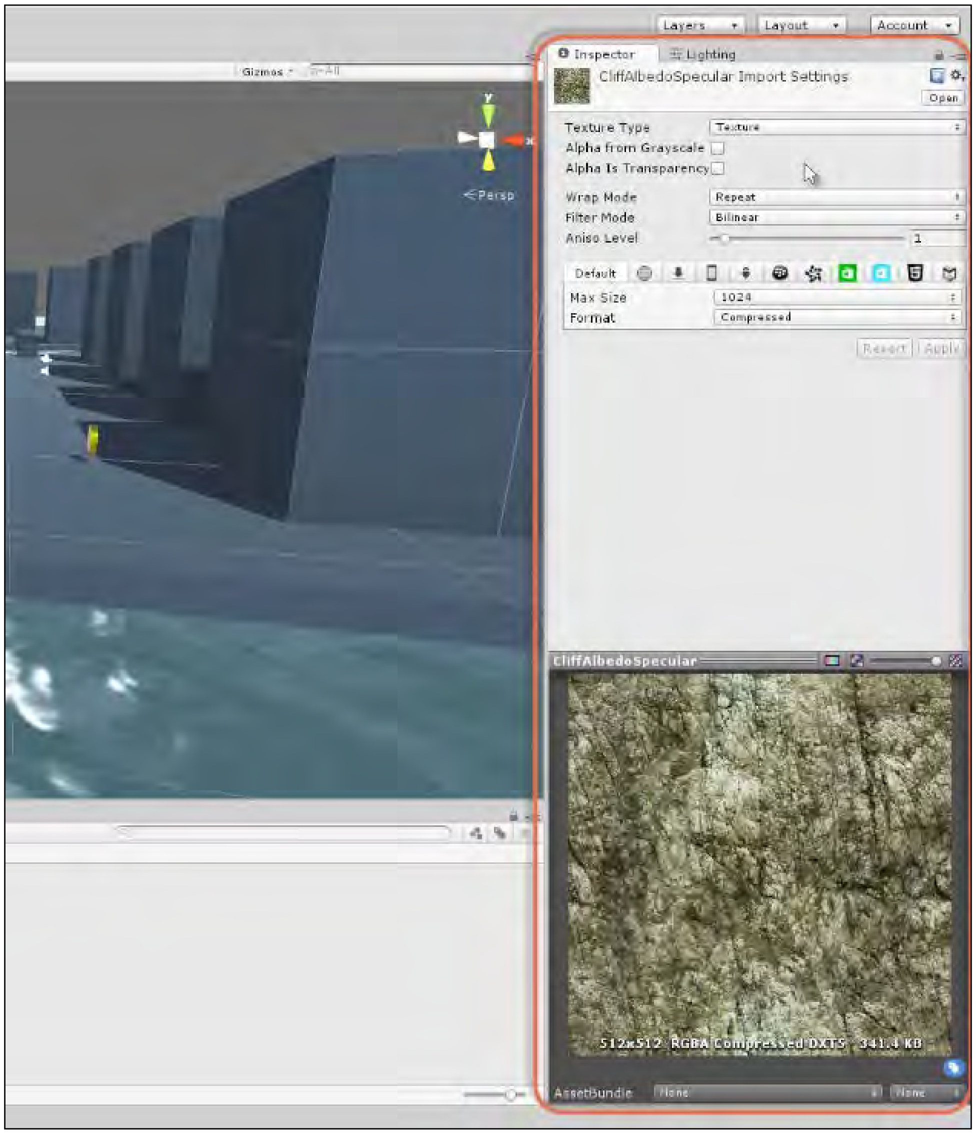
When selecting *Assets* > *Import* from the application menu, if you don't see all, or any, asset packages listed, you can download and install them separately from the Unity website at https://unity3d.com/ From the Downloads page, choose the option *Additional Downloads*, and then select the *StandardAssets* package. See Figure 1.12.


DOWNLOAD UNITY

Hello! We know you want to quickly download and start using Unity, so let's go!

Downloading the Standard Assets Package

The imported assets don't exist yet in our game. They don't appear on screen, and they won't 'do anything', yet! Rather, they're simply added to the Project Panel, which behaves as a library or repository of assets, from which we can pick and choose to build up a game. The assets imported thus far are built-into Unity and we'll be continually using them in subsequent sections to make a functional coin collection game. To get more information about each asset, you can select the asset by clicking it with the mouse, and asset-specific details will be shown on the right-hand side of the Unity Editor; inside the **Object Inspector**. The Object Inspector is a property sheet editor that appears on the right-hand side of the interface. It is context sensitive and always changes to display properties for the selected object. See Figure 1.13.



CliffAlbedoSpecular Import Settings

Texture Type: Texture
Alpha from Grayscale
Alpha Is Transparency
Wrap Mode: Repeat
Filter Mode: Bilinear
Aniso Level: 1
Max Size: 1024
Format: Compressed

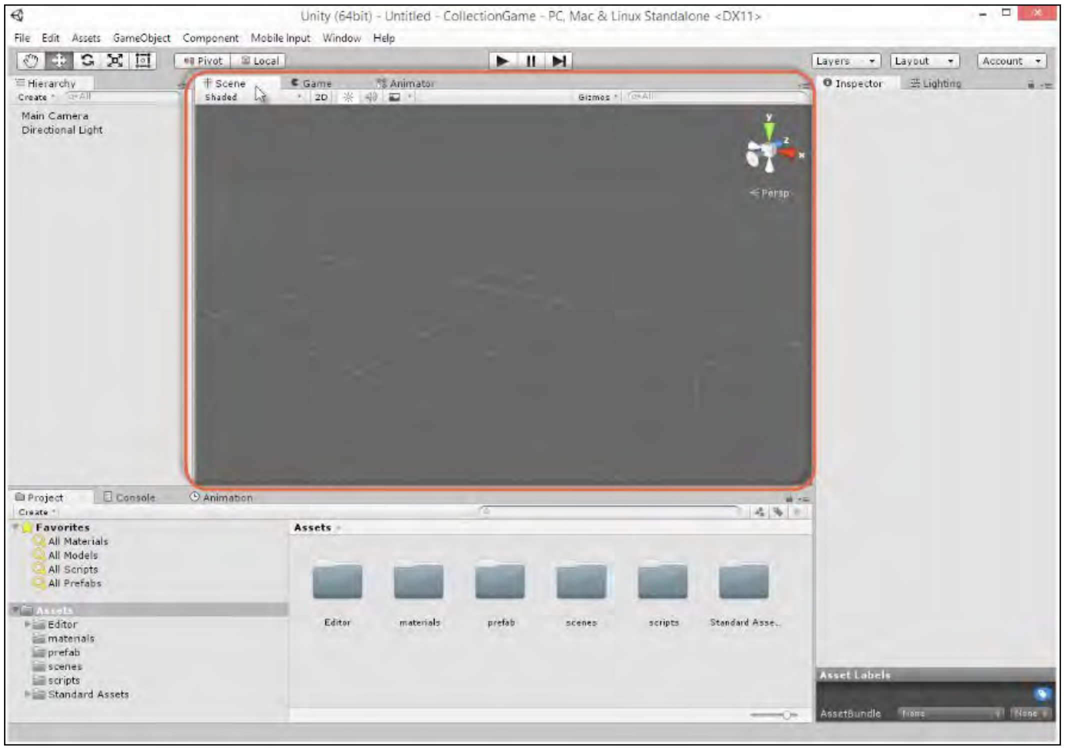
CliffAlbedoSpecular

512x512 RGBA Compressed DXT5 341.4 KB

The Object Inspector displays all properties for the currently selected object

Starting a Level

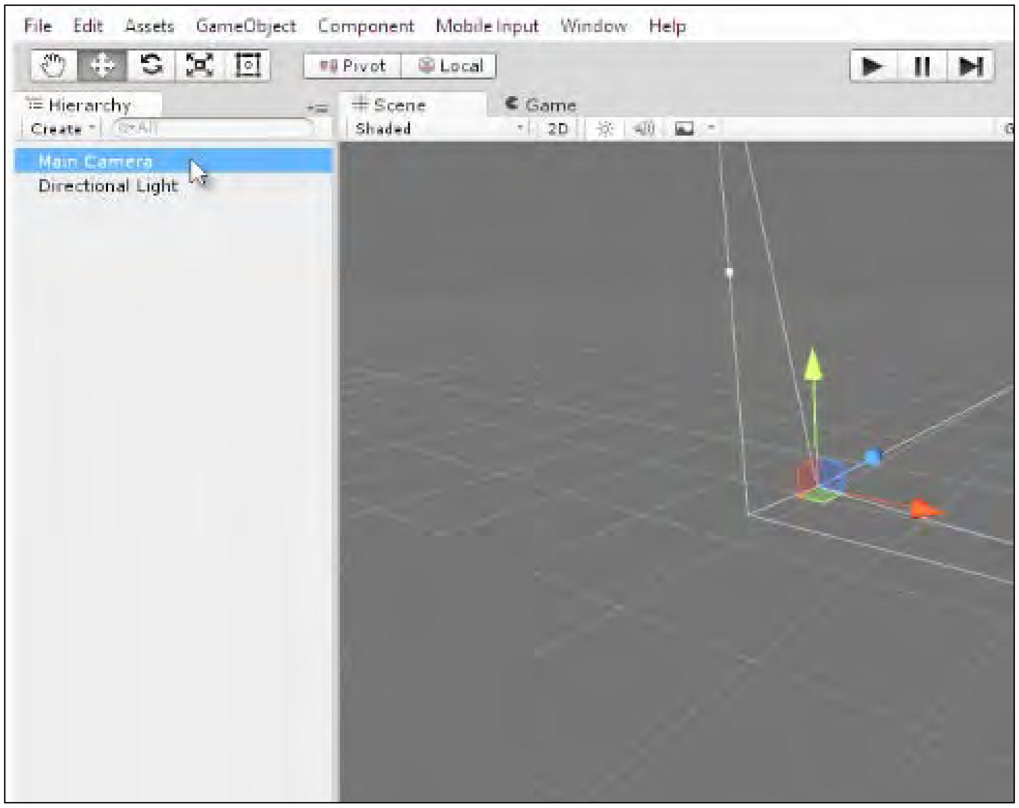
We've now created a Unity project and imported a large library of assets via the Unity standard asset packages, including architectural meshes for walls, floors, ceilings and stairs. This means we're now ready to build our first level using those assets! Remember, in Unity, a **Scene** means a **Level**. The word 'Scene' and 'Level' can be used interchangeably here. It refers simply to a 3D space. That is, the space-time of the game world; the place where 'things' exist. Since all games happen in space and time, we'll therefore need a scene for the coin collection game. To create a new Scene, select *File* > *New Scene* from the application menu, or press *Ctrl* + *N* on the keyboard. When you do this, a new and empty scene is created. You can see a visualization or preview of the scene via the **Scene** tab, which occupies the largest part of the Unity interface. See Figure 1.14.



The Scene tab displays a preview of a 3D world

As shown in Figure 1.14, other tabs beside Scene are visible and available in Unity. These include: a *Game* tab, and an *Animator* tab; and, in some cases, there could be more as well. For now, we can ignore all tabs except Scene. The Scene tab is designed for quick and easy previewing of a level during its construction.

Each new scene begins empty; well, almost empty. By default, each new scene begins with two objects. Specifically: a *Light*, to illuminate any other objects that are added, and a *Camera* to display and render the contents of the scene from a specific vantage point. You can view a complete list of all objects existing in the scene by using the **Hierarchy Panel**, which is docked to the left-side of the Unity interface. See Figure 1.15. This panel displays the name of every **GameObject** in the scene. In Unity, the word 'GameObject' simply refers to a single, independent and unique 'thing' that lives within the scene, whether visible or not: meshes, lights, cameras, props and more. Thus, the Hierarchy Panel tells us about everything in the scene.

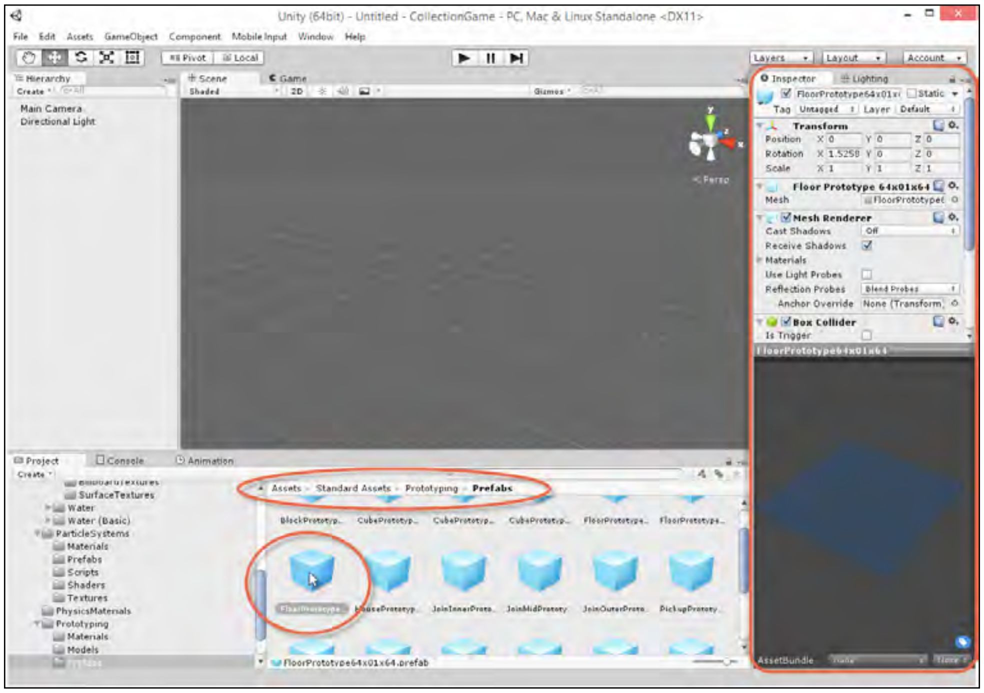


The Scene tab displays a preview of a 3D world

You can even select objects in the scene by clicking on their name inside the Hierarchy Panel

Next, let's add a floor to the scene. After all, the player needs something to stand on! We could build a floor mesh from scratch using third party modelling software, like Maya, 3DS Max or Blender. However, the Unity Standard Asset packages, which were imported earlier, contain floor meshes we can use. This is very convenient. These meshes are part of the *Prototyping* package. To access them via the Project Panel, open the *Standard Assets* folder by double-clicking it, and then access the *Prototyping* > *Prefabs* folder. From here, you can select objects and preview them from the Object Inspector. See Figure 1.16.

You *could* also quickly add a floor to the scene by choosing *GameObject* > *3D Object* > *Plane* from the application menu. But this adds just a dull, grey floor, which isn't very interesting. Of course, you *could* change its appearance. As we'll see later, Unity lets you do that. But, for this tutorial, we'll use a specifically modelled floor mesh via the Standard Assets Package, from the Project Panel…



Inspector | Lighting
FloorPrototype64x01x64 ☐ Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 1.5258 Y 0 Z 0
Scale X 1 Y 1 Z 1

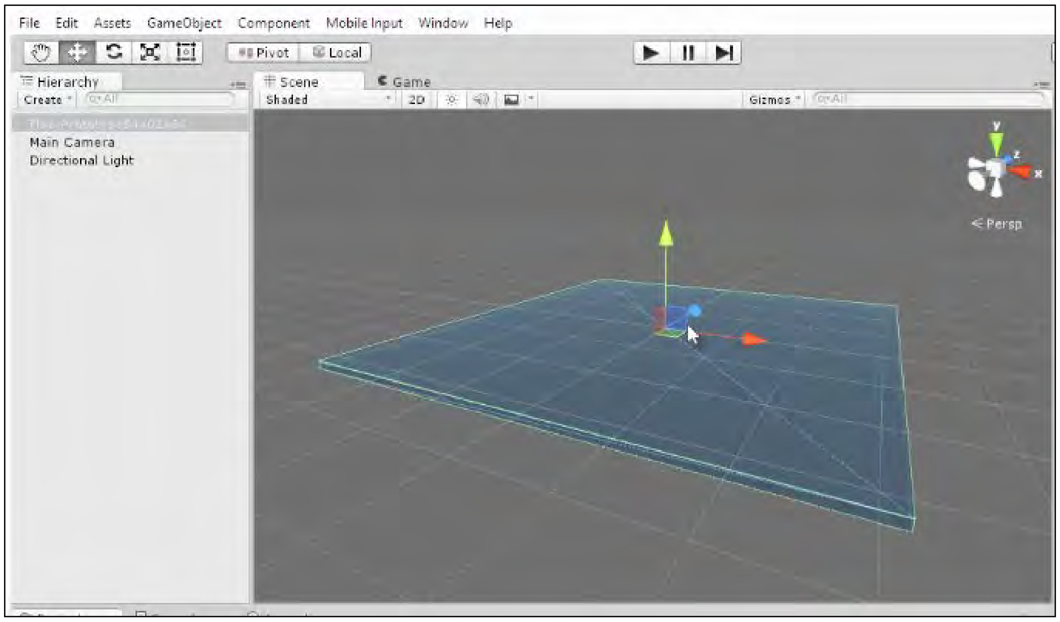
Floor Prototype 64x01x64
Mesh FloorPrototype

✓ Mesh Renderer
Cast Shadows Off
Receive Shadows ✓
Materials
Use Light Probes ☐
Reflection Probes Blend Probes
Anchor Override None (Transform)

✓ Box Collider
Is Trigger ☐

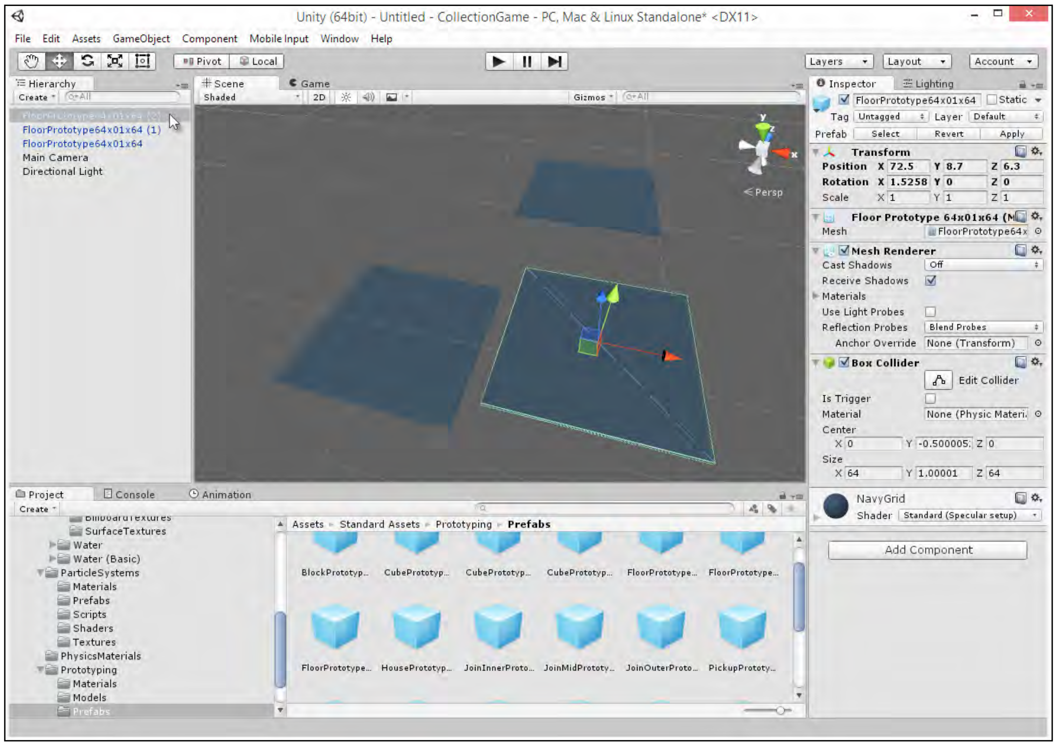
The StandardAssets/Protyping Package contains many meshes for quick scene building

The mesh named *FloorPrototype64x01x64* (As shown in Figure 1.16) is suitable as a floor. To add this mesh to the scene, simply drag and drop the object from the Project Panel into the Scene view, and then release the mouse. See Figure 1.17. When you do this, notice how the Scene view changes to display the newly added mesh within 3D space, and the mesh name also appears as a listing in the Hierarchy Panel.



Dragging and Dropping mesh assets from the Project Panel to the Scene view will add them to the scene

The floor mesh *asset* from the Project Panel has now been **instantiated** as a *GameObject* in the scene. This means a copy or clone of the mesh asset, based on the original in the Project Panel, has been added to the scene as a separate GameObject. The **Instance (**or GameObject**)** of the floor inside the scene still **Depends** on the floor asset in the Project Panel, however. Although, the Asset does not depend on the Instance. This means that, by deleting the floor *in the scene*, you will *not* delete the asset. But, if you delete the asset, you will delete or invalidate the GameObject. You can also create more floors in the Scene, if you want, by dragging and dropping the floor asset many times, from the Project Panel to the scene view. Each time a new instance of the floor is created in the scene as a separate and unique game object, although all the added instances will still depend on the single floor asset in the Project Panel. See Figure 1.18.


Unity (64bit) - Untitled - CollectionGame - PC, Mac & Linux Standalone* <DX11>
File Edit Assets GameObject Component Mobile Input Window Help
Pivot
Local
Layers
Layout
Account
Hierarchy
Create
Scene
Game
Shaded
2D
Gizmos
Persp
Inspector
Lighting
Project
Console
Animation
Add Component

Hierarchy: FloorPrototype64x01x64 (2), FloorPrototype64x01x64 (1), FloorPrototype64x01x64, Main Camera, Directional Light

Inspector: FloorPrototype64x01x64, Static; Tag Untagged, Layer Default; Prefab Select Revert Apply

Transform: Position X 72.5 Y 8.7 Z 6.3; Rotation X 1.5258 Y 0 Z 0; Scale X 1 Y 1 Z 1

Floor Prototype 64x01x64 (Mesh Filter): Mesh FloorPrototype64x

Mesh Renderer: Cast Shadows Off; Receive Shadows ✓; Materials; Use Light Probes ☐; Reflection Probes Blend Probes; Anchor Override None (Transform)

Box Collider: Edit Collider; Is Trigger ☐; Material None (Physic Materi; Center X 0 Y -0.500005 Z 0; Size X 64 Y 1.00001 Z 64

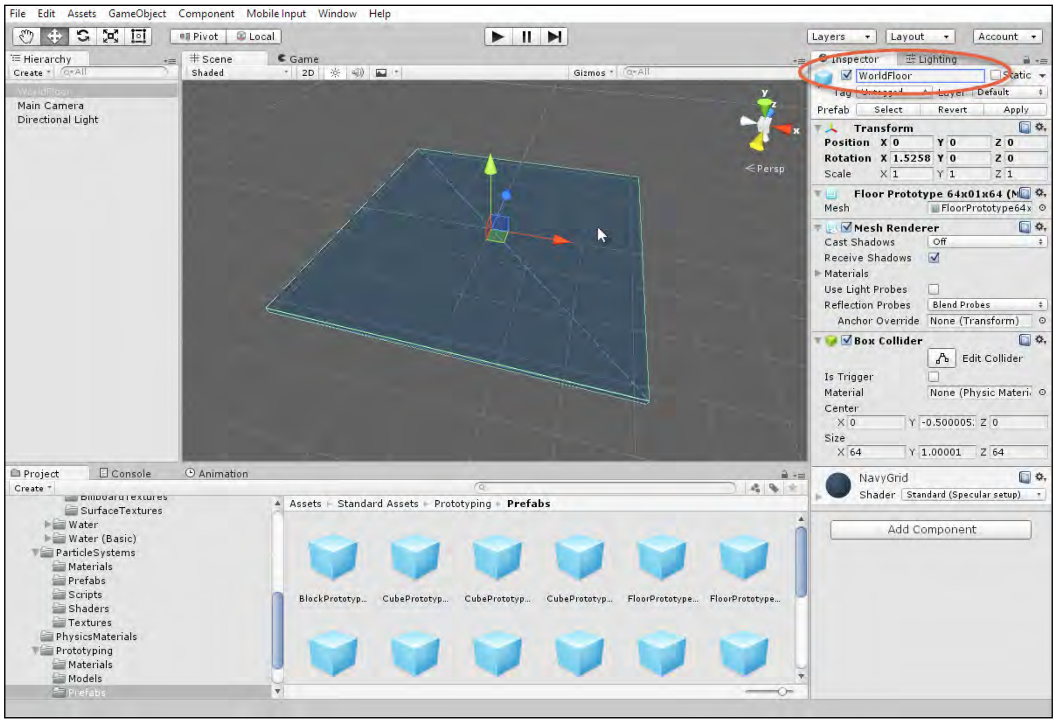
NavyGrid: Shader Standard (Specular setup)

Project: BillboardTextures, SurfaceTextures, Water, Water (Basic), ParticleSystems, Materials, Prefabs, Scripts, Shaders, Textures, PhysicsMaterials, Prototyping, Materials, Models, Prefabs

Assets ▸ Standard Assets ▸ Prototyping ▸ **Prefabs**: BlockPrototyp..., CubePrototyp..., CubePrototyp..., CubePrototyp..., FloorPrototype..., FloorPrototype..., FloorPrototype..., HousePrototyp..., JoinInnerProto..., JoinMidPrototy..., JoinOuterProto..., PickupPrototy...

Adding multiple instances of the floor mesh to the scene...

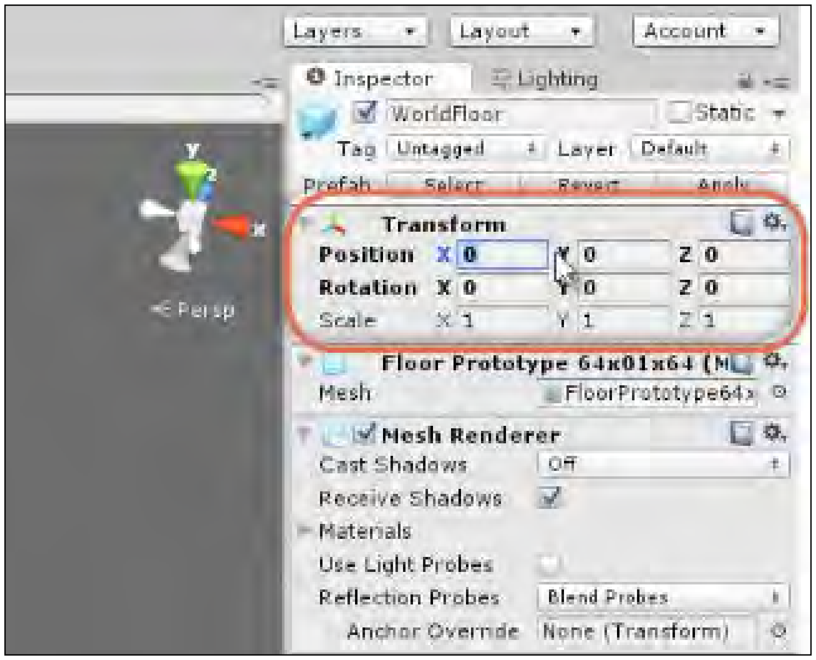
We don't actually need the duplicate floor pieces, however. So let's delete them. Just click the duplicates in the Scene view and then press *Delete* on the keyboard to remove them. Remember, you can also select and delete objects by clicking their name inside the hierarchy panel and pressing *Delete*. Either way, this leaves us with a single floor piece and a solid start to building our scene. One remaining problem, though, concerns the floor and its name. By looking carefully in the Hierarchy panel, we see the floor name is *FloorPrototype64x01x64*. This name is long, obtuse and unwieldy. We should change it to something more manageable and meaningful. This is not technically essential but is good practice to keep our work clean and organized. To rename an object, first select it and then enter a new name into the name field, inside the Object Inspector. I'll rename it to *WorldFloor* See Figure 1.19.



Renaming the floor mesh

Transformations and Navigation

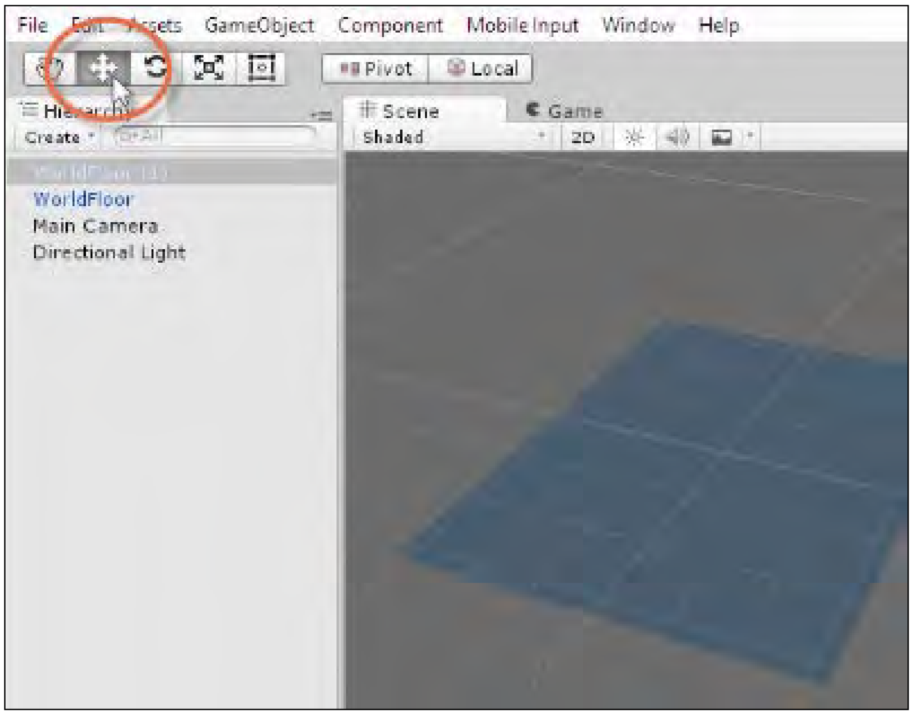
A scene with a floor mesh has been established, but this alone is uninteresting. We need to add more, such as buildings, stairs, columns, and perhaps more floor pieces. Otherwise, they'd be no world for the player to explore. Before building upon what we've got however, let's make sure the existing floor piece is centered at the world origin. Every point and location within a scene is uniquely identified by a **Coordinate**, measured as an (X, Y, Z) offset from the world center (**Origin**). The current position for the selected object is always visible from the Object Inspector. In fact, the position, rotation and scale of an object are grouped together under a category (**Component**) called **Transform**. The **Position** indicates how far an object should be moved in three axes from the world center. The **Rotation** indicates how much an object should be turned or rotated around its central axes. And **Scale** indicates how much an object should be shrunk or expanded to smaller or larger sizes. A default Scale of 1 means an object should appear at normal size; 2 means twice the size, and 0.5 means half the size, and so on. Together, the Position, Rotation and Scale of an object constitute its Transformation. To change the position of the selected object, you can simply type new values into the X, Y and Z fields for Position. To move an object to the world center, simply enter (0, 0, and 0). As shown in Figure 1.20.



WorldFloor Static
Tag Untagged Layer Default
Prefab Select Revert Apply
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Floor Prototype 64x01x64 (M
Mesh FloorPrototype64x
Mesh Renderer
Cast Shadows Off
Receive Shadows ✓
Materials
Use Light Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)

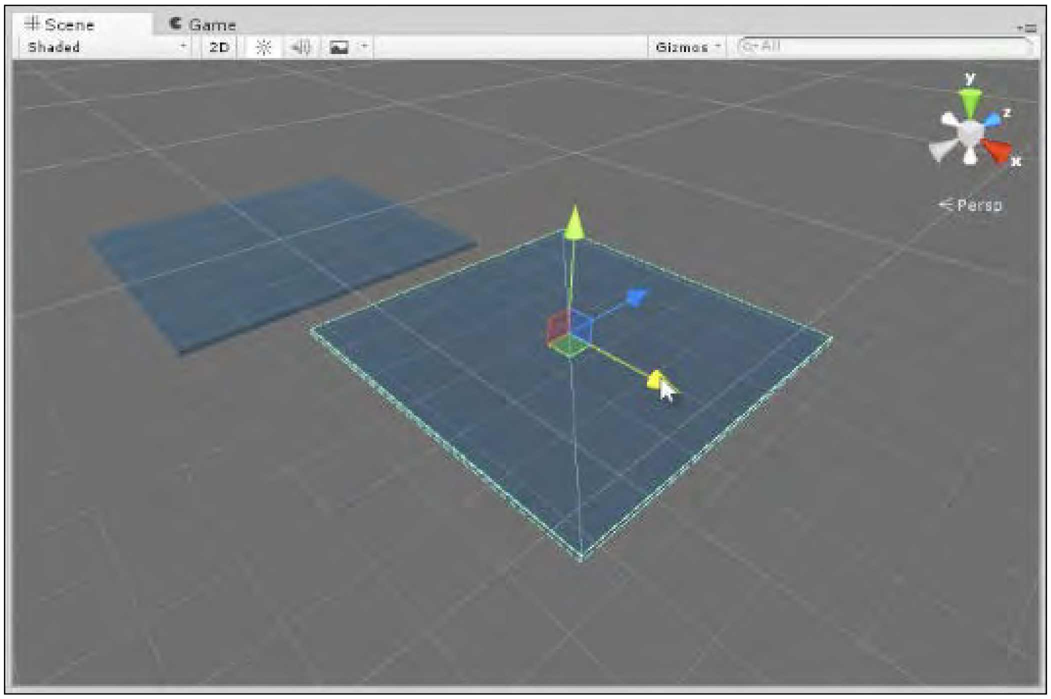
Centering an Object to the World Origin

Setting the position of an object, as we've done here, by typing numerical values is acceptable and appropriate for specifying exact positions. However, it's often more intuitive to move objects using mouse-based controls. To do that, let's add a second floor piece and position it away from the first instance. Drag and drop a floor piece from the Project Panel into the Scene to create a second floor GameObject. Then click the new floor piece to select it, and then switch to the **Translate** tool. To do that, press *W* on the keyboard, or click the Translate tool icon from the toolbar, at the top of the Editor interface. The Translate tool allows you to reposition objects in the scene. See Figure 1.21.



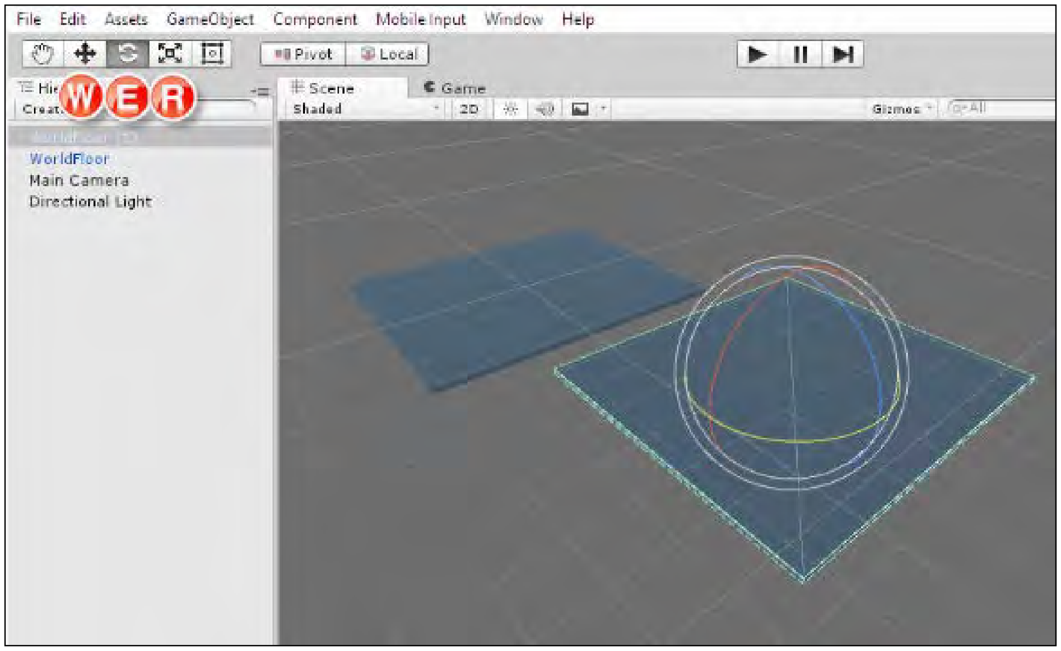
Accessing the Translate Tool

When the Translate tool is active and an object is selected, a **Gizmo** appears centered on the Object. The Translate Gizmo appears as three colored and perpendicular axes; Red, Green and Blue corresponding to X, Y, and Z respectively. To move an object, hover your cursor over one of the three axes, and then click and hold the mouse while moving it to slide the object in that direction. You can repeat this process as often as needed to ensure your objects are positioned where you need them to be. Use the Translate tool to move the second floor piece away from the first. See Figure 1.22.



Translate an object using the Translate Gizmo

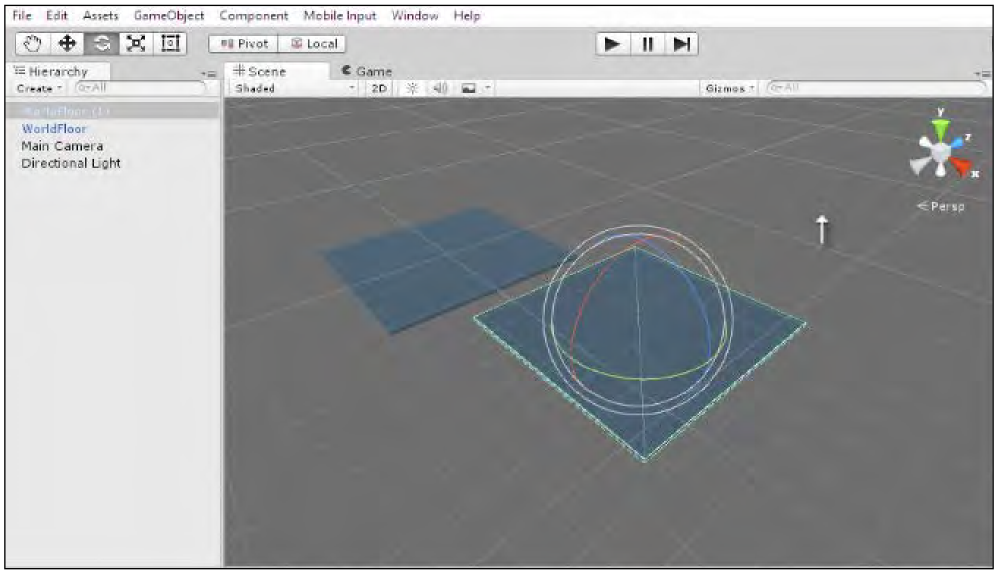
You can also Rotate and Scale Objects using the mouse, as with Translate. Press *E* to access the Rotate tool, or *R* to access the Scale tool; or you can activate these tools using their tool bar icons respectively from the top of the Editor. When these tools are activated, a Gizmo appears centered on the object, and you can click and drag the mouse over each specific axis to rotate or scale objects as needed. See Figure 1.23.



Accessing the Rotate and Scale tools...

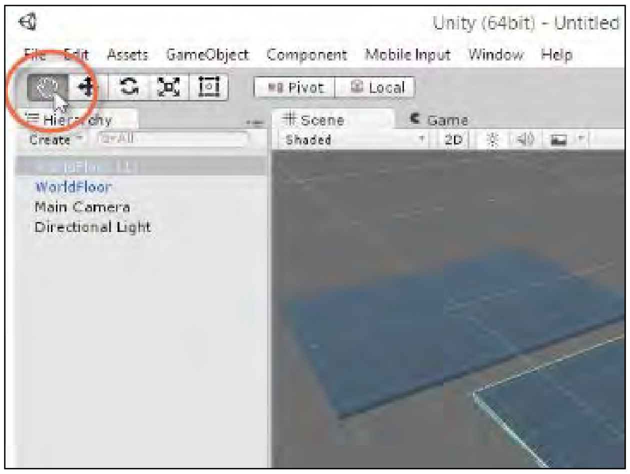
Being able to Translate, Rotate and Scale objects quickly through mouse and keyboard combinations is very important when working inside Unity. For this reason, make using the keyboard shortcuts a habit, as opposed to accessing the tools continually from the tool bar. However, in addition to moving, rotating and scaling objects, you'll frequently need to move around yourself inside the Scene view, to see the world from different positions, angles and perspectives. That is, you'll frequently need to reposition the scene preview camera inside the world. You'll want to zoom in and zoom out of the world to get a better view of objects, and to change your viewing angle to see how objects align and fit together properly. To do this, you'll need to make extensive use of both the keyboard and mouse together.

To **Zoom** closer or further from the object you're looking at, simply scroll the mouse wheel up or down. Up zooms in, and Down zooms out. See Figure 1.24.



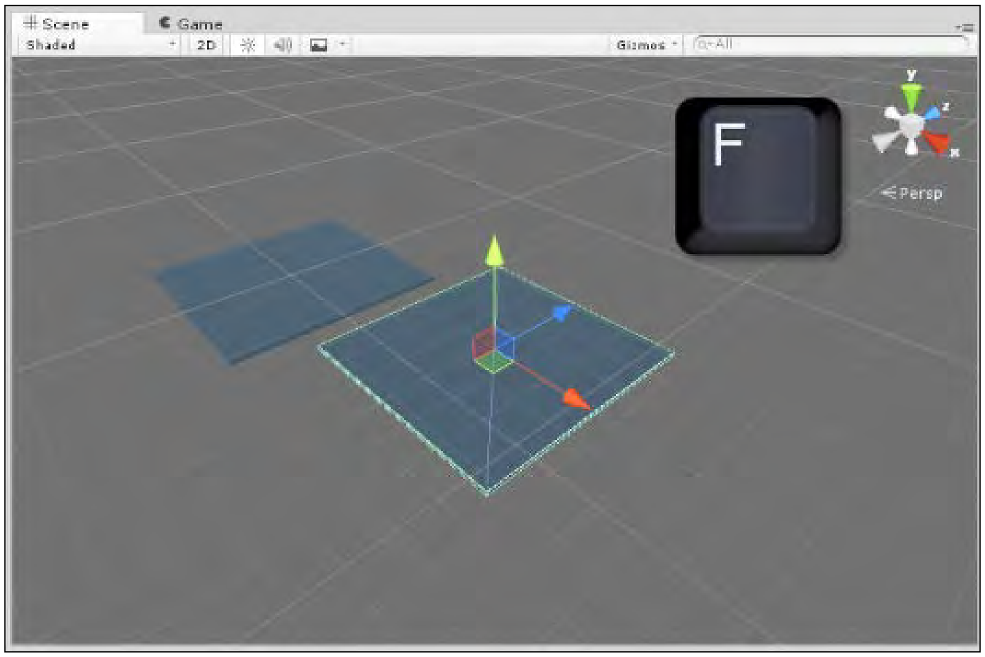
Zooming in and out...

To **Pan** the Scene view left or right, or up or down, hold down the middle mouse button while moving the mouse in the appropriate direction. Alternatively, you can access the Pan tool from the application tool bar (or Press *Q* on the keyboard) and then simply click and drag inside the Scene view while the tool is active. Pan does not zoom in or out; it simply slides the camera left or right, or up or down.



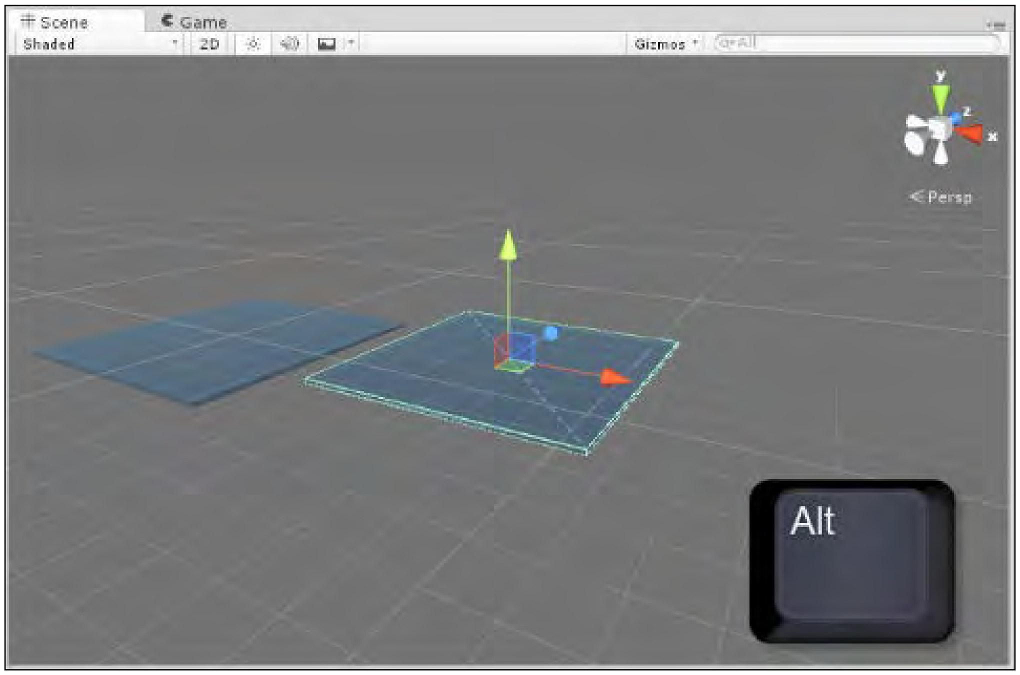
Accessing the Pan Tool

Sometimes while building levels you'll lose sight entirely of the object you need. For example, your viewport camera could be focusing on a completely different place from the object you really want to click or see. In this case, you'll often want to shift the viewport camera automatically, to focus on that specific object. Specifically, you'll want to reposition and rotate the viewport as necessary, to bring a desired object to the center of view. To do this automatically, select the object to **Focus** on (or **Frame**) by clicking its name from the Hierarchy Panel. And then press the *F* key on the keyboard. Alternatively, you can double click its name in the Hierarchy Panel. See Figure 1.26.



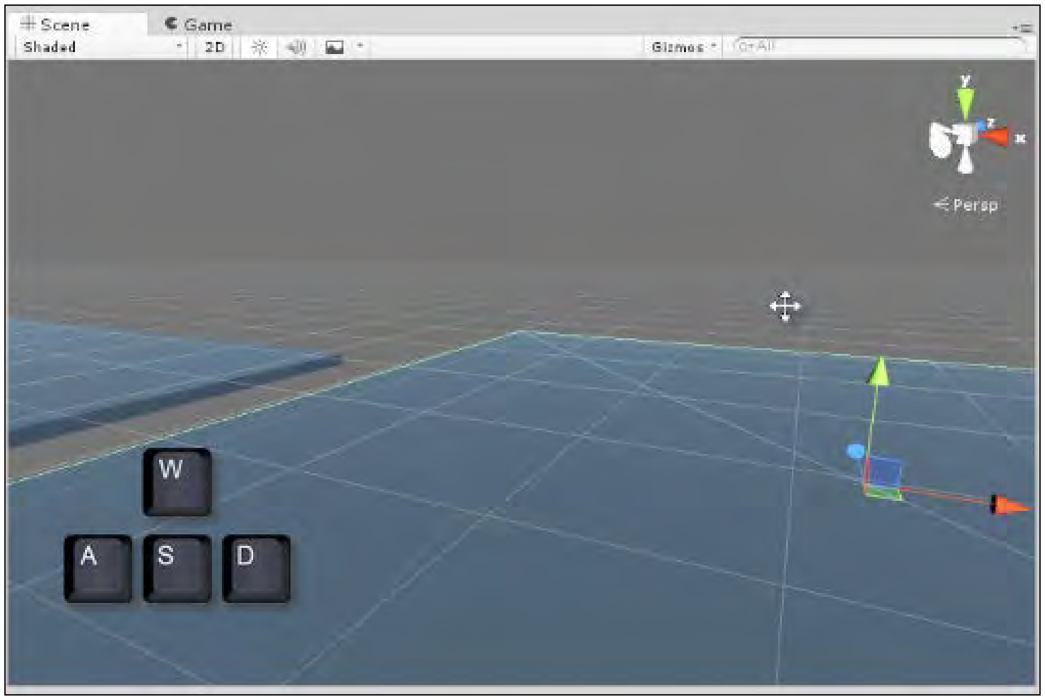
Framing a selected object

After Framing an Object, you'll often want to *rotate around* it, to quickly and easily view it from all important angles. To achieve this, hold down the *Alt* key on the keyboard while clicking and dragging the mouse to rotate the view. See Figure 1.27.



Rotating around the Framed Object

Lastly, it's helpful to navigate a level in the Scene view using First Person Controls. That is, controls which mimic how first-person games are played. This helps you experience the scene a more personal and immersive level. To do this, hold down the right-mouse button, and use the *WASD* keys on the keyboard to control forwards, backwards and strafing movement. And movement of the mouse controls head orientation. You can also hold down the *Shift* key while moving to increase movement speed. See Figure 1.28.

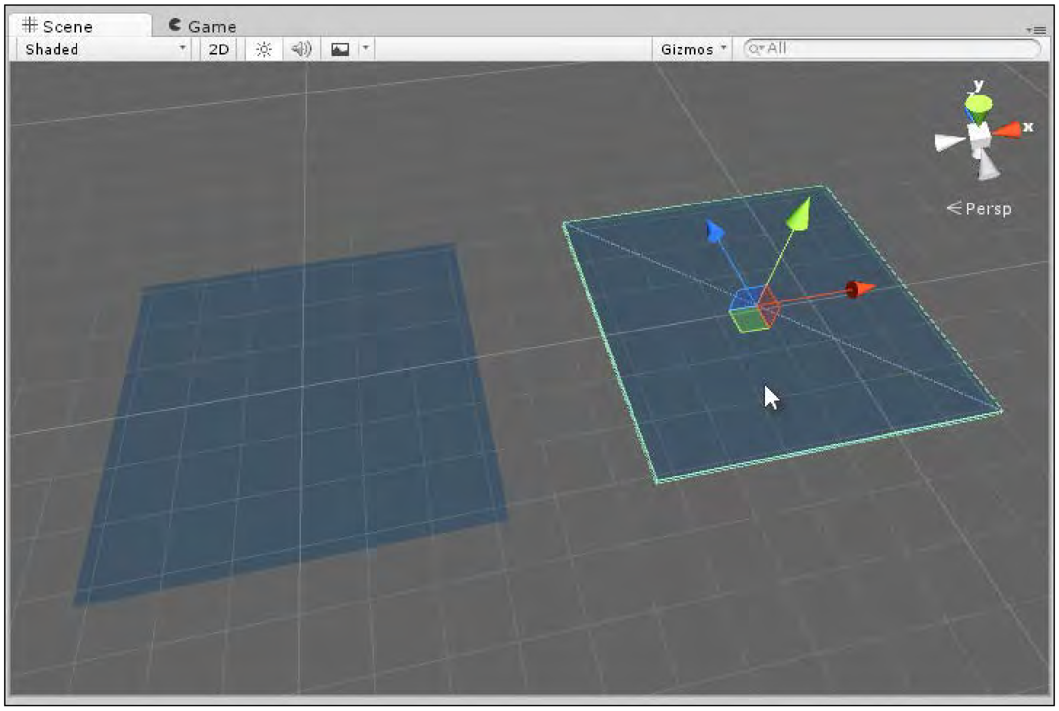


Using First Person Controls...

The great thing about learning the versatile Transformation and Navigation controls is that, on understanding them, you can move and orient practically any object in any way, and you can move and view the world from almost any position and angle. Being able to do this is critically important for building quality levels quickly. All of these controls, along with some others we'll soon see, will be used frequently throughout this book for creating scenes and for working within Unity generally.

Scene Building

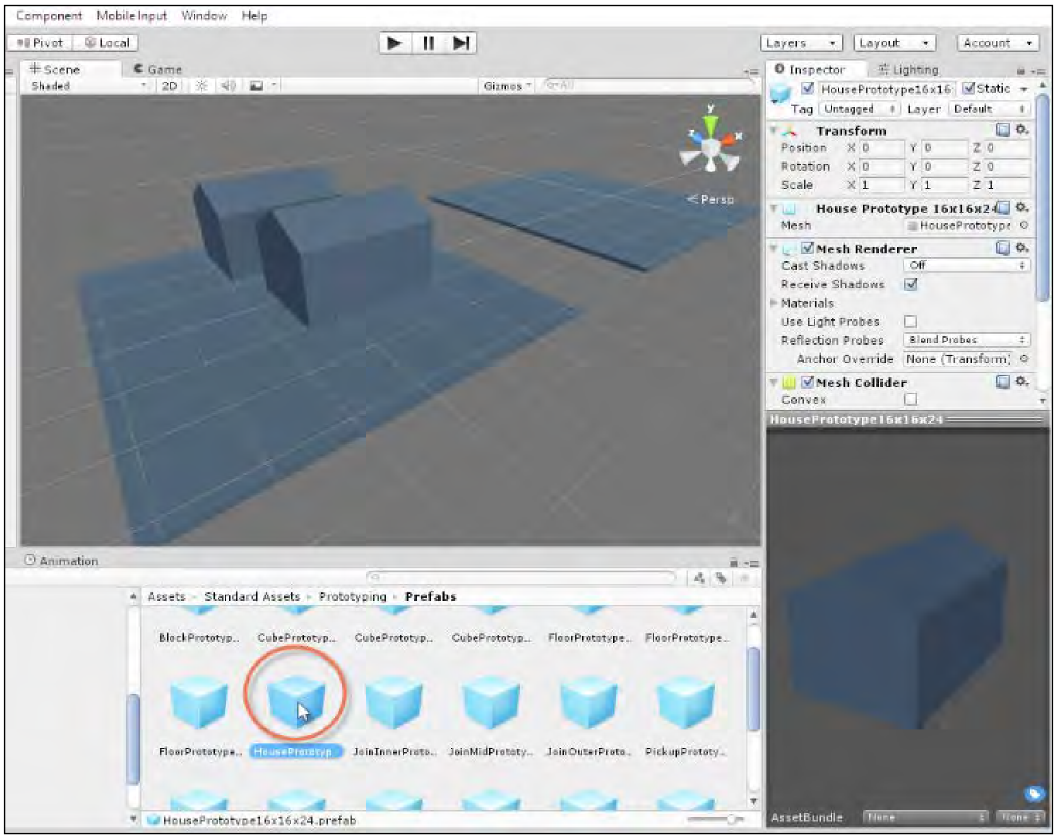
Now we've seen how to transform objects and navigate the scene viewport successfully, let's proceed to complete our first level for the coin collection game. Let's separate the two floor meshes apart in space, leaving a gap between them that we'll fix by creating a bridge, which the player will be able to cross, moving between the floor spaces like islands. We can use the Translate tool (W) to move objects around. See Figure 1.29.



Separating the floor meshes into islands

If you want to create more floor objects, you can use the method we've seen already by dragging and dropping the mesh asset in the Project Panel into the Scene Viewport. Or, alternatively, you can duplicate the selected object in the viewport by pressing *Ctrl* + *D* on the keyboard. Both methods produce the same result.

Next we'll add some props and obstacles to the scene. Drag and drop some house objects onto the floor. The house object (*HousePrototype16x16x24*) is found inside the folder *Assets* > *Standard Assets* > *Prototyping* > *Prefabs*. See Figure 1.30.



HousePrototype16x16 Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

House Prototype 16x16x24
Mesh HousePrototype

Mesh Renderer
Cast Shadows Off
Receive Shadows ✓
Materials
Use Light Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)

Mesh Collider
Convex

HousePrototype16x16x24

AssetBundle None None

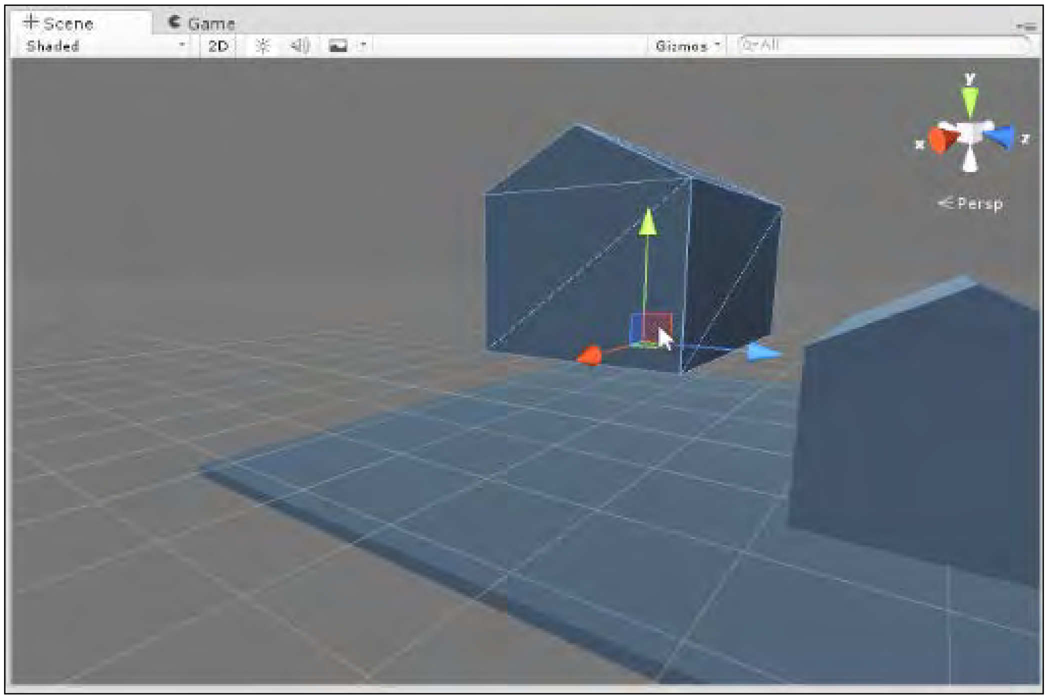
Assets > Standard Assets > Prototyping > **Prefabs**

HousePrototype16x16x24.prefab

Adding house props to the scene

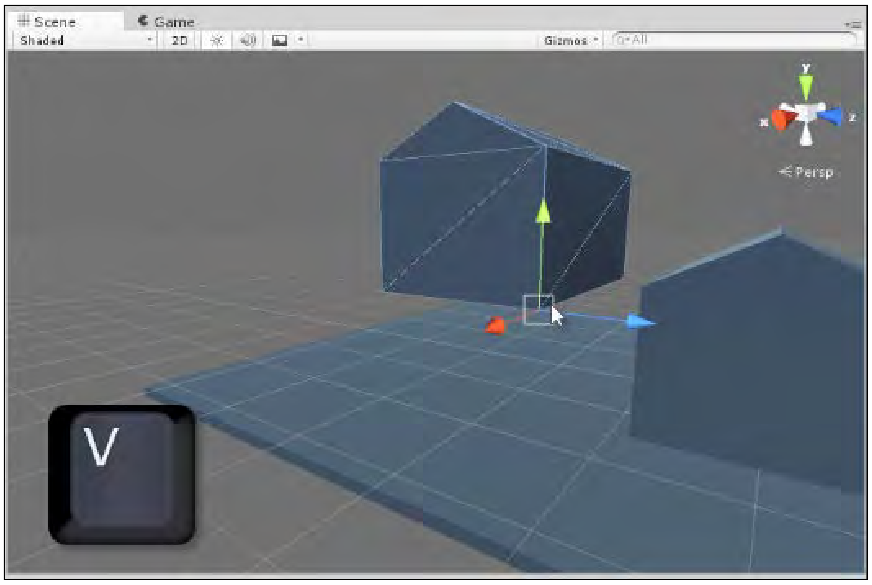
On dragging and dropping the house into the scene, it may align to the floor nicely with the bottom against the floor, or it may not align like that. If it does, that's splendid and great luck! But we shouldn't rely on luck every time, because we're professional game developers! Thankfully, we can make any two mesh objects align easily inside Unity, by using **Vertex Snapping**. This feature works by forcing two objects into positional alignment within the scene, by overlapping their vertices at a specific and common point.

For example, consider Figure 1.31. Here, a house object hovers awkwardly above the floor, and we naturally want it to align level with the floor, and perhaps over to the floor corner. To achieve this, start by selecting the house object (click it, or select it from the Hierarchy Panel). The object to be selected is the one that *should move* to align, and not the destination (which is the floor), which should remain in place.



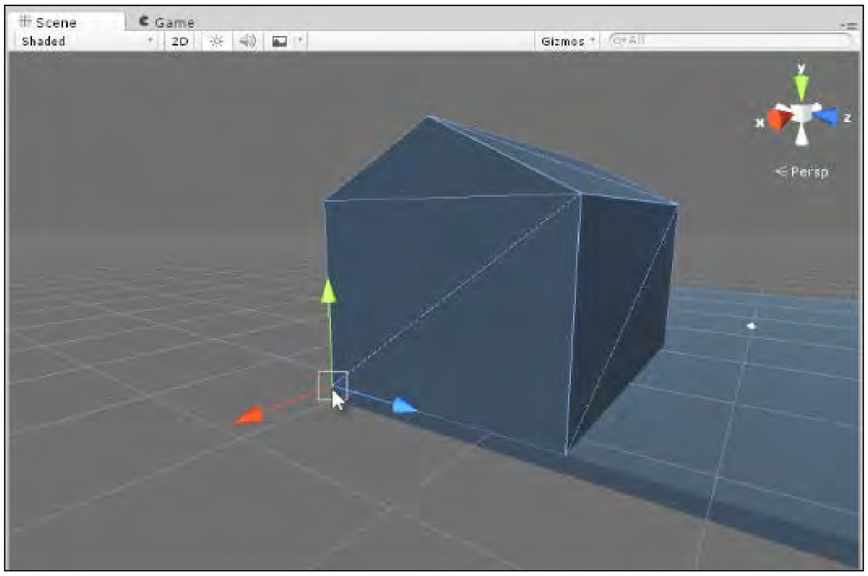
Misaligned objects can be snapped into place with Vertex Snapping...

Next, activate the Translate tool (*W*) and hold down the *V* key, for Vertex Snapping. With *V* held down, move the cursor around and see how the Gizmo cursor sticks to the nearest vertex of the selected mesh. See Figure 1.32. Unity is asking you to pick a source vertex for snapping.



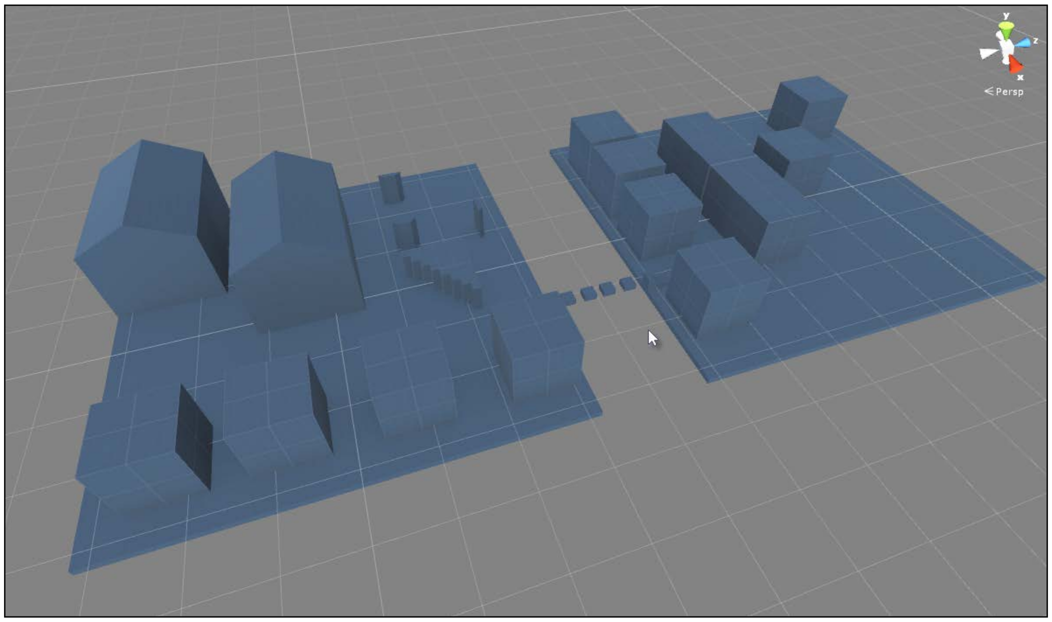
Hold down V to activate Vertex Snapping

With *V* held down, move the cursor to the bottom-corner of the house, and then click and drag from the corner to the floor mesh corner. The house will then snap align to the floor, corner to corner. When aligned this way, release the *V* key and now the two meshes are aligned exactly at the vertices. See Figure 1.33.



Align together two meshes by vertices

Now you can assemble a complete scene using the mesh assets included in the Prototyping package. Drag and drop props into the scene, and by using Translate, Rotate and Scale you can reposition, re-align and rotate those objects; and by using Vertex Snapping you can align them wherever you need. Give this some practice. See Figure 1.34 for the scene arrangement I made using only these tools and assets.

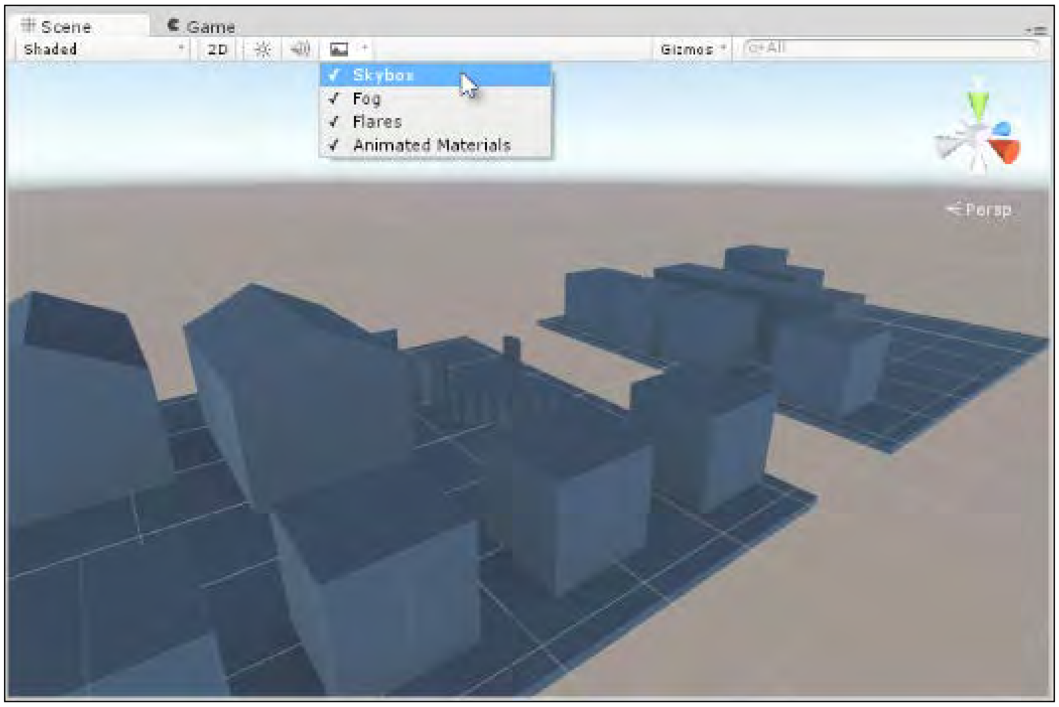


Building a complete level...

Lighting and Sky

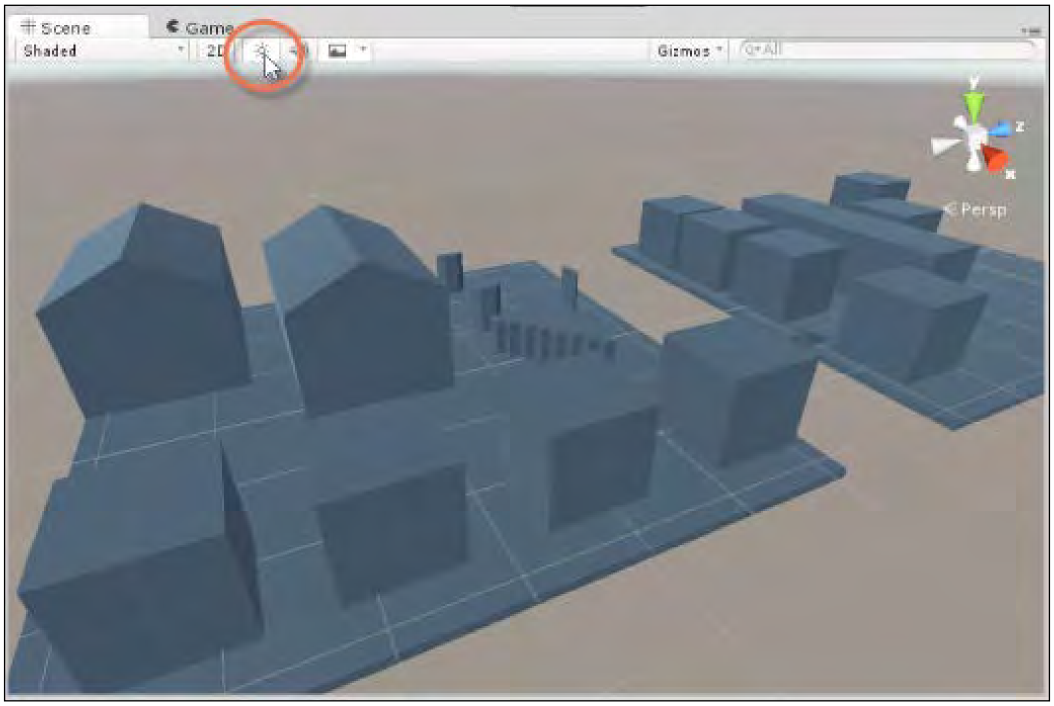
The basic level has been created, in terms of architectural models and layout; and this was achieved using only a few mesh assets and some basic tools. Nevertheless, these tools are powerful and offer us a multitude of combinations and options for creating great variety and believability in game worlds. One important ingredient is missing for us, however. That ingredient is Lighting. You'll notice from Figure 1.34 that everything looks relatively flat, with no highlights, shadows or light or dark areas. This is because scene lighting is not properly configured for best results, even though we already have a light in the scene, which was created initially by default.

Let's start setting the scene for the coin collection game by enabling the Sky, if it's not already enabled. To do that, click the *Extras* drop-down menu from the top toolbar in the Scene viewport. From the context menu, select *Skybox* to enable Skybox viewing. A Skybox simply refers to a large cube that surrounds the whole scene. Each interior side has a continuous texture (image) applied to simulate the appearance of a surrounding sky. For this reason, clicking the Skybox option displays a default sky inside the Scene viewport. See Figure 1.35.



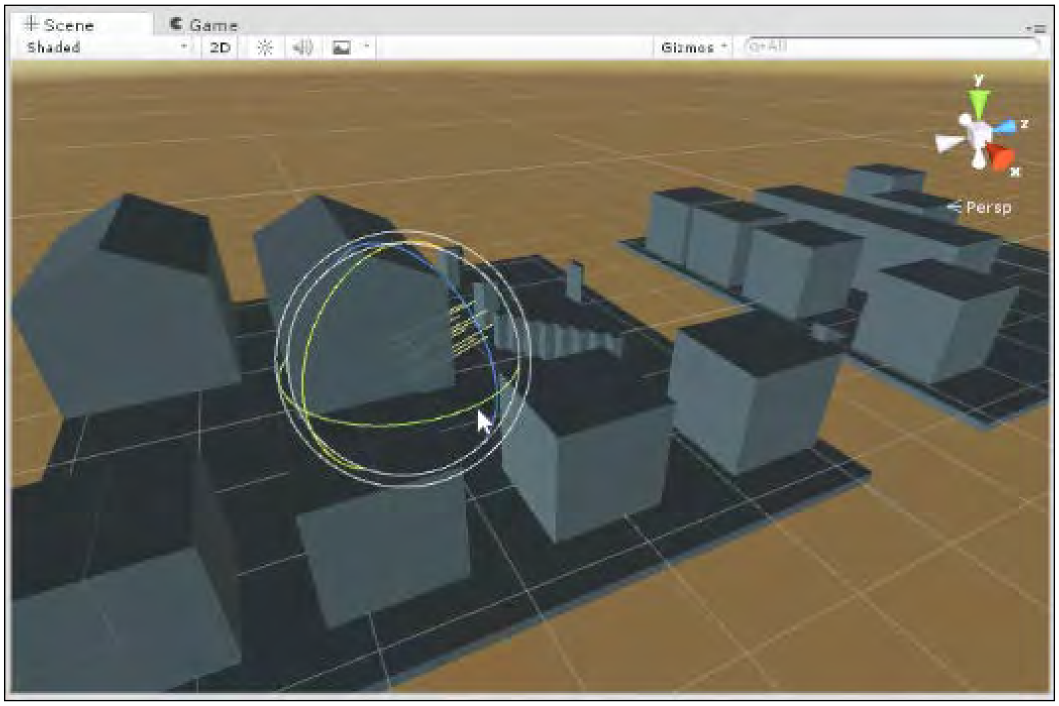
Enabling the sky...

Now, although the Skybox is now enabled and the scene looks better than before, it's still not being illuminated properly- the objects lack shadows and highlights. To fix this, be sure that Lighting is enabled for the scene, by toggling on the Lighting icon at the top of the Scene viewport. See Figure 1.36. This setting is for *display* purposes only. It only affects whether lighting effects are *shown* in the Scene viewport, and *not* whether lighting is truly enabled for the final game.



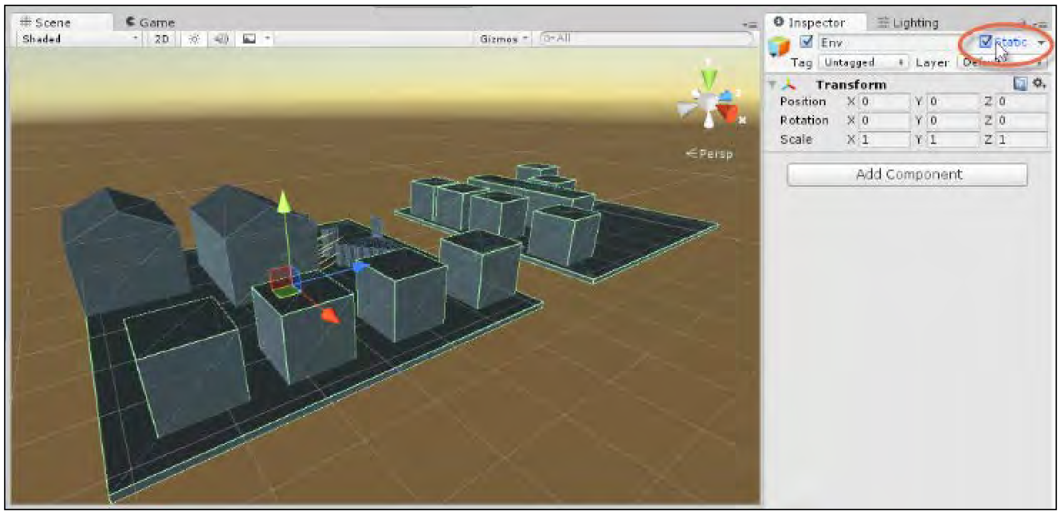
Enabling Scene Lighting inside the Scene Viewport

Enabling Lighting display for the viewport will result in some differences to the scene appearance, and, again, the scene should look better than before. You can confirm that scene lighting is taking affect by selecting the Directional Light from the Hierarchy Panel and rotating it. Doing this controls the time of day; rotating the light cycles between day and night, changing the light intensity and mood. This changes how models are rendered. See Figure 1.37.



Rotating the Scene Directional Light changes the time of day

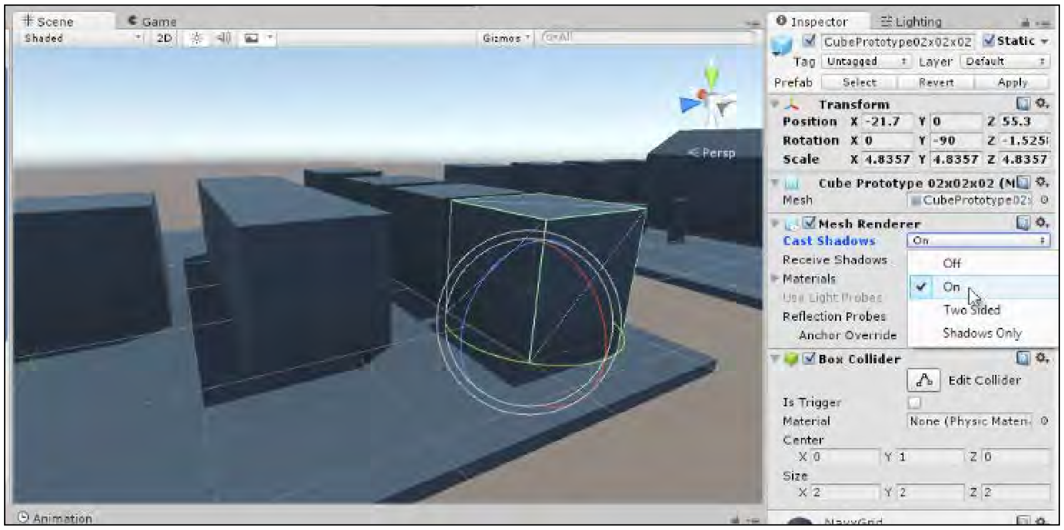
Let's undo any rotations to the directional light, by pressing *Ctrl* + *Z* on the keyboard. To prepare for final and optimal lighting, all non-movable objects in the scene (like walls, floors, chairs, tables, ceilings, grass, hills, towers and more) should be marked as *Static*. This signifies to Unity that the objects will never move, no matter what happens during gameplay. By marking non-movable objects *ahead of time*, you can help Unity optimize the way it renders and lights a scene. To mark objects as static, simply select all non-movable objects (which includes practically the entire level so far), and then enable the *Static* check box via the Object Inspector. Note: you don't need to enable the *Static* setting for each object separately. Rather, by holding down the *Shift* key while selecting objects, you can select multiple objects together, allowing you to adjust their properties as a batch through the Object Inspector. See Figure 1.38.



Enabling the Static option for multiple non-movable Objects improves lighting and performance

When you enable the Static check box for geometry, Unity auto-calculates scene lighting in the background- effects such as shadows, indirect-illumination and more. It generates a batch of data called the **GI Cache**, featuring **Light Propagation Paths**, which instructs Unity how light rays should bounce and move around the scene to achieve greater realism. Even so, enabling the static check box as we've done, still won't produce cast shadows for objects, and this seriously detracts from realism. This happens because most mesh objects have the *Cast Shadows* option disabled.

To fix this, select all meshes in the scene. Then, from the Object Inspector, click the *Cast Shadows* check box from the *Mesh Renderer* component, and choose the option *On* from the context menu. When you do this, all mesh objects should be casting shadows. See Figure 1.39.



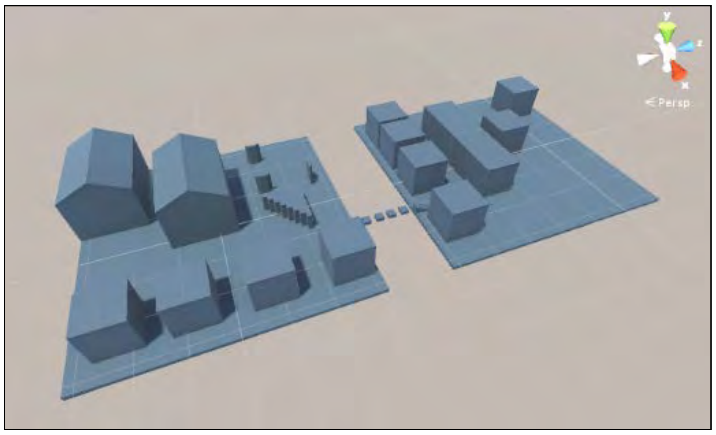
CubePrototype02x02x02 Static
Tag Untagged Layer Default
Prefab Select Revert Apply
Transform
Position X -21.7 Y 0 Z 55.3
Rotation X 0 Y -90 Z -1.525
Scale X 4.8357 Y 4.8357 Z 4.8357
Cube Prototype 02x02x02 (M
Mesh CubePrototype02
Mesh Renderer
Cast Shadows On
Receive Shadows
Materials
Use Light Probes
Reflection Probes
Anchor Override
Off
✓ On
Two Sided
Shadows Only
Box Collider
Edit Collider
Is Trigger
Material None (Physic Mater
Center X 0 Y 1 Z 0
Size X 2 Y 2 Z 2

Enabling Cast Shadows from the Mesh Renderer Component

And voila! Your meshes now cast shadows. Splendid work: in reaching this far you've created a new project, populated a scene with meshes, and successfully illuminated them with directional lighting. That's excellent. But, it'd be even better if we could explore our environment in First Person mode. And we'll see how next.

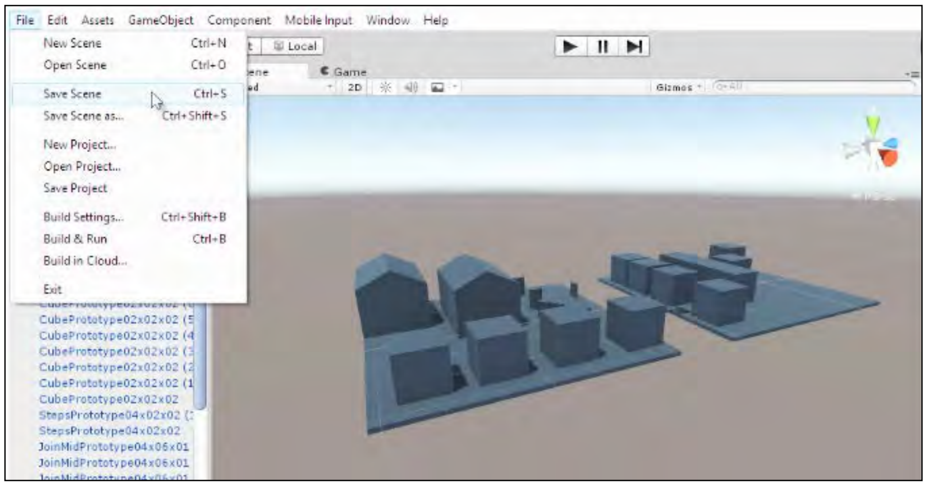
Play Testing and the Game Tab

The environment created thus far for the coin collection game has been assembled using only mesh assets included with the native Prototyping Package. My environment, as shown in Figure 1.40, features two main floor islands with houses, and the islands themselves are connected together by a stepping stone bridge. Your version may be slightly different, and that's fine.



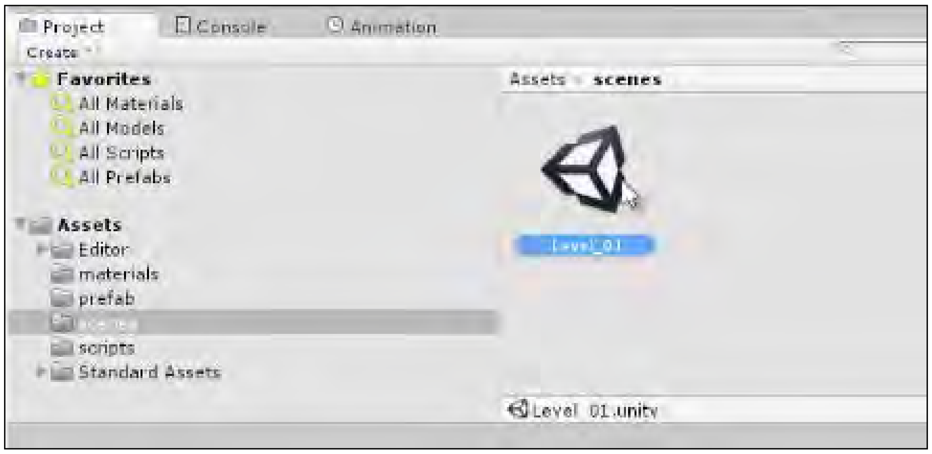
The Scene created so far contains two island areas...

Over all, the scene is good work. It's well worth saving. To save the scene, press *Ctrl+S* on the keyboard, or else choose *File* > *Save Scene* from the application menu. See Figure 1.41. If you're saving the scene for first time, Unity displays a pop-up *Save* dialog, prompting you to name the scene descriptively (I called it: *Level_01*).



Saving a Scene...

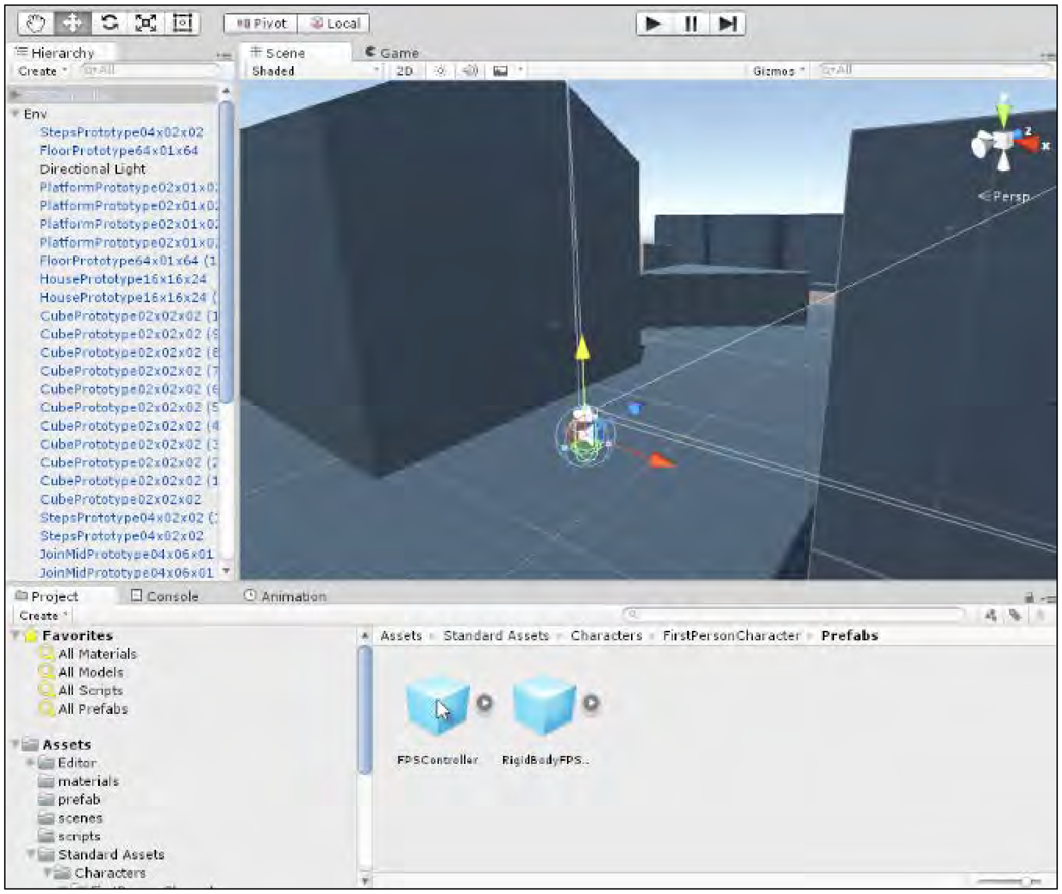
After saving the scene it becomes a *Scene Asset* of the Project, and appears in the Project Panel. See Figure 1.42. This means the scene is now a genuine and integral part of the project, and not just a temporary 'work-in-progress' as it was before. *Notice* also that saving a scene is conceptually *different* from saving a project. For example, the application menu has entries for *Save Scene* and *Save Project*. Remember, a *Project* is a collection of files and folders, including assets and scenes. A scene, by contrast is one asset *within* the project, and represents a complete 3D map that may be populated by other assets, such as meshes, textures and sounds.



Saved scenes are added as assets within your project

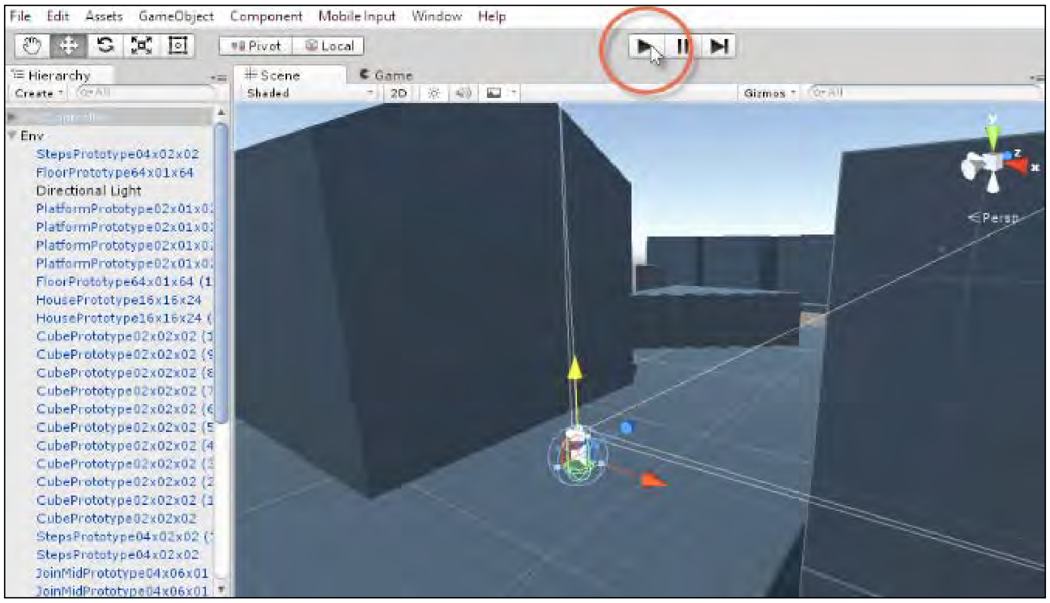
See from Figure 1.42 that I've saved my scene inside a folder, named *Scenes*. Folders can be created in your project by right-clicking on any empty area in the Project Panel, and choosing *New Folder* from the context menu. Or else choose *Assets* > *Create* > *Folder* from the application menu. You can easily move and rearrange assets among folders by simply dragging and dropping them.

Now, the level as it stands contains nothing really 'playable'. It's simply a static, lifeless and non-interactive 3D environment made using the Editor tools. Let's correct that by making our scene 'playable'; allowing the player to wander around and explore the world in first person mode, controlled using the standard WASD keys on the keyboard. To achieve this, we'll add a First Person Character Controller to the scene. This is a ready-made asset, included with Unity, which contains everything necessary to create quick and effective first person controls. Open the folder *Standard Assets* > *Characters* > *FirstPersonCharacter* > *Prefabs*. Then drag and drop the *FPSController* asset from the Project Panel into the Scene. See Figure 1.43.



Adding an FPS Controller to the Scene...

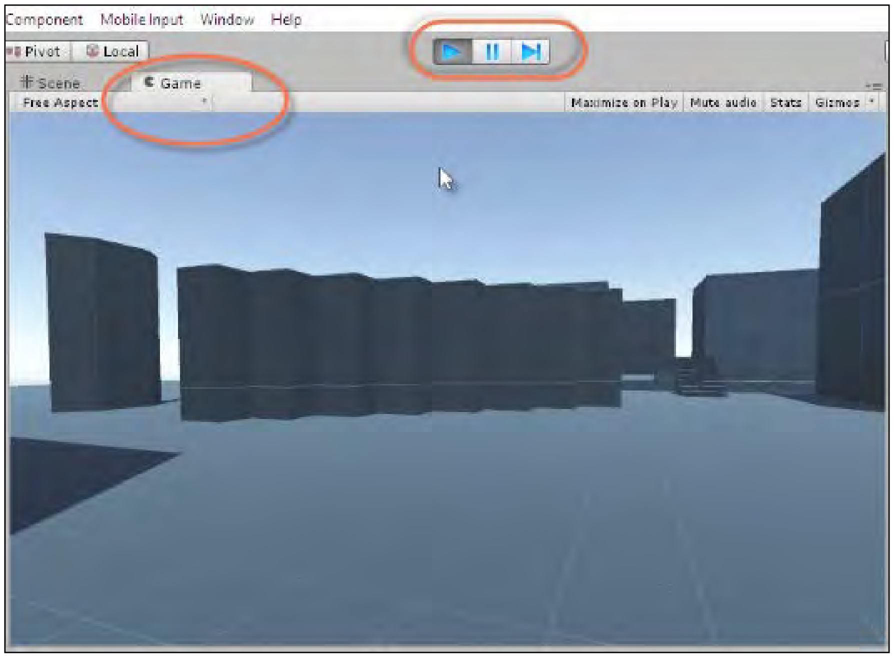
After adding the First Person Controller, click the *Play* button from the Unity tool bar to play test the game in First Person Mode. See Figure 1.44.



Unity scenes can be play tested by clicking the Play button from the toolbar

On clicking Play, Unity automatically switches from the *Scene* tab to the *Game* tab. As we've seen, the scene tab is a 'Director Eye View' of the active scene; it's where a scene is edited, crafted and designed. In contrast, the Game tab is where the active scene is played and tested, from the perspective *of the gamer*. From this view, the scene is displayed through the main game camera.

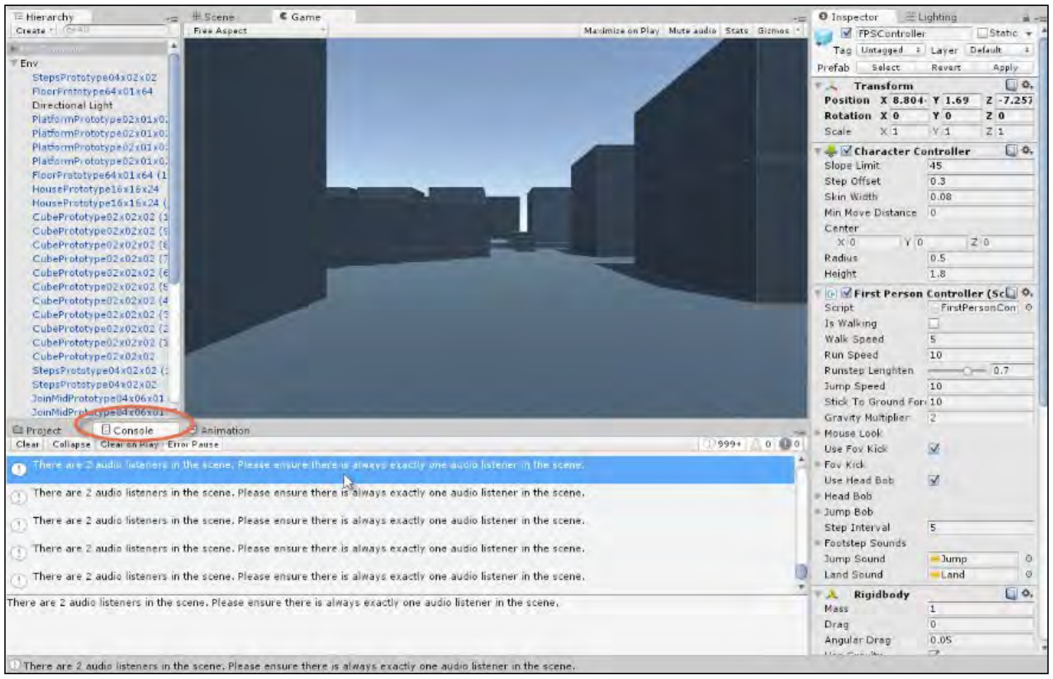
While Play mode is active, you can play test your game using the default game controls, provided the Game tab is 'in focus'. The first person controller uses the WASD keys on the keyboard, and mouse movement controls head orientation. See Figure 1.45.



Play testing levels inside the Game Tab

You can switch back to the Scene tab while in Play mode. And you can even edit the scene, and change and move and delete objects there too! However, any and all scene changes made during Play mode will automatically revert back to their original settings when Play mode ends. This behavior is intentional. It lets you edit properties during gameplay to observe their effects and debug any issues, without permanently changing the scene.

Congratulations! Your level should now be walkable in first person mode. When completed, you can easily stop playback by clicking the *Play* button again, or by pressing *Ctrl* + *P* on the keyboard. Doing this will return you to the *Scene* tab. You should notice that, on playing the level with a first person controller, you receive an information message printed to the **Console Window**. By default, this Window appears at the bottom of the Unity Editor, docked beside the Project Panel. This Window is also accessible manually from the application menu *Window* > *Console*. The Console Window is where all encountered errors or warnings are displayed for your review, as well as information messages. Errors are printed in red and Warnings in yellow, and information messages appear as a default grey. Sometimes a message appears just once, or sometimes it appears many times repeatedly. See Figure 1.46.



```
There are 2 audio listeners in the scene. Please ensure there is always exactly one audio listener in the scene.
There are 2 audio listeners in the scene. Please ensure there is always exactly one audio listener in the scene.
There are 2 audio listeners in the scene. Please ensure there is always exactly one audio listener in the scene.
There are 2 audio listeners in the scene. Please ensure there is always exactly one audio listener in the scene.
There are 2 audio listeners in the scene. Please ensure there is always exactly one audio listener in the scene.

There are 2 audio listeners in the scene. Please ensure there is always exactly one audio listener in the scene.

There are 2 audio listeners in the scene. Please ensure there is always exactly one audio listener in the scene.
```

FPSController Static
Tag Untagged Layer Default
Prefab Select Revert Apply

Transform
Position X 8.804 Y 1.69 Z -7.257
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

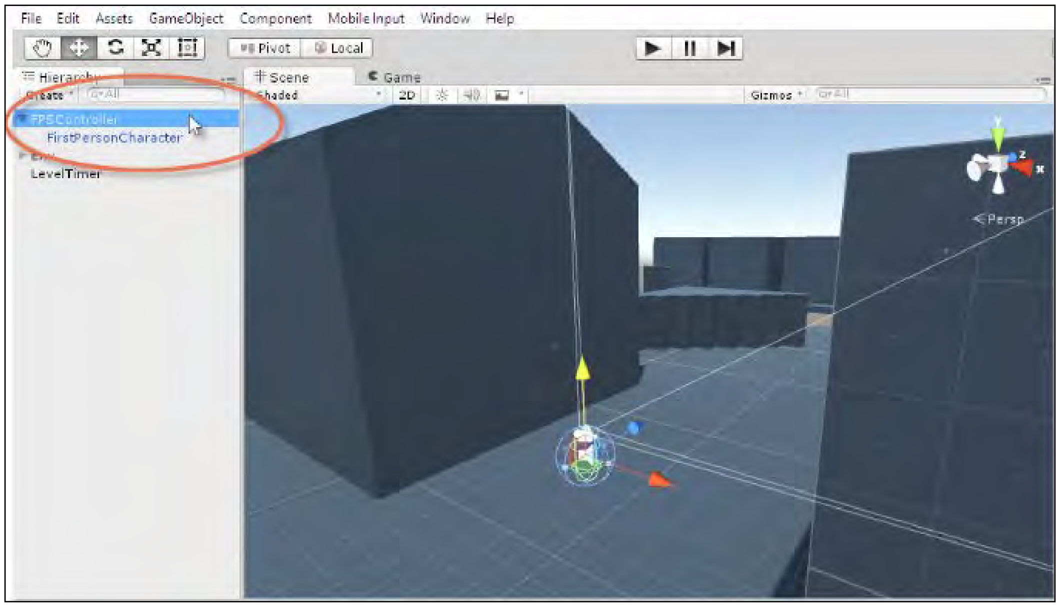
Character Controller
Slope Limit 45
Step Offset 0.3
Skin Width 0.08
Min Move Distance 0
Center X 0 Y 0 Z 0
Radius 0.5
Height 1.8

First Person Controller (Sc
Script FirstPersonCon
Is Walking ☐
Walk Speed 5
Run Speed 10
Runstep Lenghten 0.7
Jump Speed 10
Stick To Ground For 10
Gravity Multiplier 2
Mouse Look
Use Fov Kick ✓
Fov Kick
Use Head Bob ✓
Head Bob
Jump Bob
Step Interval 5
Footstep Sounds
Jump Sound Jump
Land Sound Land

Rigidbody
Mass 1
Drag 0
Angular Drag 0.05

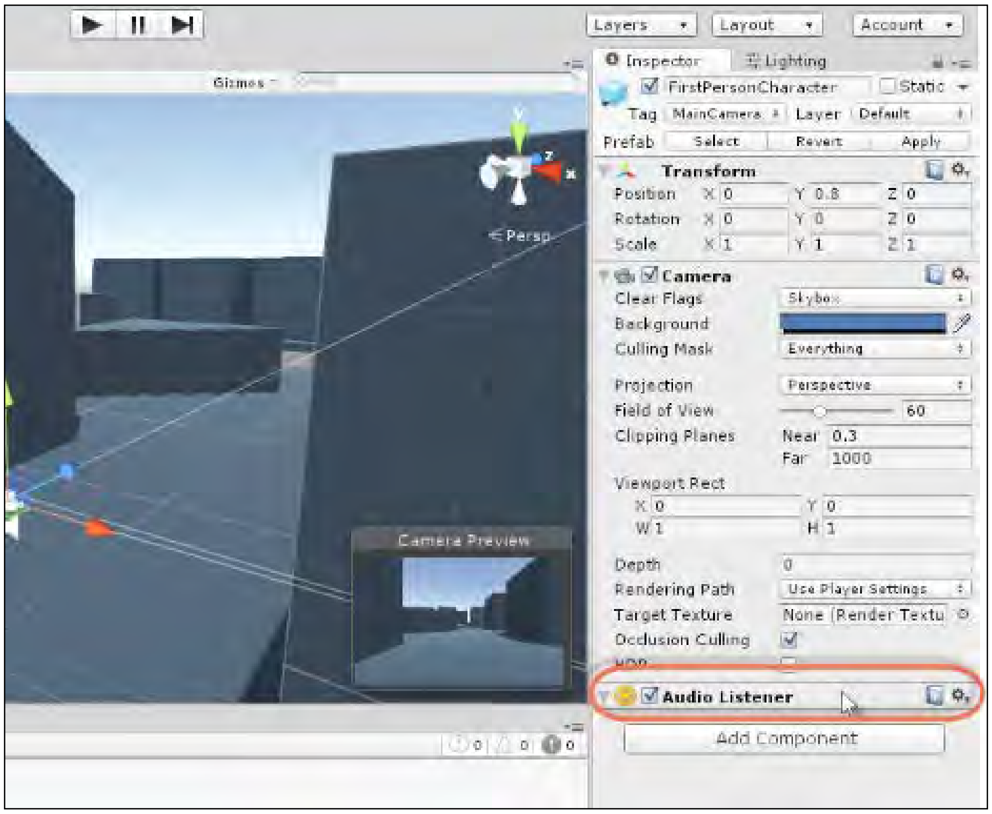
The Console outputs information, warnings and errors...

As mentioned, the Console Window outputs three distinct types of messages: information, warnings and errors. Information messages are typically Unity's way of making best-practice recommendations or suggestions based on how your project is currently working. Warnings are slightly more serious and represent problems either in your code or in your scene, which (if not corrected) could result in unexpected behaviors and sub-optimal performance. And finally, Errors describe areas in your scene or code which require careful and immediate attention. Sometimes errors will prevent your game from working altogether, and sometimes errors happen at run-time and can result in game crashes or freezes. The Console Window therefore is helpful, because it helps us debug and address issues with our games. Figure 1.46 has identified an issue concerning duplicated 'audio listeners'. An Audio Listener is a Component attached to a Camera object. Specifically, each and every Camera, by default, has an Audio Listener component attached. This represents an 'ear point'; that is, the ability to hear sound within the scene, from the position of the camera. Unfortunately, Unity doesn't support multiple active Audio Listeners in the same scene; which means you can only hear audio from one place at any one time. This problem happens because our scene now contains two cameras, one which was added automatically when the scene was created, and the other, which is included in the First Person Controller. To confirm this: select the First Person Controller object in the hierarchy panel, and click the Triangle icon beside its name to reveal more objects underneath, which are part of the First Person Controller. See Figure 1.47.



Finding the Camera on a First Person Controller

Select the *FirstPersonCharacter* object, which is underneath the *FPSController* object (as shown in Figure 1.47). The FirstPersonCharacter object is a **Child** of the FPSController, which is the **Parent**. This is because *FPSController* contains or encloses the *FirstPersonCharacter* object in the hierarchy panel. From the Object Inspector, you can see the object has an Audio Listener component. See Figure 1.48.



FirstPersonCharacter ☑ Static ☐
Tag MainCamera Layer Default
Prefab Select Revert Apply

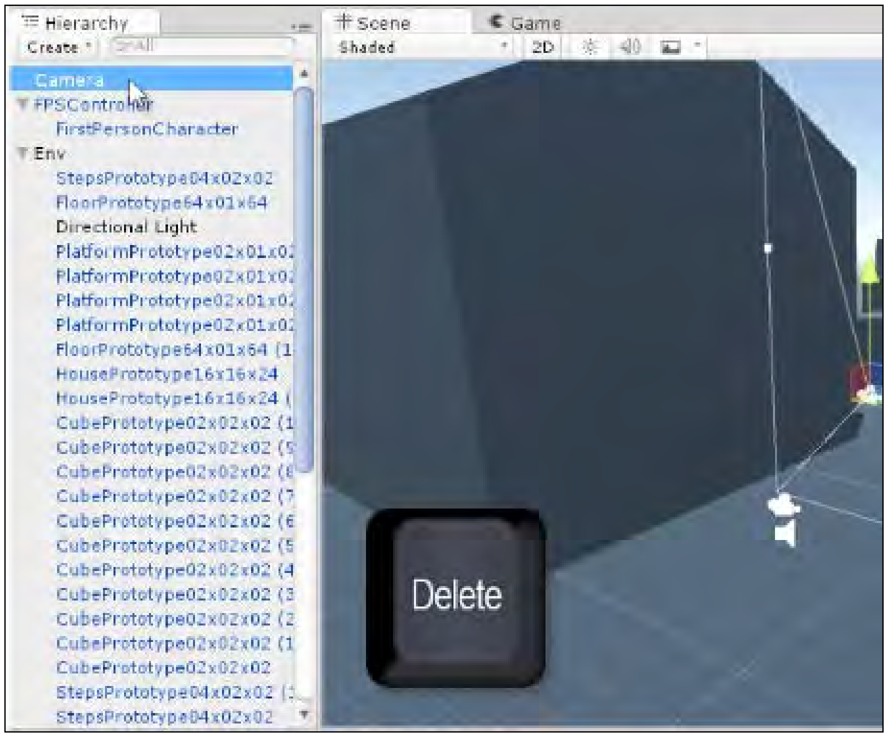
Transform
Position X 0 Y 0.8 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Camera ☑
Clear Flags Skybox
Background
Culling Mask Everything
Projection Perspective
Field of View 60
Clipping Planes Near 0.3 Far 1000
Viewport Rect X 0 Y 0 W 1 H 1
Depth 0
Rendering Path Use Player Settings
Target Texture None (Render Textu
Occlusion Culling ☑
HDR ☐

Audio Listener ☑

The FirstPersonController object contains an AudioListener component

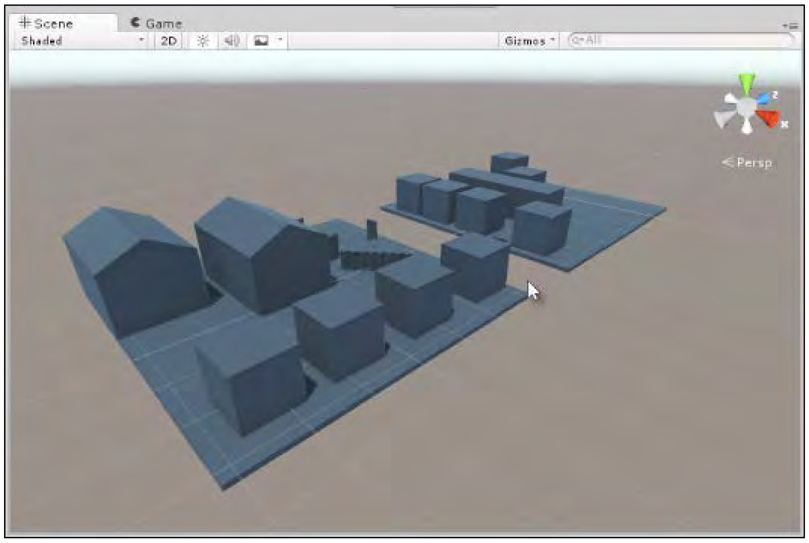
We could remove the AudioListener component from the *FPSController*, but this would prevent the player hearing sound in first person perspective. So instead, we'll delete the *original* camera created by default in the scene. To do this, select the original camera object in the hierarchy and press *Delete* on the keyboard. See Figure 1.49. This removes the AudioListener warning in the Console during gameplay. Now give the game a play test!



Deleting a Camera Object...

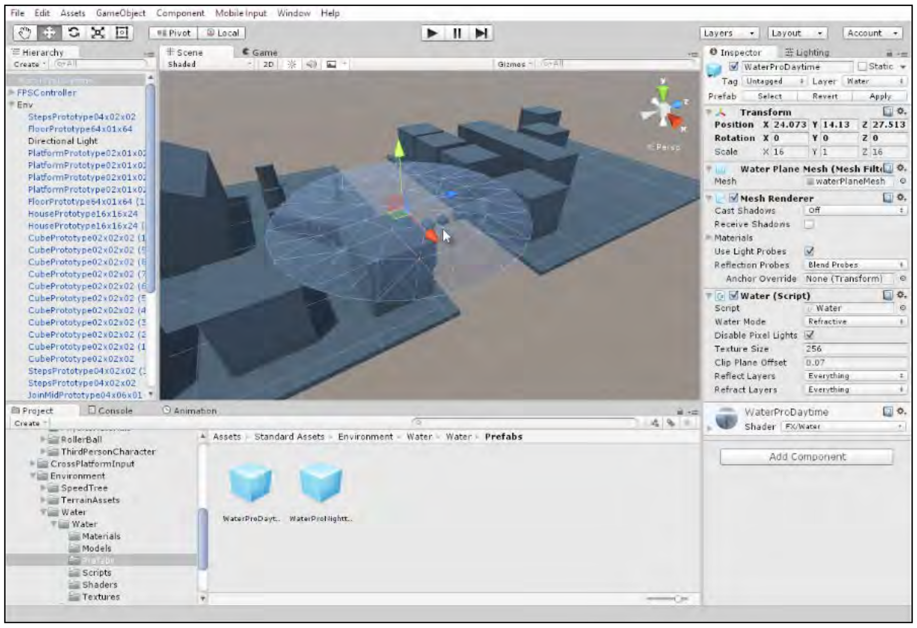
Adding a Water Plane

The Collection Game is making excellent progress. We now have something playable insofar as we can run around and explore the environment in first person mode. But, the environment could benefit from additional polish. Right now, for example, the floor meshes appear suspended in mid-air with nothing beneath them to offer support. See Figure 1.50. Further, it's possible to walk over the edge and fall into an infinite drop. So let's add some water beneath the floors, to complement the scene as a complete environment.



The world floor appears to float and have no support

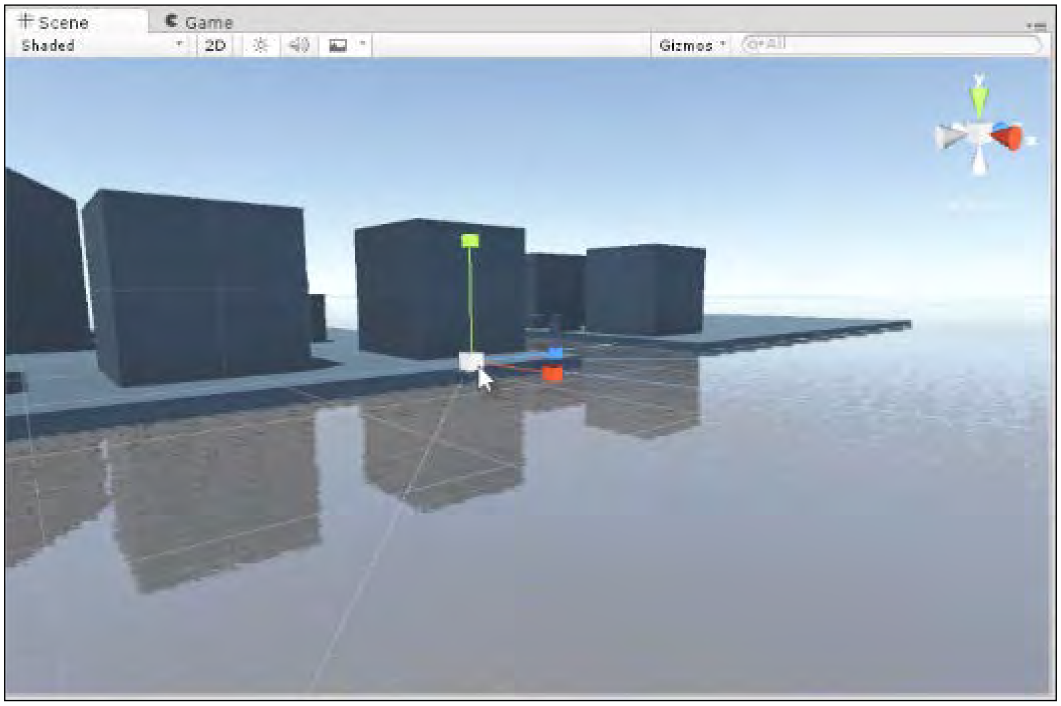
To add water, we can use another ready-made Unity asset, included in the Project Panel. Open the folder *Standard Assets* > *Environment* > *Water* > *Water* > *Prefabs*. Then drag and drop the asset WaterProDaytime from the Project Panel into the scene. See Figure 1.51. This appears as a circular object, which is initially smaller than needed.



WaterProDaytime
Tag Untagged Layer Water
Transform
Position X 24.073 Y 14.13 Z 27.513
Rotation X 0 Y 0 Z 0
Scale X 16 Y 1 Z 16
Water Plane Mesh (Mesh Filter)
Mesh waterPlaneMesh
Mesh Renderer
Cast Shadows Off
Receive Shadows
Use Light Probes ✓
Reflection Probes Blend Probes
Anchor Override None (Transform)
Water (Script)
Script Water
Water Mode Refractive
Disable Pixel Lights ✓
Texture Size 256
Clip Plane Offset 0.07
Reflect Layers Everything
Refract Layers Everything
WaterProDaytime
Shader FX/Water

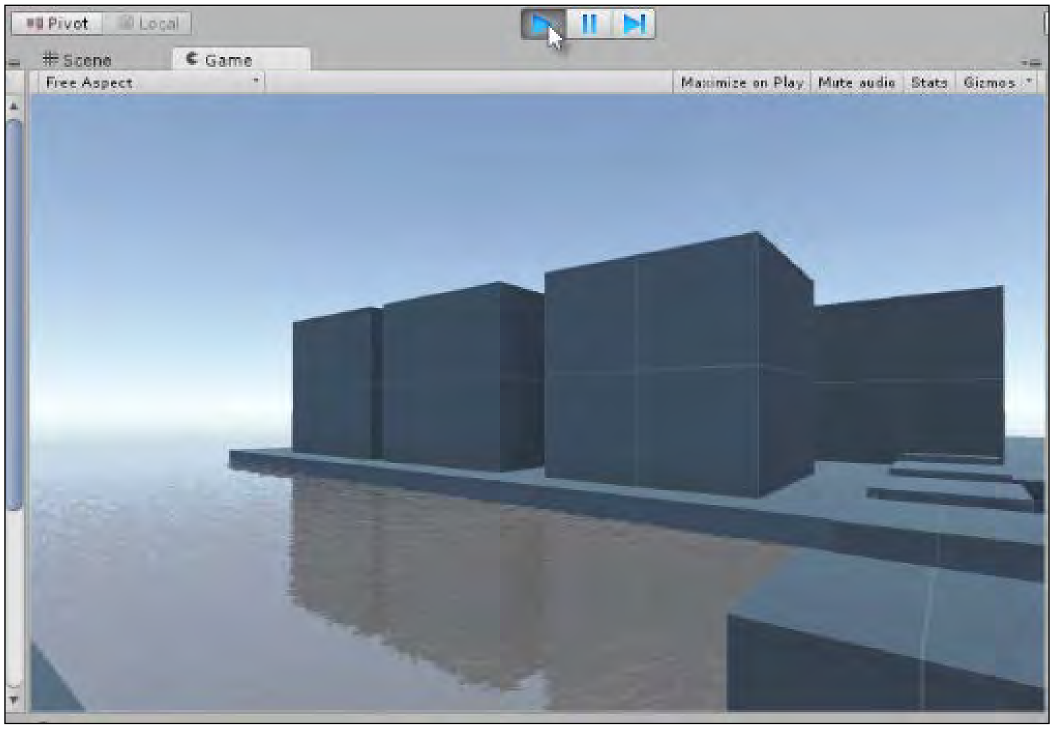
Adding water to the environment...

After adding the Water prefab, position it below the floor level and use the Scale tool (*R*) to increase its planar size (X, Z) to fill the environment outwards into the distant horizon. This creates the feel that the floor meshes are smaller islands within an expansive world of water. See Figure 1.52.



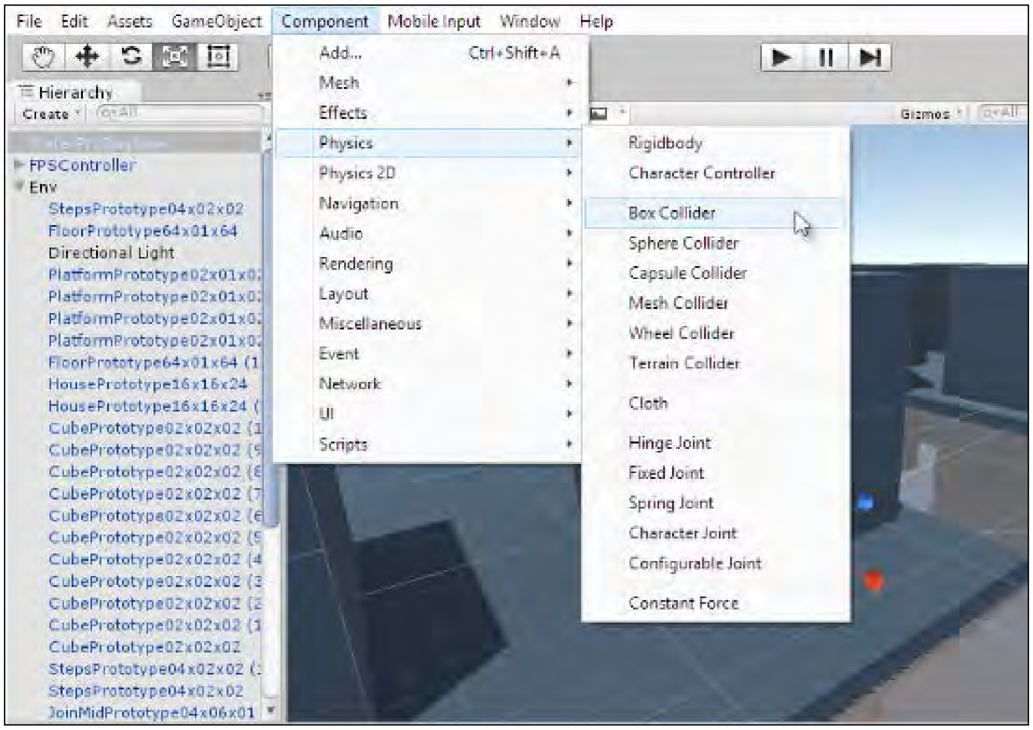
Scaling and sizing water for the environment

Now let's take another test run in the Game tab. Press Play on the tool bar and navigate the character around in first person mode. See Figure 1.53. You should see the water in the level. Of course, you can't walk on the water! Neither can you swim or dive beneath it. If you try walking on it you'll simply fall through it, descending into infinity, as though the water had never been there. Right now, the water is an entirely cosmetic feature, but it makes the scene look much better.



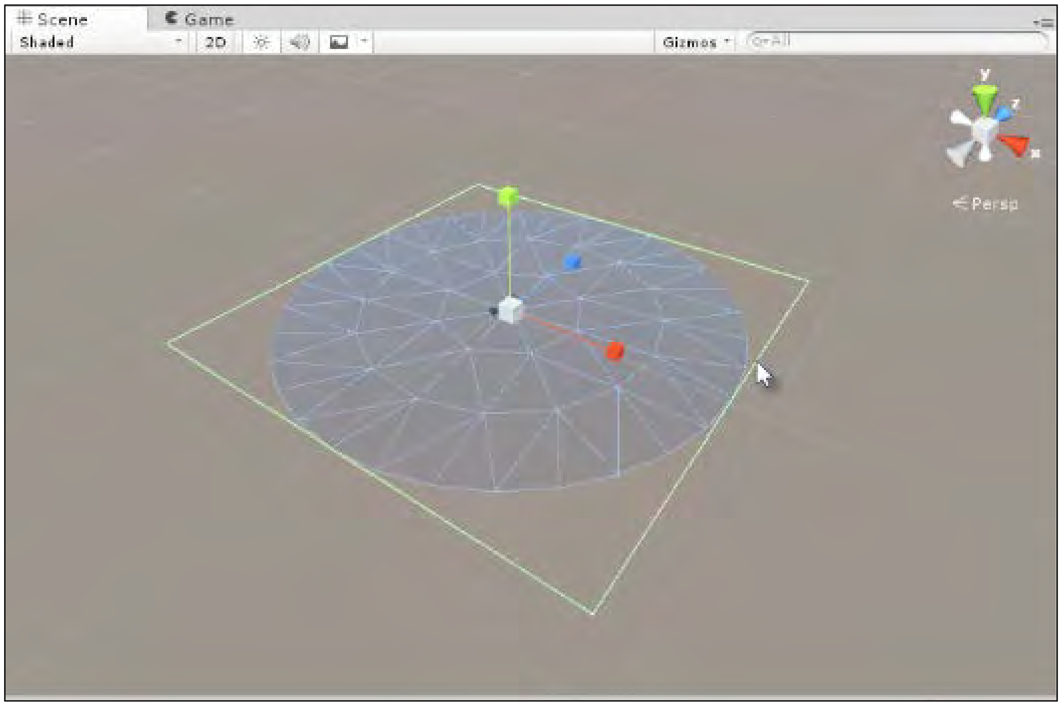
Testing the environment with water in FPS mode

The water is really a substance-less, ethereal object through which the player can pass easily. Unity doesn't recognize it as a solid or even a semi-solid object. As we'll see in more detail later, you can make an object solid very quickly by attaching a *BoxCollider* component to it. Colliders and Physics is covered in more depth from Chapter 3 onwards. For now, however, we can add solidity to the water by first selecting the Water object from the Hierarchy Panel (or in the scene viewport) and then by choosing *Component* > *Physics* > *Box Collider* from the application menu. See Figure 1.54. Attaching a component to the selected object changes the object itself; changes how it behaves. Essentially, components add behavior and functionality to objects; making them behave in different ways. Even so, resist the temptation to simply add lots of components to an object without reason and with the view that it makes them more versatile or powerful. Better is to have as few components on an object as necessary. This strategy of preferring relevant simplicity keeps your workflow neater, simpler and optimized.



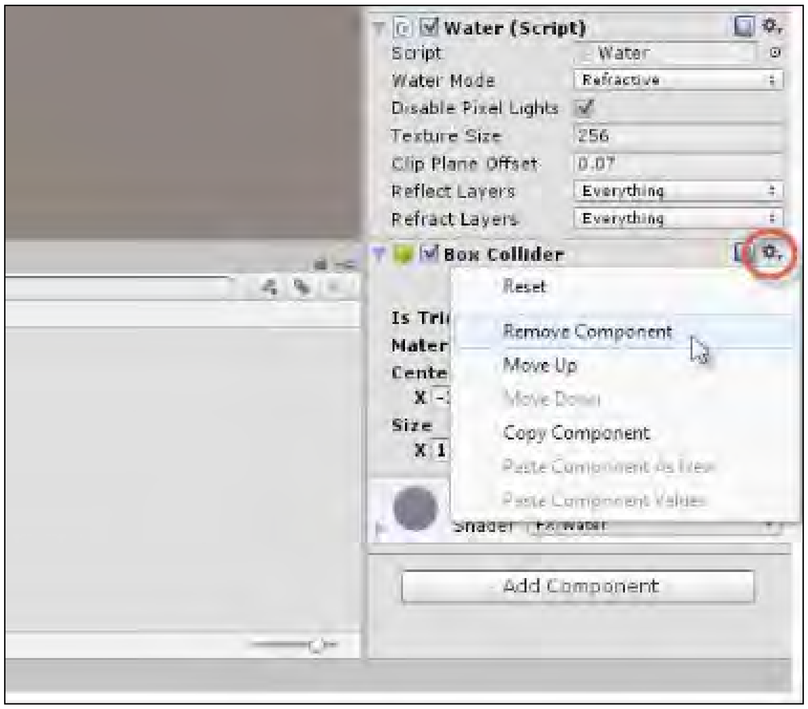
Attaching a Box Collider to Water Object

When a Box Collider is added to the water, a surrounding green cage or mesh appears. This approximates the volume and shape of the water object, and represents its physical volume; namely, the volume of the object that Unity recognizes as solid. See Figure 1.55.



Box Colliders approximate physical volume

If you play the game now your character will walk on water as opposed to falling through. True, the character should properly be able to swim- but walking might be better than falling. To achieve full swimming behavior would require significantly more work and is not covered here. If you want to remove the Box Collider functionality and return the water back to its original, ethereal state, then select the water object, and click the Cog icon on the box collider component, and then choose *Remove Component* from the context menu. See Figure 1.56.

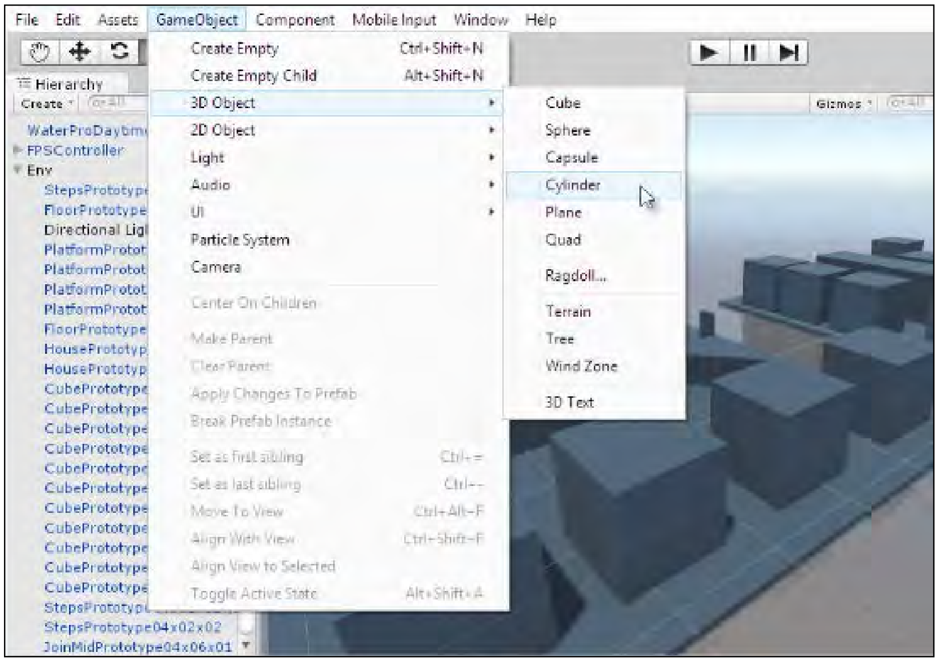


Water (Script)
Script: Water
Water Mode: Refractive
Disable Pixel Lights: ✓
Texture Size: 256
Clip Plane Offset: 0.07
Reflect Layers: Everything
Refract Layers: Everything

Removing a Component

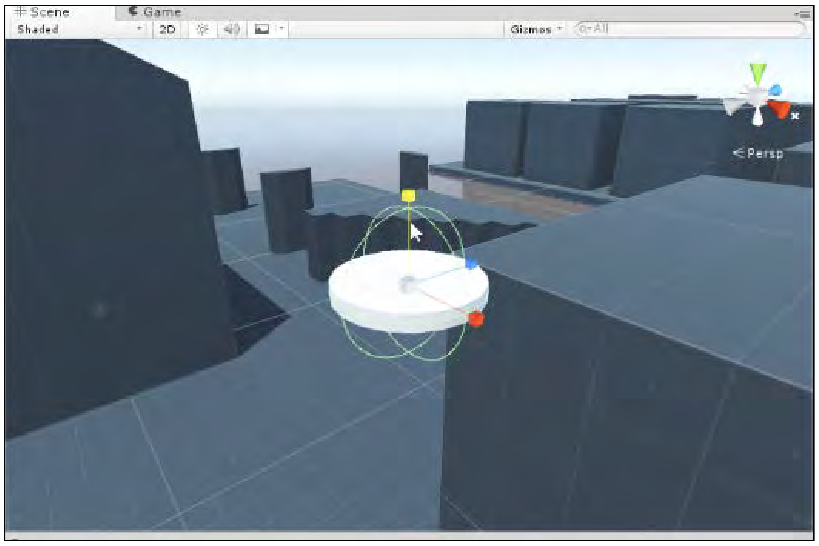
Adding a Coin to Collect

On reaching this far our game has many features, namely a complete environment, a first person controller and water. However, we're supposed to be making a coin collection game, and there aren't yet any coins for the player to collect. Now, to achieve fully collectible coins we'll need to write some C# script, which will happen in the next chapter of this book. But, we can at least get started here at creating the coin object itself. To do that, we'll use a Cylinder primitive that's scaled to form a coin looking shape. To create a cylinder, select *GameObject* > *3D Object* > *Cylinder* from the application menu.



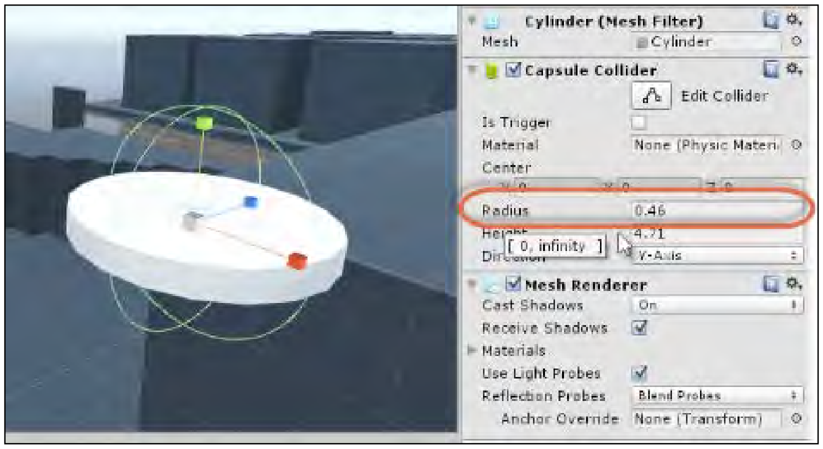
Create a Cylinder

Initially, the Cylinder looks nothing like a coin. But this is easily changed by scaling non-uniformly in the Z axis to make the cylinder thinner. Switch to the Scale tool (R) and then scale the Cylinder inwards. See Figure 1.58.



Scaling the Cylinder to make a Collectible Coin

After re-scaling the coin, its collider no longer represents its volume. It appears much larger than it should do (See Figure 1.58). By default the Cylinder is created with a *CapsuleCollider*, as opposed to a *BoxCollider*. You can change the size of the *CapsuleCollider* component by adjusting the Radius field from the Object Inspector, when the coin is selected. Lower the Radius field to shrink the Collider to a more representative size and volume. See Figure 1.59. Alternatively, you could remove the *CapsuleCollider* altogether and add a *BoxCollider* instead. Either way is fine. The colliders will be used in script in the next chapter, to detect when the player collides with the coin to collect them.



Mesh: Cylinder
Is Trigger: ☐
Material: None (Physic Materi
Center: X 0 Y 0 Z 0
Radius: 0.46
Height: 4.71
Direction: Y-Axis
Cast Shadows: On
Receive Shadows: ✓
Use Light Probes: ✓
Reflection Probes: Blend Probes
Anchor Override: None (Transform)

Adjusting the CapsuleCollider for the coin...

And there we are! We now have the basic shape and structure for a coin. We will, of course improve it carefully and critically in many ways in the next chapter. For example: we'll make it collectible and assign it a material to make it look shiny. But, here, by using only a basic Unity primitive and the Scale tool, we're able to generate a shape that truly resembles a coin.

Summary

Congratulations! On reaching this point you have laid the foundations for a coin collection game that will be completed and functional in the next chapter. Here, we've seen how to create a Unity project from scratch and populate it with assets, like meshes, textures and scenes. In addition, we've seen how to create a scene for our game and use a range of assets to populate it with useful functionality that ships out-of-the-box with the Unity engine, such as Water, First Person Controllers, and Environment Prototyping assets. In the next chapter, we'll resume work from where we ended here by making a coin that is collectible, and by establishing a set of rules and logic for the game, making it possible to win and lose.

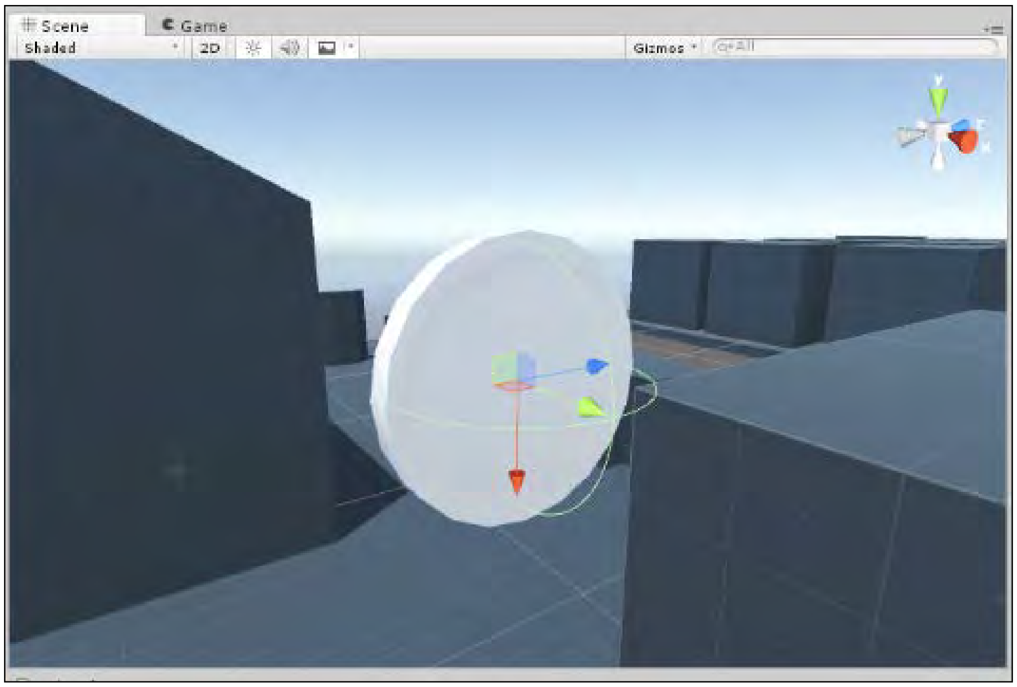
2
Project A: The Collection Game Continued

This chapter continues from the previous by building a collection game with Unity. In this game, the player wanders an environment in first person mode, searching for and collecting all coins in a scene before a global timer expires. If all coins are collected before timer expiry, the game is won. However, if the timer expires before all coins are collected, the game is lost. The project created so far features a complete environment, with a floor, props and water, and it also features a first person controller, along with a basic coin object, which looks correct in shape and form but still cannot be collected. This chapter completes the project by creating a coin object to collect, and adding a timer system to determine whether the total game time has elapsed. In essence, this chapter is about defining a system of logic and rules governing the game. To achieve this, we'll need to code in C#, and so this chapter requires a basic understanding of programming. This book is about Unity and developing games with that engine. The basics of programming as a subject is, however, beyond the scope of this book. So I'll assume you already have a working knowledge of coding generally but have simply not coded in Unity before. Overall this chapter demonstrates:

1. Material Creation
2. Prefabs
3. Coding with C#
4. Writing Script Files
5. Using Particle Systems
6. Building and Compiling Games

Creating a Coin Material

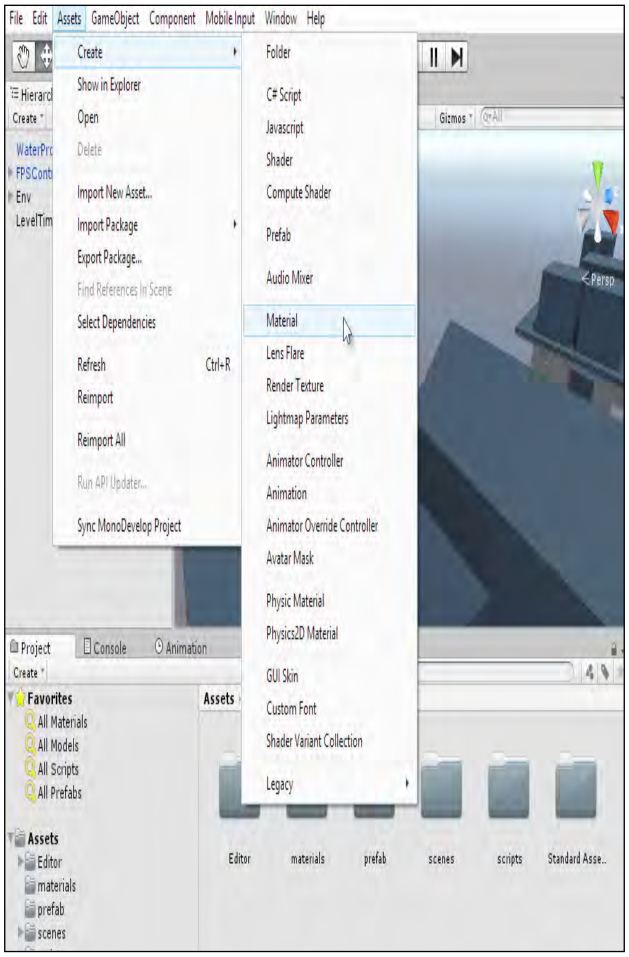
The previous chapter closed by creating a basic coin object from a non-uniformly scaled cylinder primitive. This object was created by selecting *GameObject* > *3D Object* > *Cylinder* from the application menu. See Figure 2.1. The coin object as a concept represents a basic or fundamental unit in our game logic, because the player character should be actively searching the level looking for coins to collect before a timer runs out. This means the coin is more than mere 'appearance'; its purpose in-game is not simply eye-candy, but is *functional*. It makes an immerse difference to the game outcome whether or not the coin is collected by the player. Therefore, the coin object, as it stands, is lacking in two important respects. Firstly, it looks dull and grey- it doesn't really stand out and grab the player's attention. And secondly, the coin cannot actually be collected yet. Certainly, the player can walk into the coin, but nothing appropriate happens in response.



The coin object so far...

The completed 'Collection Game' project, as discussed in this chapter and the next, can be found in the book companion files, inside the *Chapter02/CollectionGame* folder.

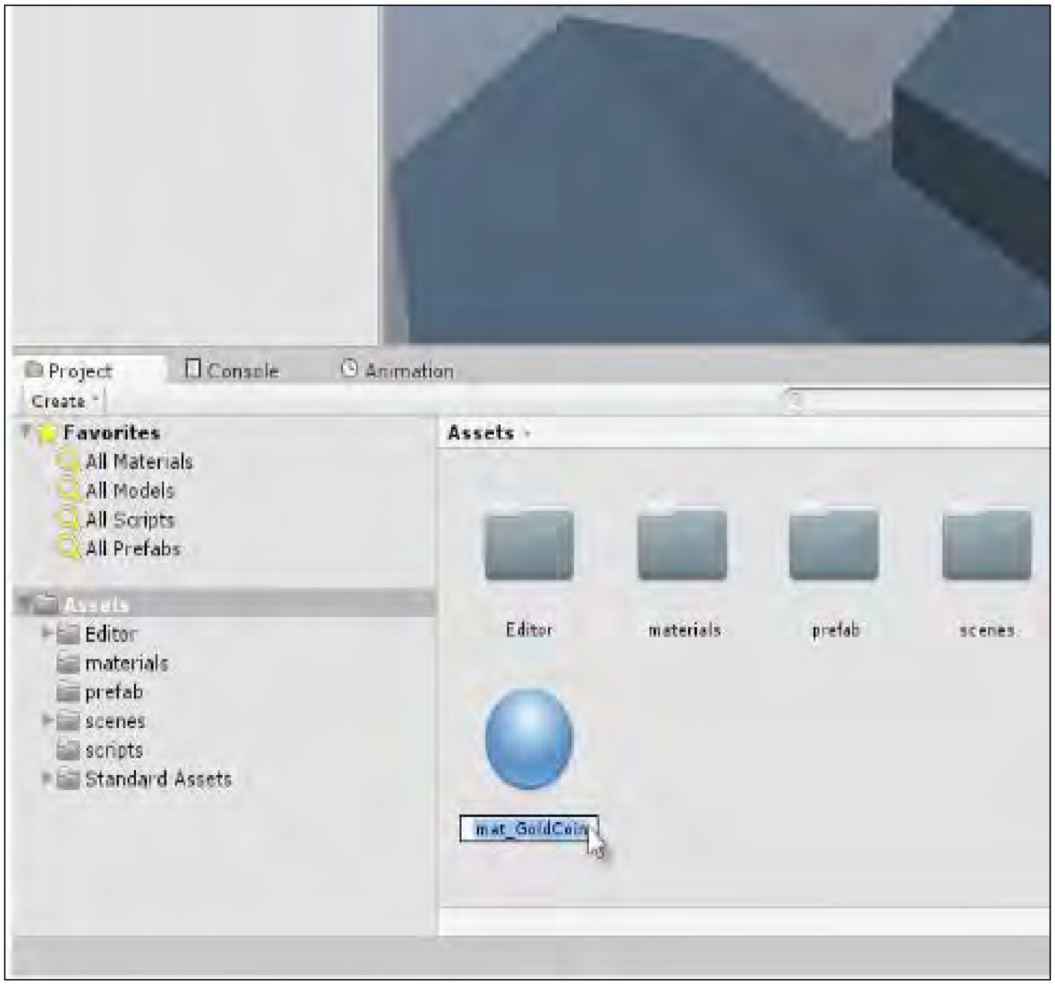
In this section we'll focus on improving the coin appearance using a **Material**. A material defines an algorithm (or instruction set) specifying how the coin should be rendered. A material doesn't just say what the coin should look like in terms of *color*; it defines how *shiny* or smooth a surface is, as opposed to *rough* and diffuse. This is important to recognize, and is why a **Texture** and **Material** refer to different things. A Texture is simply an image file loaded in memory, which can be wrapped around 3D Object via its UV Mapping. In contrast, a Material defines how one or more textures can be combined together and applied to an object to shape its appearance. To create a new Material asset in Unity, right-click on an empty area in the Project Panel, and from the context menu choose *Create* > *Material*. See Figure 2.2. You can also choose *Assets* > *Create* > *Material* from the application menu.



Creating a material

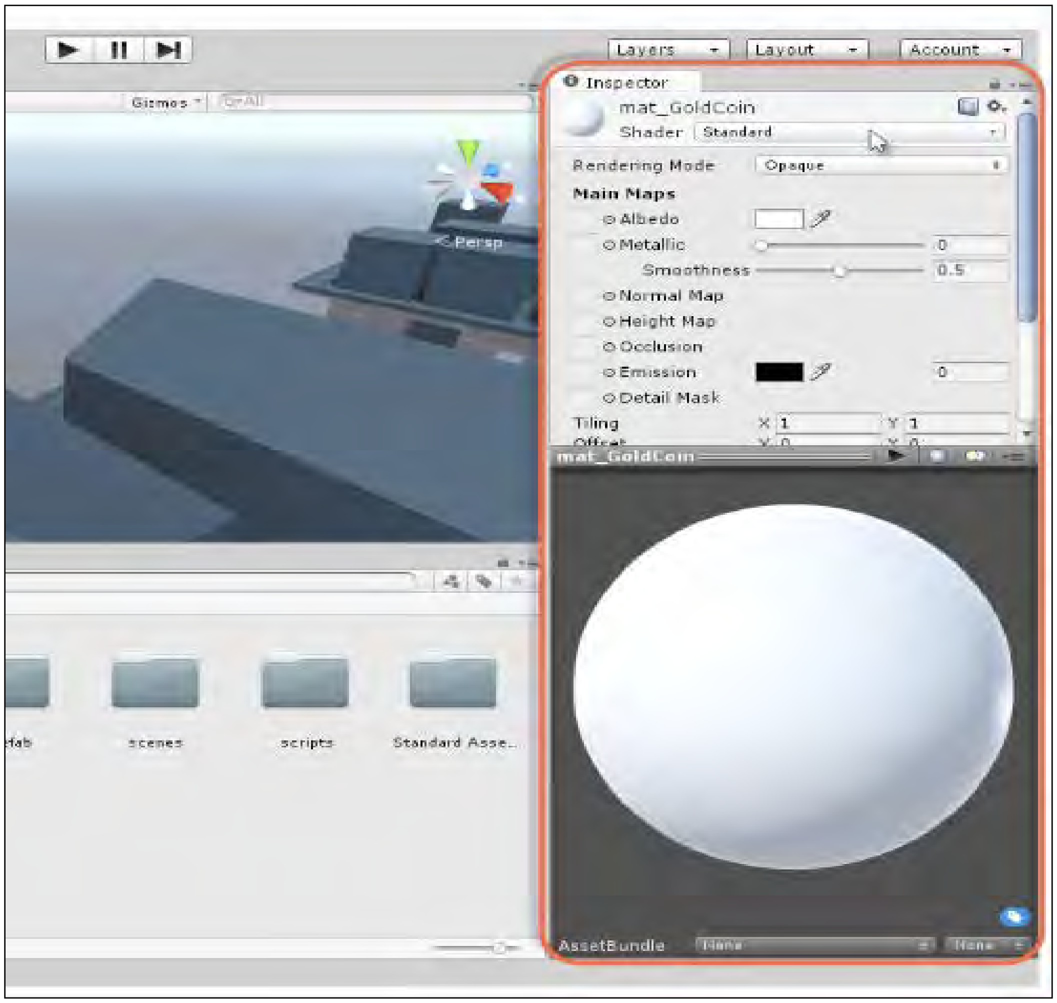
A material is sometimes called a **Shader**. If needed, you can create custom materials using a Shader Language, or you can use a Unity Add-On, such as ShaderForge

After creating a new material, assign it an appropriate name from the Project Panel. Since I'm aiming for a gold look, I'll name the material *mat_GoldCoin*. Prefixing the asset name with *mat* helps me know, just from the asset name, that it's a material asset. Simply type a new name into the text edit field to name the material. You can also click the material name twice to edit the name at any time later. See Figure 2.3.



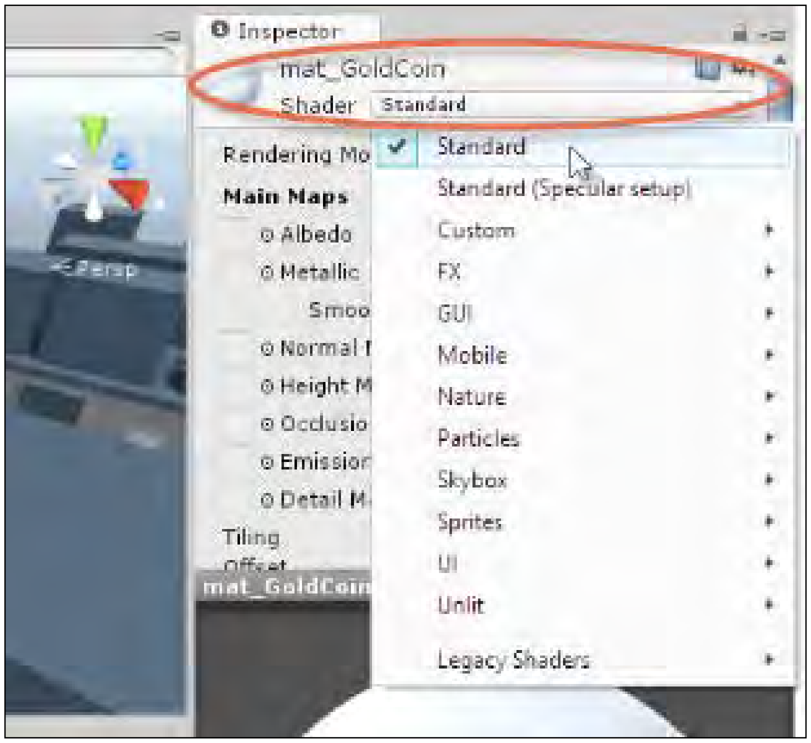
Naming a Material Asset

Next, select the Material Asset in the Project Panel, if it's not already selected, and its properties display immediately in the Object Inspector. There're lots of properties listed! In addition, a material preview displays at the bottom of the Object Inspector, showing you how the material *would* look, based on its current settings, *if* it were applied to a 3D object, like a sphere. As you change material settings from the Inspector, the preview panel updates automatically to reflect your changes, offering instant feedback on how the material would look. See Figure 2.4.



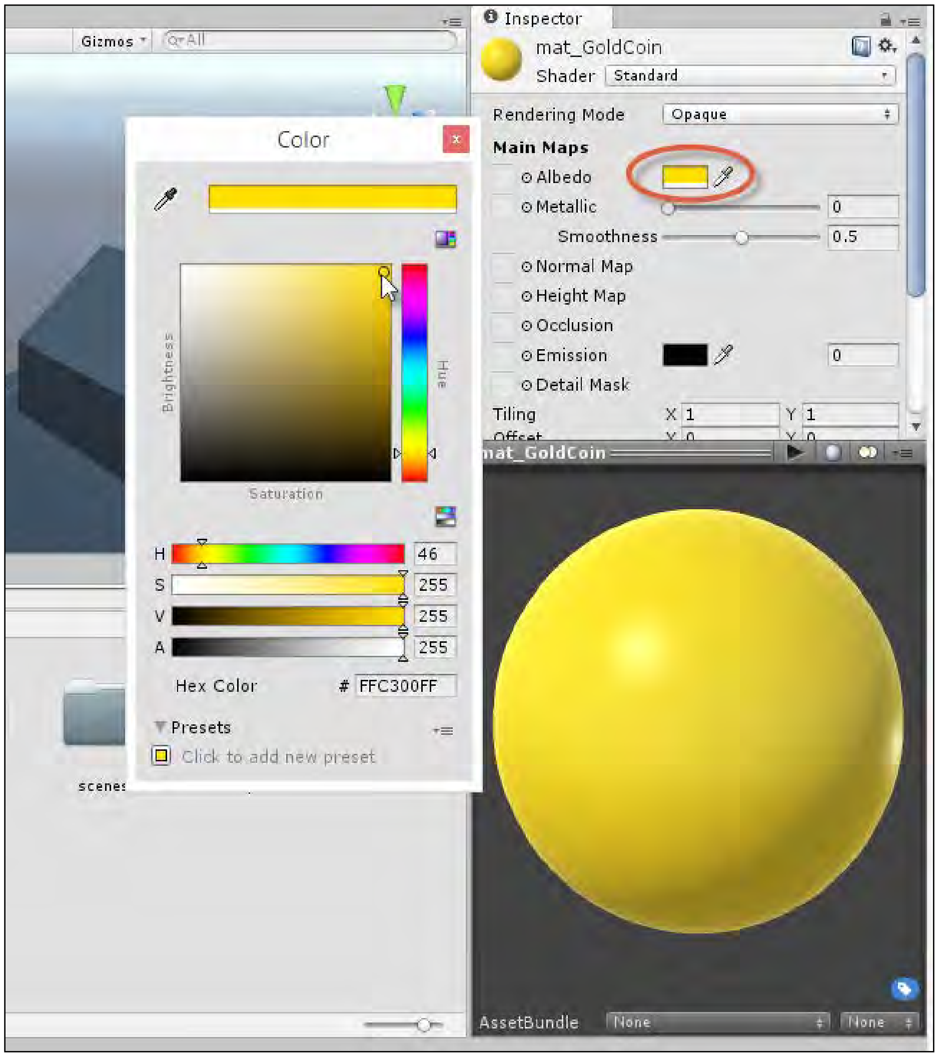
Material properties are changed from the Object Inspector

Let's now create a gold material for the coin. When creating any material, the first setting to choose is the *Shader* type, because this setting effects all other parameters available to you. The Shader type determines which algorithm will be used to shade your object. There are many different choices, but most material types can be approximated using either the *Standard Shader*, or the *Standard (Specular Setup)*. For the gold coin, we can leave the Shader as *Standard*. See Figure 2.5.



Setting the material Shader Type

Right now, the preview panels displays the material as a dull grey, which is far from what we need. To define a gold color, we must specify the *Albedo Channel*. To do this, click the *Albedo* Color slot to display a color picker, and from the Color Picker dialog select a gold color. The material preview updates in response to reflect the changes. See Figure 2.6.

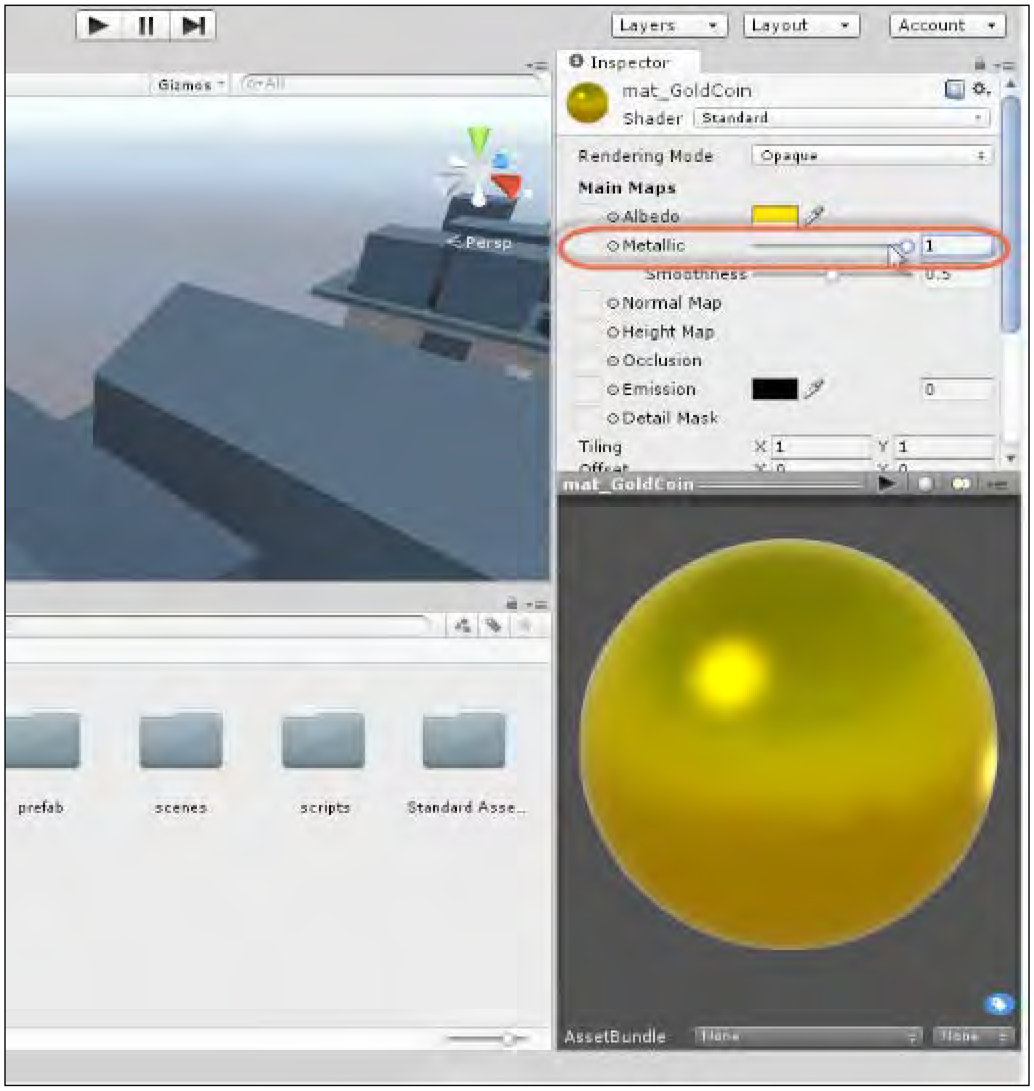


mat_GoldCoin
Shader Standard
Rendering Mode Opaque
Albedo
Metallic 0
Smoothness 0.5
Emission 0
Tiling X 1 Y 1

H 46
S 255
V 255
A 255
Hex Color # FFC300FF

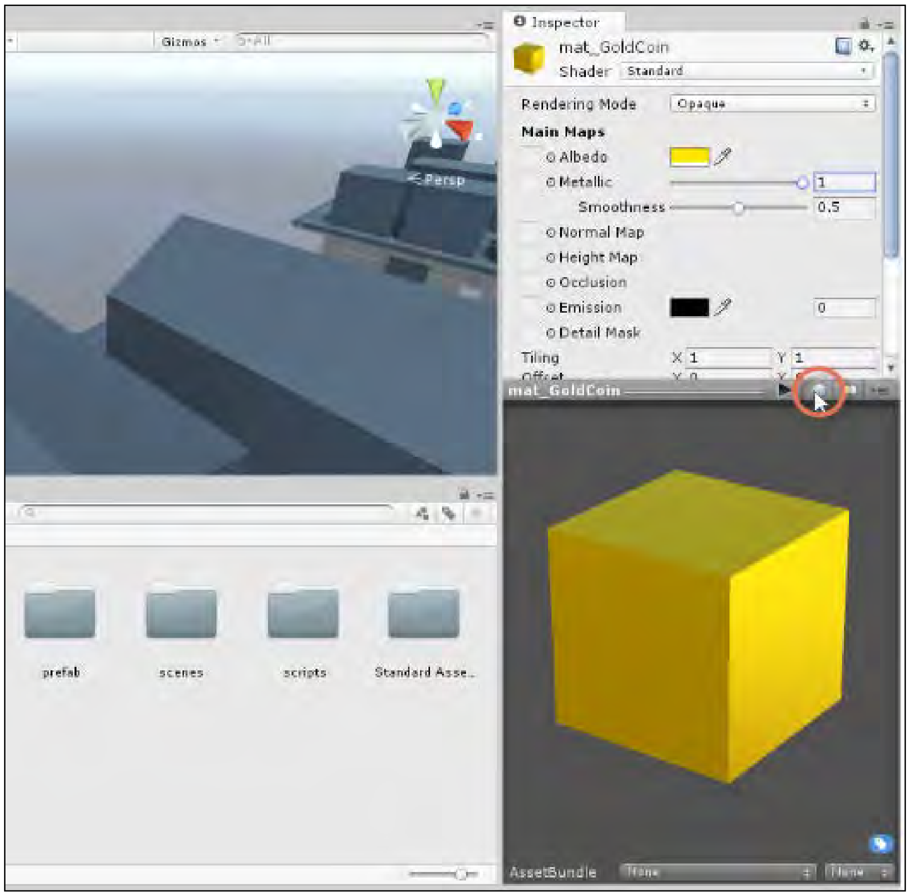
Selecting a Gold Color for the Albedo Channel

The coin material is looking better than it did, but it's still supposed to represent a metallic surface, which tends to be shiny and reflective. To add this quality to our material, click and drag the Metallic Slider in the Object Inspector to the right-hand side, setting its value to 1. This indicates that the material represents a fully metal surface, as opposed to a diffuse surface like cloth or hair. Again, the preview panel will update to reflect the change. See Figure 2.7.



Creating a Metallic Material

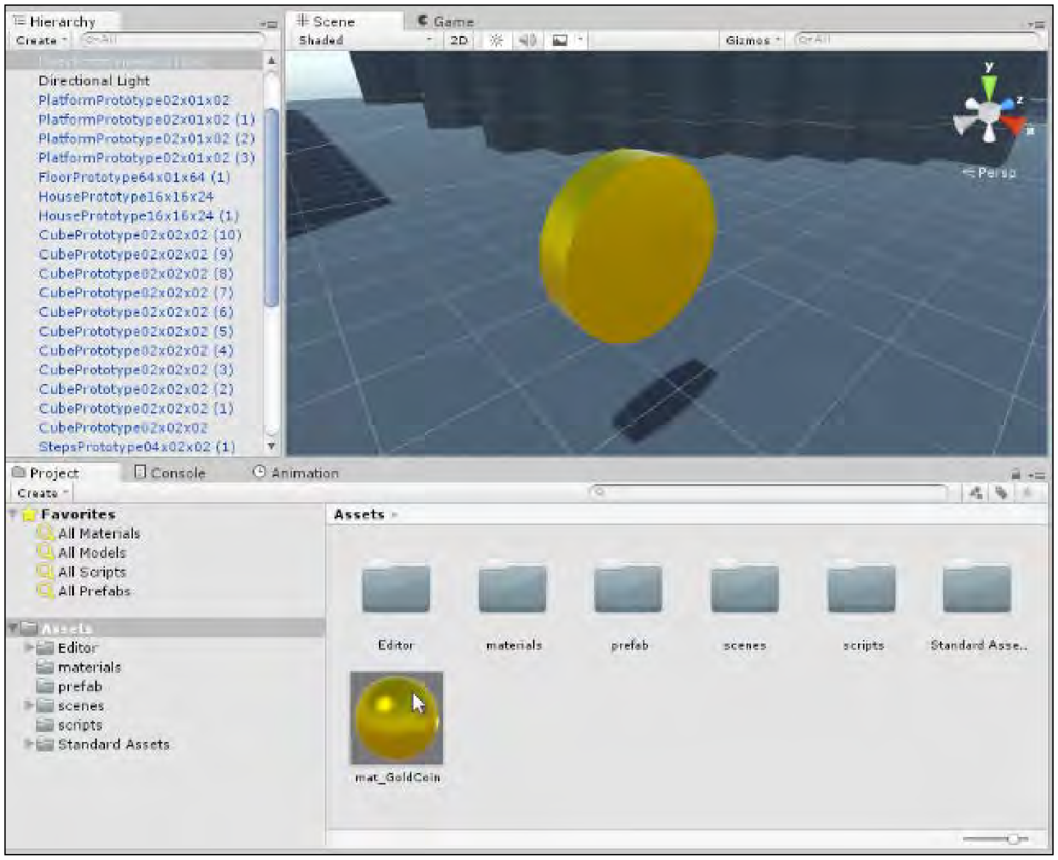
We now have a Gold material created, and it's looking good in the Preview Panel. If needed, you can change the kind of object used for a preview. By default, Unity assigns the created material to a sphere, but other primitive objects are allowed, including Cubes, Cylinders, and a Torus. This helps you preview materials under different conditions. You can change objects by clicking the geometry button directly above the preview panel to cycle through them. See Figure 2.8.



mat_GoldCoin
Shader Standard
Rendering Mode Opaque
Main Maps
Albedo
Metallic 1
Smoothness 0.5
Normal Map
Height Map
Occlusion
Emission 0
Detail Mask
Tiling X 1 Y 1
Offset X 0 Y 0
mat_GoldCoin
AssetBundle None None

Previewing a Material on an Object

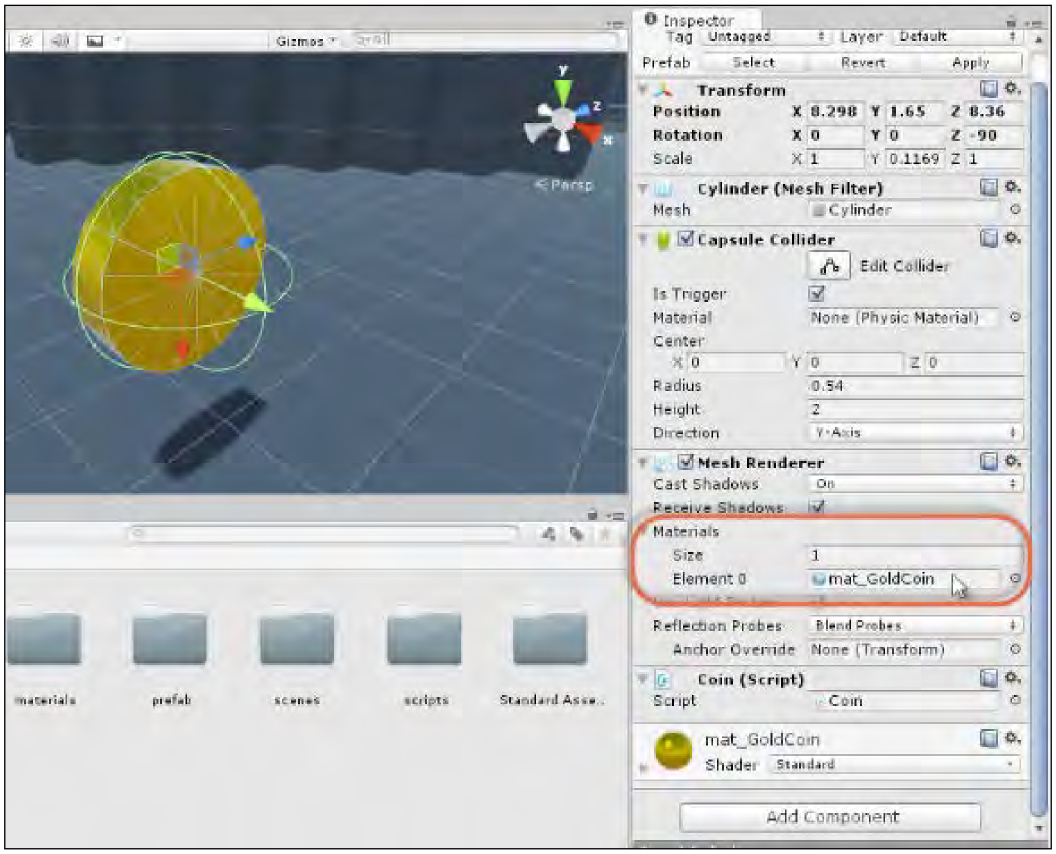
When your material is ready, you can assign it directly to meshes in your scene just by dragging and dropping. Let's assign the coin material to the coin. Click and drag the material from the Project Panel onto the Coin object in the scene. On dropping the material, the coin will change appearance. See Figure 2.9.


Hierarchy
Create
Directional Light
PlatformPrototype02x01x02
PlatformPrototype02x01x02 (1)
PlatformPrototype02x01x02 (2)
PlatformPrototype02x01x02 (3)
FloorPrototype64x01x64 (1)
HousePrototype16x16x24
HousePrototype16x16x24 (1)
CubePrototype02x02x02 (10)
CubePrototype02x02x02 (9)
CubePrototype02x02x02 (8)
CubePrototype02x02x02 (7)
CubePrototype02x02x02 (6)
CubePrototype02x02x02 (5)
CubePrototype02x02x02 (4)
CubePrototype02x02x02 (3)
CubePrototype02x02x02 (2)
CubePrototype02x02x02 (1)
CubePrototype02x02x02
StepsPrototype04x02x02 (1)
Scene
Game
Shaded
2D
Gizmos
Persp
Project
Console
Animation
Favorites
All Materials
All Models
All Scripts
All Prefabs
Assets
Editor
materials
prefab
scenes
scripts
Standard Assets
Standard Asse..
mat_GoldCoin

Assigning the material to the Coin

You can confirm that material assignment occurred successfully, and can even identify which material was assigned, by selecting the coin object in the scene and viewing its **Mesh Renderer** Component from the Object Inspector. The Mesh Renderer component is responsible for making sure a mesh object is actually visible in the scene when the camera is looking. The Mesh Renderer component contains a *Materials* field. This lists all materials currently assigned to the object. By clicking the material name from the *Materials* field, Unity automatically selects the material in the Project Panel, making it quick and simple to locate materials. See Figure 2.10.

Mesh objects may have multiple materials, with different materials assigned to different faces. For best in-game performance, use as few unique materials on an object as necessary. Make the extra effort to share materials across multiple objects, if possible. Doing so can significantly enhance the performance of your game. For more information on optimizing rendering performance, see the online documentation here: `http://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html`



Tag: Untagged Layer: Default

Prefab

Transform

Position	X 8.298	Y 1.65	Z 8.36
Rotation	X 0	Y 0	Z -90
Scale	X 1	Y 0.1169	Z 1

Cylinder (Mesh Filter)

Mesh: Cylinder

✓ Capsule Collider

Is Trigger	✓
Material	None (Physic Material)
Center	X 0 Y 0 Z 0
Radius	0.54
Height	2
Direction	Y-Axis

✓ Mesh Renderer

Cast Shadows	On
Receive Shadows	✓
Materials	
Size	1
Element 0	mat_GoldCoin
Reflection Probes	Blend Probes
Anchor Override	None (Transform)

Coin (Script)

Script: Coin

mat_GoldCoin

Shader: Standard

The MeshRenderer Component lists all materials assigned to an object

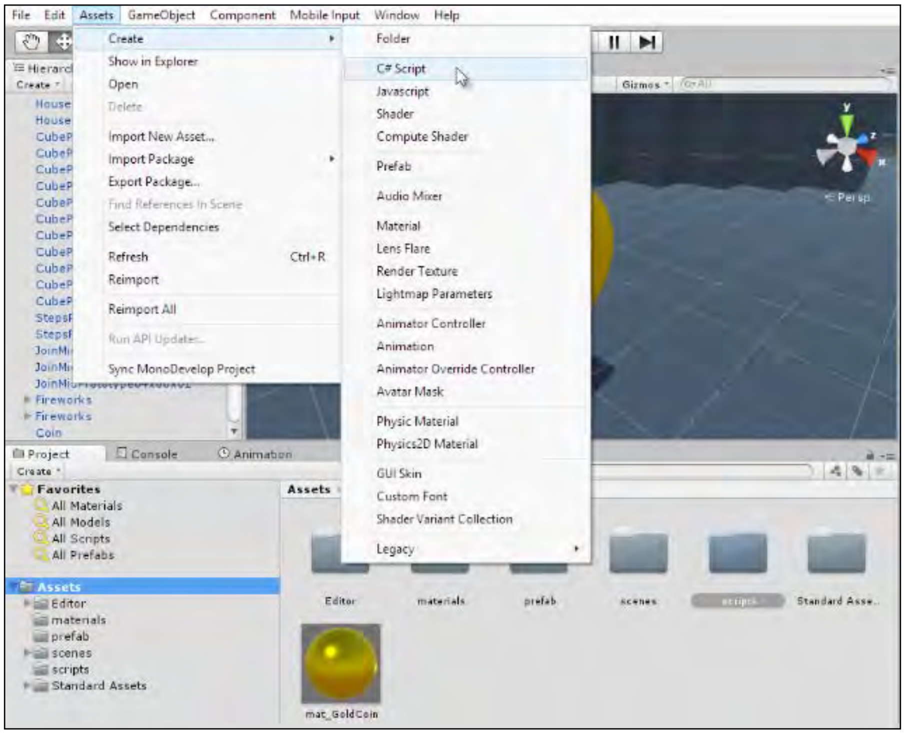
And that's it! You now have a complete and functional gold material for the collectible coin. It's looking good. But, we're still not finished with the coin overall. The coin *looks* right, but it doesn't *behave* right. Specifically, it doesn't disappear when touched, and we don't yet keep track of how many coins the player has collected overall. To address this, then, we'll need to script.

C# Scripting in Unity

Defining game logic, rules and behavior often requires scripting. Specifically, to transform a static and lifeless scene with objects into an environment that 'does something', requires a developer to code behaviors. It requires someone to define how things should act and react under specific conditions. The coin collection game is no exception to this. In particular it requires three main features:

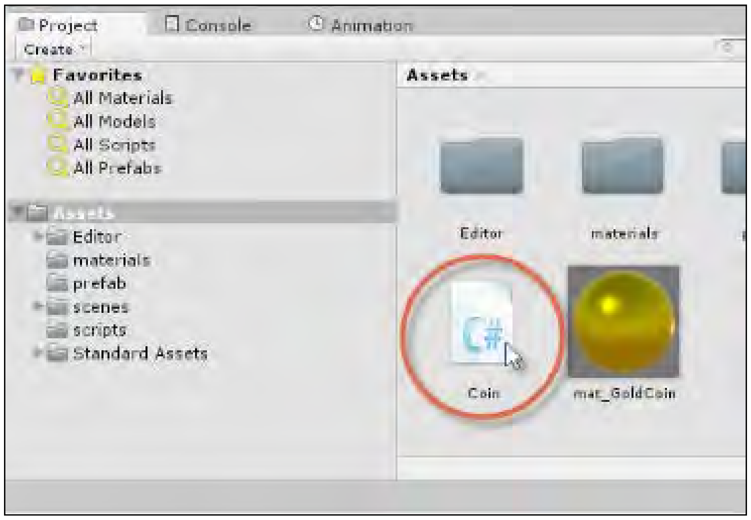
1. To know when the player collects a coin
2. To keep track of how many coins are collected during gameplay
3. To determine whether a timer has expired.

There's no default 'out of the box' functionality included with Unity to handle this scenario. So we must write some code to achieve it. Unity supports two languages, namely *UnityScript* (sometimes called *JavaScript*), and C#. Both are capable and useful languages, but this book uses C#. Let's start coding these three features in sequence. To create a new script file, right-click on an empty area inside the Project Panel, and from the context menu choose *Create* > *C# Script*. Alternatively, you can select *Assets* > *Create* > *C# Script* from the application menu. See Figure 2.11.



Creating a new C# Script

After the file is created, you'll need to assign it a descriptive name. I'll call it *Coin.cs*. In Unity, each script file represents a single, discrete class of matching name. Hence, the file *Coin.cs* encodes the class *Coin*. The coin class will encapsulate the behavior of a coin object and will, eventually, be attached to the coin object in the scene. See Figure 2.12.

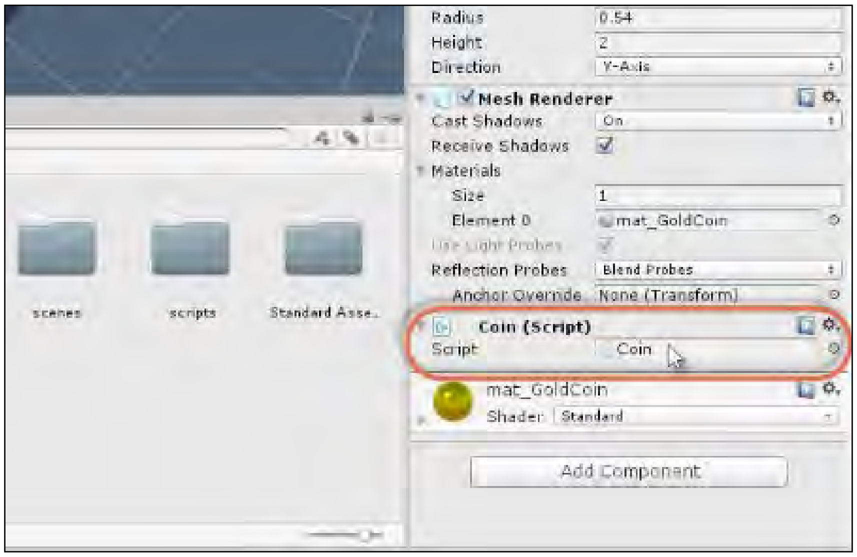


Naming a Script File

Double-click the *Coin.cs* file from the Object Inspector to open it for editing inside MonoDevelop, a third party IDE application that ships with Unity. This program lets you edit and write code for your games. Once opened in MonoDevelop, the source file will appear as shown in Code Sample 2.1.

```
using UnityEngine;
using System.Collections;

public class Coin : MonoBehaviour
{

   // Use this for initialization
   void Start () {}

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

By default, all newly created classes derive from **MonoBehaviour**, which defines a common set of functionality shared by all *Components*. The Coin class features two auto-generated functions, namely *Start* and *Update*. These functions are events invoked automatically by Unity. *Start* is called once as soon as the GameObject (to which the script is attached) is created in the scene. *Update* is called once per frame on the object to which the script is attached. *Start* is useful for initialization code, and *Update* is useful for creating behaviors over time, such as motion and change. Now, before moving any further, let's attach the newly created Script file to the Coin object in the scene. To do that, drag and drop the Coin.cs script file from the Project Panel onto the Coin Object. When you do this, a new Coin component is added to the Object. This means the script is *instantiated* and lives on the object. See Figure 2.13.



Radius 0.54
Height 2
Direction Y-Axis
Cast Shadows On
Receive Shadows ✓
Size 1
Element 0 mat_GoldCoin
Use Light Probes ✓
Reflection Probes Blend Probes
Anchor Override None (Transform)
Script Coin
mat_GoldCoin
Shader Standard

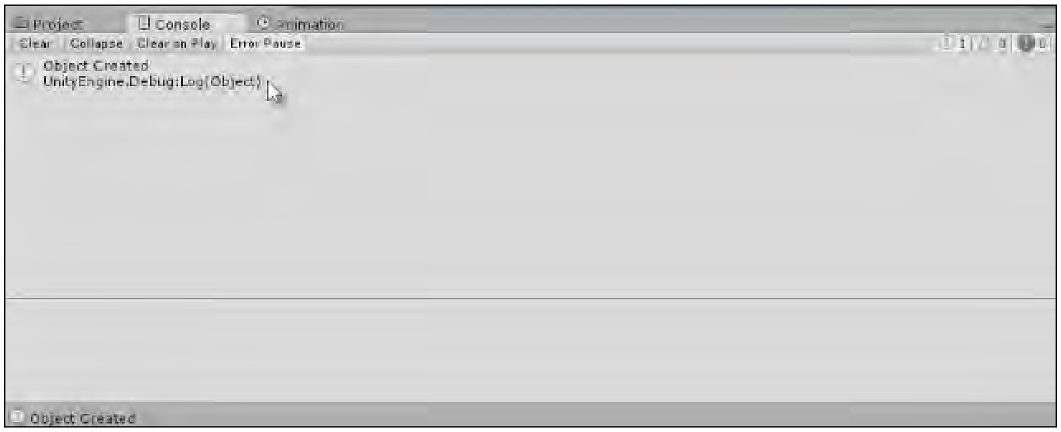
Attaching a Script File to an Object

When a Script is attached to an Object, it exists *on the object* as a Component. A Script file can normally be added to multiple objects, and can even be added to the same object multiple times. Each Component represents a separate and unique instantiation of the class. When a Script is attached in this way, Unity automatically invokes its events, like *Start* and *Update*. You can confirm your script is working as normal by including a **Debug.Log** statement in the *Start* function. This prints a debug message to the Console Window when the GameObject is created in the scene. Consider the following code in Sample 2.2, which achieves this.

```
using UnityEngine;
using System.Collections;
```

```
public class Coin : MonoBehaviour
 {
   // Use this for initialization
   void Start () {
          Debug.Log ("Object Created");
   }

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

   }
}
```

If you press *Play* (*Ctrl+P*) on the toolbar to run your game with the above script attached to an object, you will see the message "Object Created" printed to the Console Window, once for each instantiation of the class. See Figure 2.14.



```
Object Created
UnityEngine.Debug:Log(Object)
```

```
Object Created
```

Printing messages to the Console Window

Good work! We've now created a basic script for the coin class and attached it to the Coin. Next, let's define its functionality to keep track of coins as they are collected.

Counting Coins

The coin collection game wouldn't really be much of a 'game' if there were only one coin. The central idea is that a level should feature many coins, all of which the player should collect before a timer expires. Now, to know whether all coins have been collected, we'll need to know how many coins there are in total in the scene. After all, if we don't know how many coins there are, then we can't know if we've collected them all. So, our first task in scripting is to configure the coin class so we can know, easily, the total number of coins in the scene at any moment. Consider Code Sample 2.3, which adapts the Coin class to achieve this. Comments follow.

```
//------------------------
using UnityEngine;
using System.Collections;
//------------------------
public class Coin : MonoBehaviour
{
   //------------------------
   //Keeps track of total coin count in scene
   public static int CoinCount = 0;
   //------------------------
   // Use this for initialization
   void Start ()
{
         //Object created, increment coin count
         ++Coin.CoinCount;
   }
   //------------------------
   //Called when object is destroyed
   void OnDestroy()
   {
         //Decrement coin count
         --Coin.CoinCount;

         //Check remaining coins
         if(Coin.CoinCount <= 0)
         {
               //We have won
         }
   }
   //------------------------
}
//------------------------
```

Comments on Code Sample 2-3

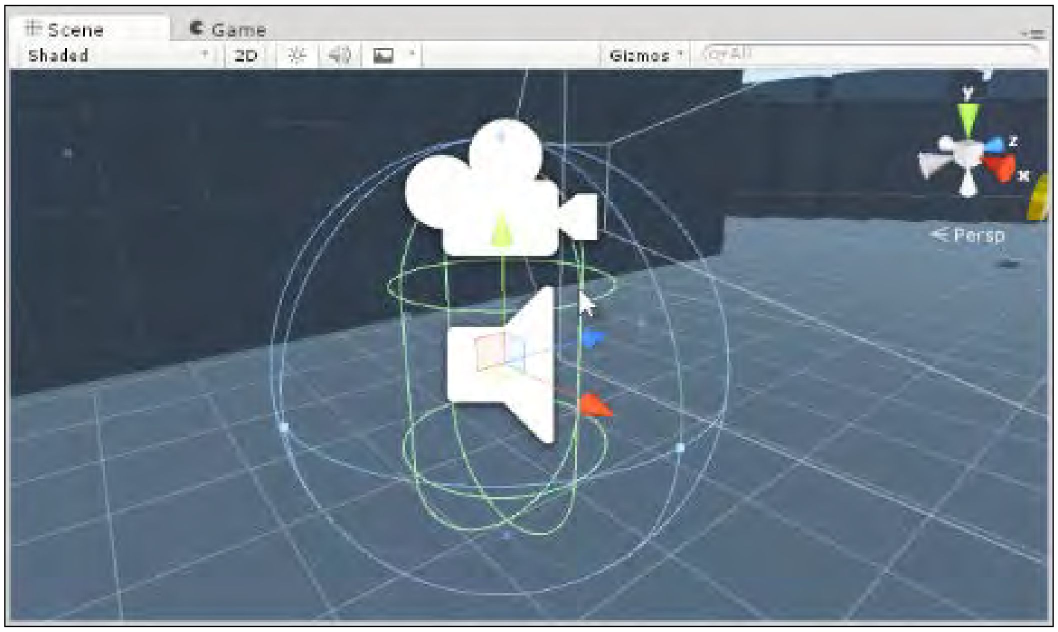
- The Coin class maintains a static member variable *CoinCount*, which, in being static, is shared across all instances of the class. This variable keeps count of the total number of coins in the scene, and each instance has access to it.
- The *Start* function is called once per Coin instance when the object is created in the scene. For Coins that are present when the scene begins, the Start event is called at scene startup. This function increments the *CoinCount* variable by 1 per instance, thus keeping count of all coins.
- The *OnDestroy* function is called once per instance when the object is destroyed. This decrements the *CoinCount* variable, reducing the count for each coin destroyed.

Altogether Code Sample 2-3 maintains a *CoinCount* variable. In short, this variable allows us to always keep track of the total coin count. We can query it easily to determine how many coins remain. This is good, but is only the first step towards completing the coin collection functionality.

Collecting Coins

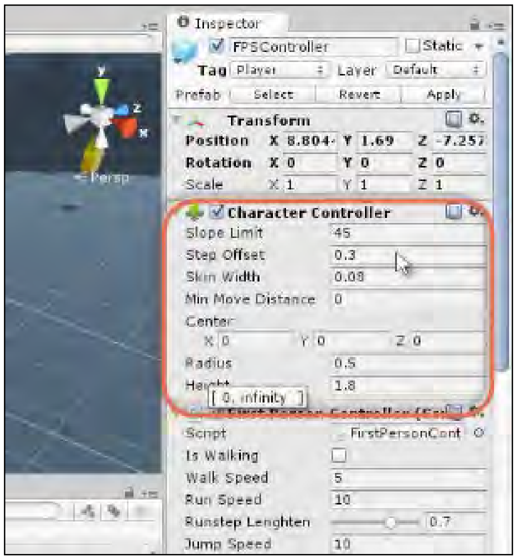
Previously, we developed a coin counting variable, telling us how many coins are in the scene. But regardless of the count, the player still can't collect the coins during gameplay. Let's fix that now. To start, we need to think about **Collisions**. Thinking carefully, we know that a coin is considered *collected* whenever the player *walks into* it. That is, a coin is collected when the player and the coin intersect, or *collide*.

To determine when a Collision happens like that, we must approximate the volume of both the player and a coin, to determine when the two volumes overlap in space. This is achieved in Unity through **Colliders**. Colliders are special physics objects attached to meshes. They tell us when two meshes intersect. The *FPSController* object (First Person Controller) already has a Collider on it, through its **CharacterController** component. This approximates the physical body of a generic person. This can be confirmed by selecting the *FPSController* in the Scene and examining the green wireframe-cage surrounding the main camera. It is capsule-shaped. See Figure 2.15.



The Character Controller features a Collider to approximate the Player Body

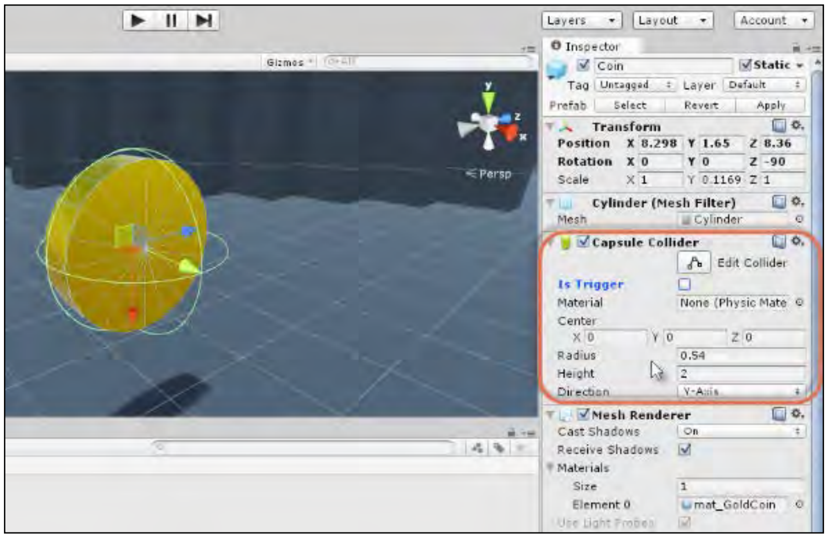
The *FPSController* features a Character Controller component attached, which is configured by default with *Radius*, *Height* and *Center* settings, defining the physical extents of the character in the scene. See Figure 2.16. These settings can be left unchanged for our game.



```
FPSController   Static
Tag Player   Layer Default
Transform
Position  X 8.804  Y 1.69  Z -7.257
Rotation  X 0      Y 0     Z 0
Scale     X 1      Y 1     Z 1
Character Controller
Slope Limit          45
Step Offset          0.3
Skin Width           0.08
Min Move Distance    0
Center
  X 0   Y 0   Z 0
Radius               0.5
Height               1.8
[ 0, infinity ]
Script               FirstPersonCont
Is Walking
Walk Speed           5
Run Speed            10
Runstep Lenghten     0.7
Jump Speed           10
```

The FPSController features a character controller

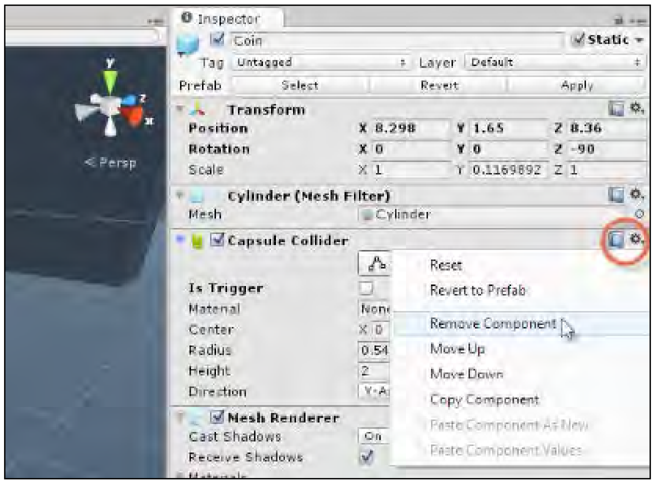
The Coin Object, in contrast, features only a *Capsule Collider* component, which was added automatically when we created the Cylinder primitive earlier to resemble a coin. This approximates the Coin's physical volume in the scene, without adding any additional features specific to characters and motion as found in the *CharacterController* component. This is fine, because the coin is a static object as opposed to a moving and dynamic object, like the FPS Controller. See Figure 2.17.



```
Coin   Static
Tag Untagged   Layer Default
Transform
Position  X 8.298  Y 1.65    Z 8.36
Rotation  X 0      Y 0       Z -90
Scale     X 1      Y 0.1169  Z 1
Cylinder (Mesh Filter)
Mesh              Cylinder
Capsule Collider
Is Trigger
Material          None (Physic Mate
Center
  X 0   Y 0   Z 0
Radius            0.54
Height            2
Direction         Y-Axis
Mesh Renderer
Cast Shadows      On
Receive Shadows   ✓
Materials
  Size            1
  Element 0       mat_GoldCoin
Use Light Probes  ✓
```

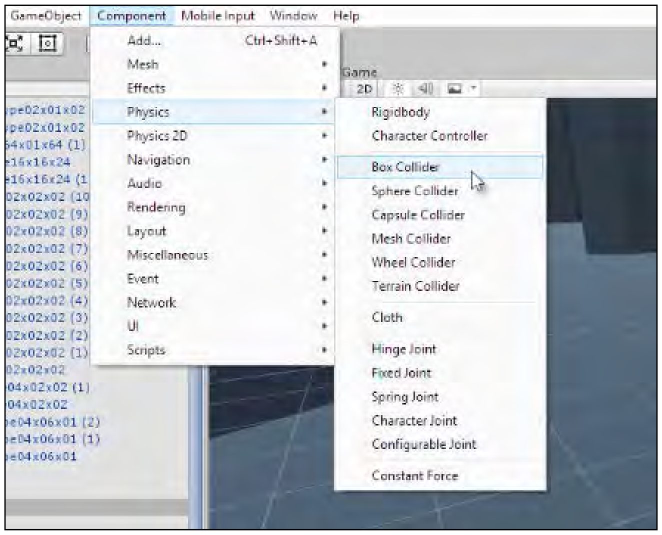
Cylinder Primitives feature a Capsule Collider Component

For this project, I'll stick to using a *CapsuleCollider* component for the coin object. However, if you wanted to change the attached collider to a different shape instead, like a box or a sphere, you can do that by first removing any existing collider components on the Coin: click the *Cog* icon of the component in the Object Inspector, and then select *Remove Component* from the context menu. See Figure 2.18.



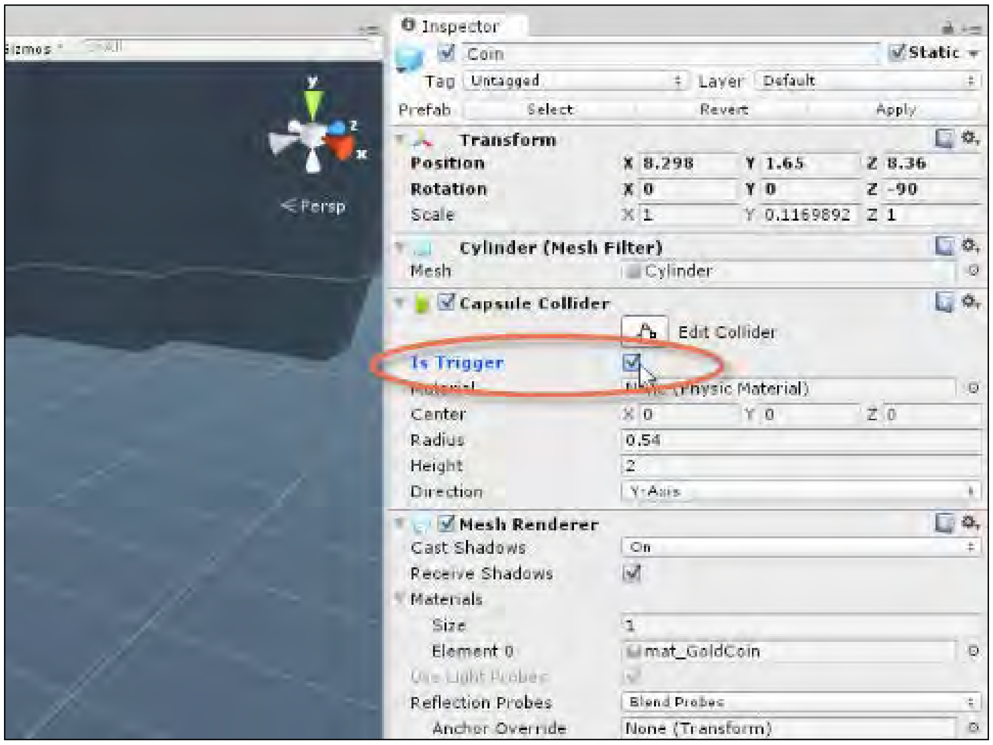
Removing a Component from an Object

You may then add a new Collider Component to the selected object, by choosing *Component* > *Physics* from the application menu, and then choose a suitable shaped collider. See Figure 2.19.



Adding a Component to the selected object

Regardless of the collider type used, there's a minor problem. If you play the game now and try to run through the coin, it'll block your path. The coin acts as a solid, physical object through which the *FPSController* cannot pass. But, for our purposes, this isn't how the coin should behave. It's supposed to be a collectible object. The idea is that: when *walk through it*, the coin is collected and disappears. We can fix this easily, by selecting the Coin object and enabling the *Is Trigger* check box inside the Collider Component, in the Object Inspector. The *Is Trigger* setting appears for almost all collider types. It lets us detect collisions and intersections with other colliders while allowing them to pass through. See Figure 2.20.



```
Coin                                    Static
Tag  Untagged          Layer  Default
Prefab   Select        Revert        Apply
Transform
Position     X 8.298   Y 1.65        Z 8.36
Rotation     X 0       Y 0           Z -90
Scale        X 1       Y 0.1169892   Z 1
Cylinder (Mesh Filter)
Mesh         Cylinder
Capsule Collider
             Edit Collider
Is Trigger   ✓
Material     None (Physic Material)
Center       X 0       Y 0           Z 0
Radius       0.54
Height       2
Direction    Y-Axis
Mesh Renderer
Cast Shadows      On
Receive Shadows   ✓
Materials
  Size            1
  Element 0       mat_GoldCoin
Use Light Probes  ✓
Reflection Probes Blend Probes
  Anchor Override None (Transform)
```

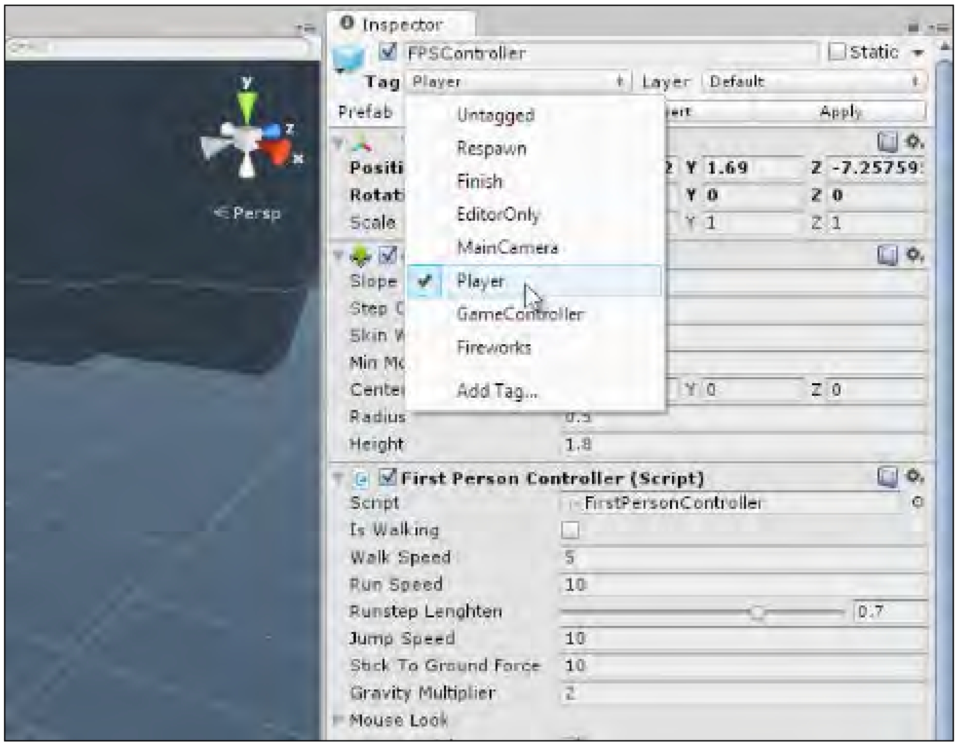
The Is Trigger setting allows objects to pass through colliders

If you play the game now, the *FPSController* will easily walk through all coin objects in the scene. This is a good start. However, the coins don't actually disappear when touched; they still don't get collected. To achieve that, we'll need to add more script to the *Coin.cs* file. Specifically, we'll add an *OnTriggerEnter* function. This function is called automatically when an object, like the player, enters a collider. For now, we'll add a *Debug.Log* statement to print a Debug message when the player enters the collider, just for test purposes. See Code Sample 2.4.

```
//------------------------
using UnityEngine;
using System.Collections;
//------------------------
public class Coin : MonoBehaviour
{
   //------------------------
   public static int CoinCount = 0;
   //------------------------
   // Use this for initialization
   void Start () {
          //Object created, increment coin count
          ++Coin.CoinCount;
   }
   //------------------------
   void OnTriggerEnter(Collider Col)
   {
          Debug.Log ("Entered Collider");
   }
   //------------------------
   //Called when object is destroyed
   void OnDestroy()
   {
          //Decrement coin count
          --Coin.CoinCount;

          //Check remaining coins
          if(Coin.CoinCount <= 0)
          {
                 //We have won
          }
   }
   //------------------------
}
//------------------------
```

More information on the *OnTriggerEnter* function can be found at the online Unity documentation here: `http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerEnter.html`

Test the code in Sample 2.4 by pressing *Play* on the toolbar. When you run into a coin, the *OnTriggerEnter* function will be executed and the message displayed. However, the question remains as to what object initiated this function in the first place. It's true that *something* collided with the coin, but what exactly? Was it the player, an enemy, a falling brick, or something else? To check this, we'll use **Tags**. The Tag feature lets you mark specific objects in the scene with specific tags or labels, allowing those objects to be easily identified in code so we can quickly check that the player, rather than other objects, are colliding with the coins. After all, it should only be the player that can collect coins. So, firstly, we'll tag the player object with a tag called 'Player'. To do this, select the *FPSController* object in the scene, and then click the tag dropdown box in the Object Inspector. From here, select the *Player* tag. This marks the *FPSController* as the *Player* object. See Figure 2.21.



Tagging the FPSController as 'Player'

With the *FPSController* now tagged as *Player*, we can refine the Coin.cs file as shown in Code Sample 2.5. This handles coin collection, making the coin disappear on touch and decreasing the coin count. Comments follow.

```
//------------------------
using UnityEngine;
using System.Collections;
//------------------------
public class Coin : MonoBehaviour
{
   //------------------------
   public static int CoinCount = 0;
   //------------------------
   // Use this for initialization
   void Start () {
         //Object created, increment coin count
         ++Coin.CoinCount;
   }
   //------------------------
   void OnTriggerEnter(Collider Col)
   {
         //If player collected coin, then destroy object
         if(Col.CompareTag("Player"))
               Destroy(gameObject);
   }
   //------------------------
   //Called when object is destroyed
   void OnDestroy()
   {
         //Decrement coin count
         --Coin.CoinCount;

         //Check remaining coins
         if(Coin.CoinCount <= 0)
         {
               //We have won
         }
   }
   //------------------------
}
//------------------------
```

Comments on Code Sample 2-5

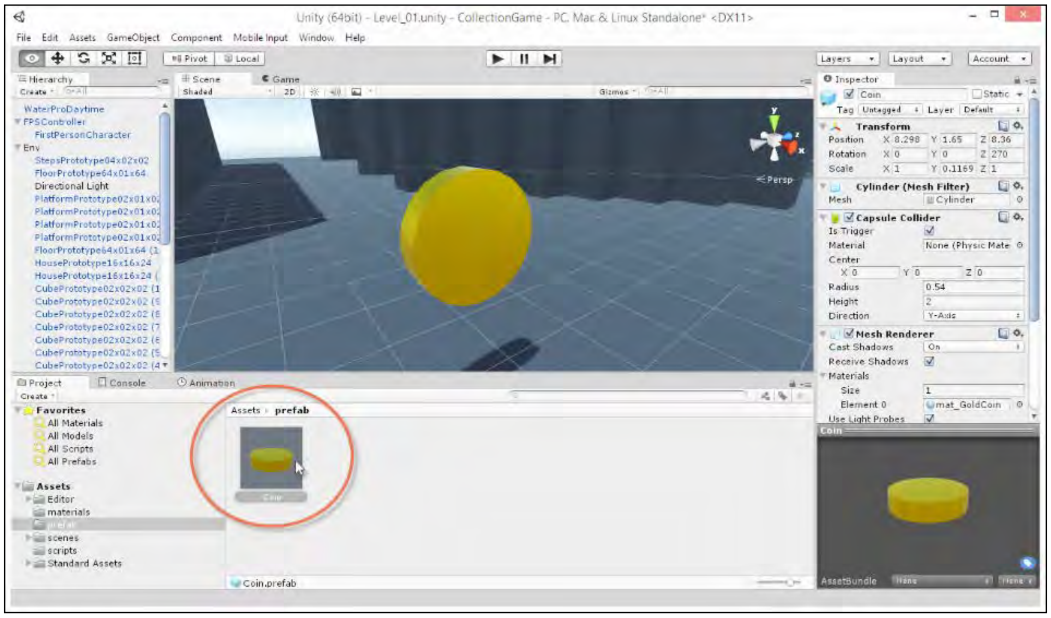
- *OnTriggerEnter* is called once automatically by Unity each time the *FPSController* intersects the Coin collider
- When *OnTriggerEnter* is called, the argument *Col* contains information about the object that entered the collider on this occasion
- The *CompareTag* function is used to determine if the colliding object is the *Player*, as opposed to a different object
- The *Destroy* function is called to destroy the coin object itself, represented internally by the inherited member variable *gameObject*
- When the *Destroy* function is called, the *OnDestroy* event is invoked automatically. This decrements the Coin Count.

Excellent work! You've just created your first working coin. The player can now run into the coin, collecting it and removing it from the scene. This is a great beginning, but the scene should contain more than one coin. We *could* solve this by duplicating the existing coin many times and repositioning each duplicate to a different place. *But* there's a better way, as we'll see next…

Coins and Prefabs

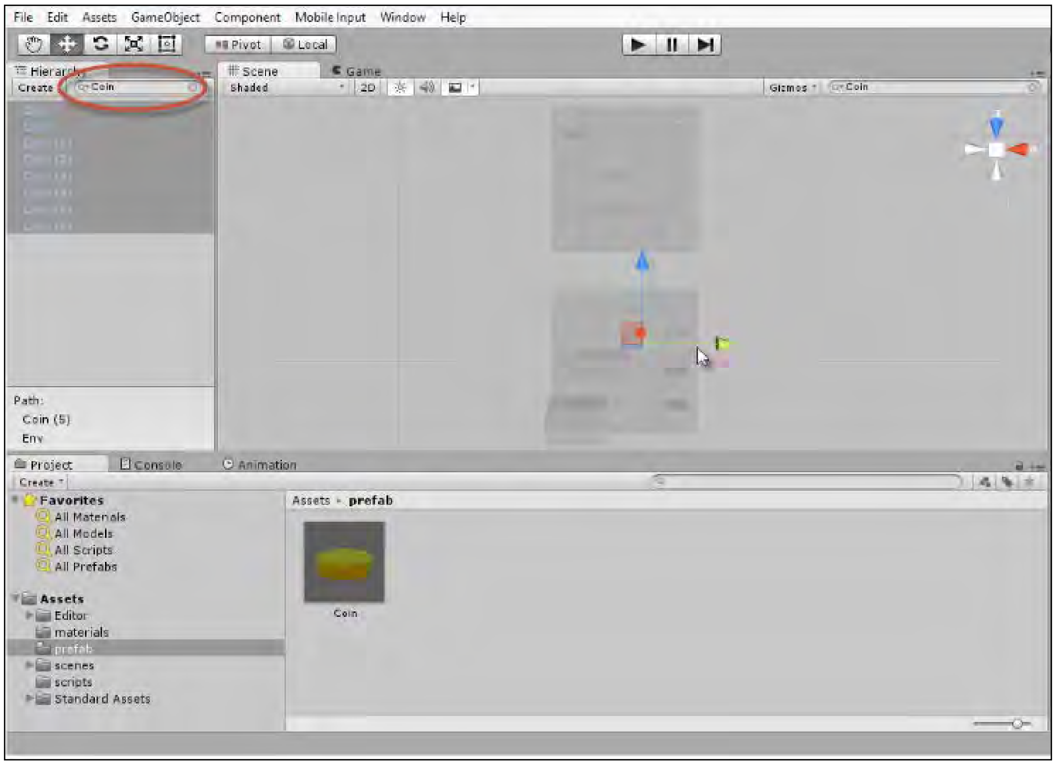
The basic coin functionality is now created. But the scene needs more than one coin. The problem with simply duplicating a coin and scattering the duplicates is that, if we make a change later to one coin and need to propagate that change to all other coins, we'd need to delete the former duplicates and manually replace those with newer and amended duplicates. To avoid this tedious repetition, we can use **Prefabs**.

Prefabs let you convert an object in the scene to an **Asset** in the Project Panel. This can be instantiated in the scene as frequently as needed, as though it were a mesh asset. The advantage is that changes made to the *asset* are automatically applied to all *instances* automatically, even across multiple scenes. This makes it easier to work with custom assets; so let's Prefab the coin right now. To do this, select the coin object in the scene, and then drag and drop it into the Project Panel. When this happens, a new Prefab is created. The object in the scene is automatically updated to be an Instance of the Prefab. This means that, if the Asset is deleted from the Project Panel, the instance will become invalidated. See Figure 2.22.



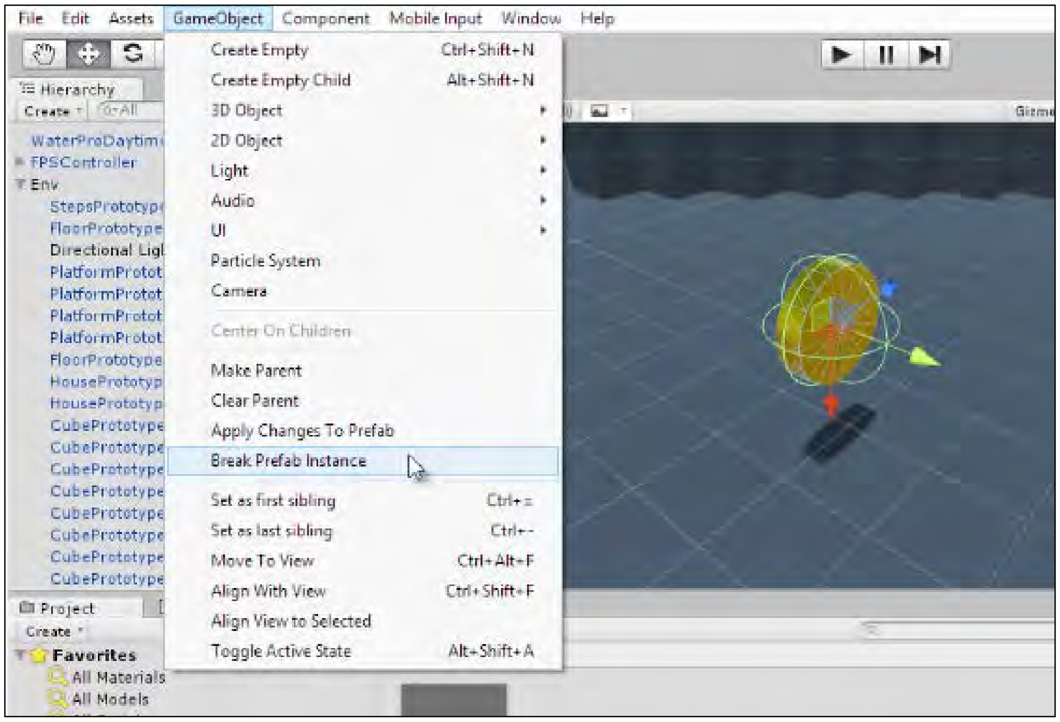
Creating a Coin Prefab

After the prefab is created, you can add more instances of the coin easily to the level, by dragging and dropping the Prefab from the Project Panel into the scene. Each instance is linked to the original prefab asset, which means that all changes made to the *asset* will immediately be made to all *instances*. With this in mind, go ahead now and add as many coin prefabs to the level as suitable for your coin collection game. See figure 2.23 below for my arrangement.



Adding coins prefabs to the level...

One question that naturally arises is how you can transform a prefab back into an independent GameObject that is no longer connected to the Prefab asset. This is useful to do if you want some objects to be *based* on the prefab but to deviate from it slightly. To achieve this, select a Prefab instance *in the scene*, and then choose *GameObject* > *Break Prefab Instance* from the application menu. See Figure 2.24.

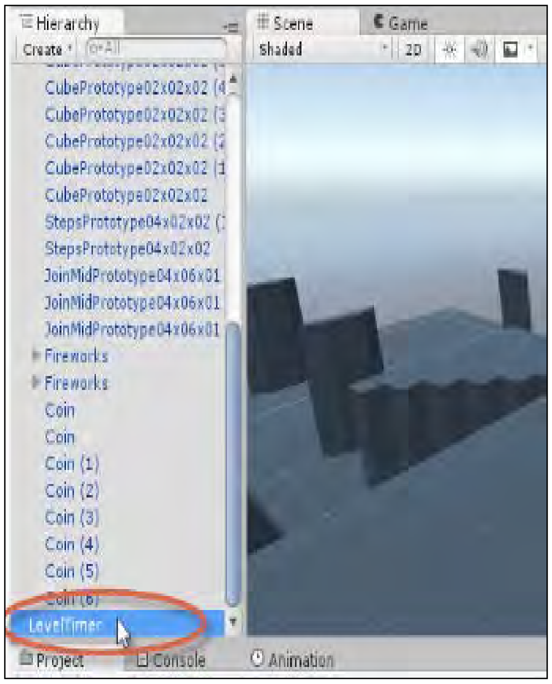


Breaking the prefab instance

TIP. If you add a Prefab instance to the scene and make changes to it that you like and want to distribute upstream back to the Prefab asset, then select the object and choose GameObject > Apply Changes to Prefab.

Timers and Count Downs

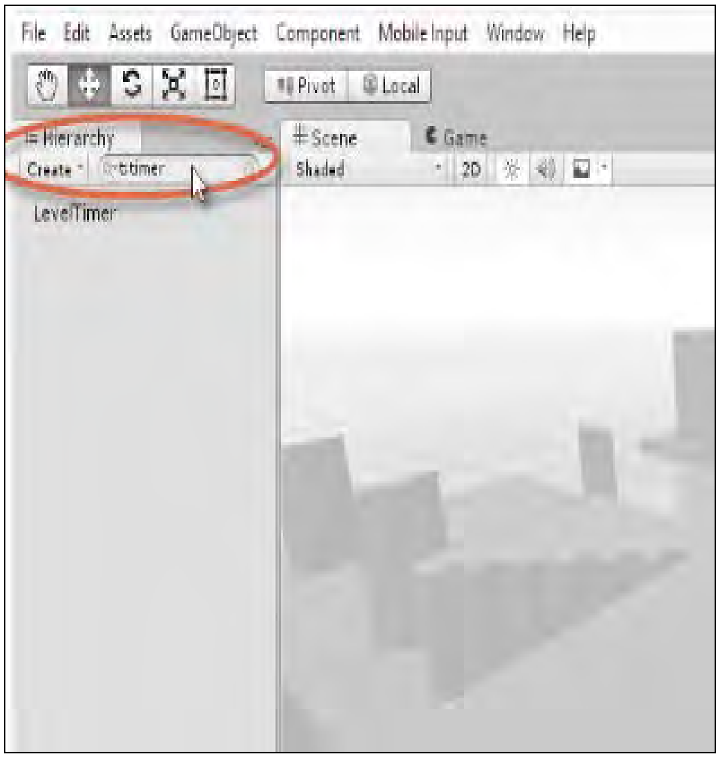
You should now have a level complete with geometry and coin objects. And thanks to our newly added *Coin.cs* script, the coins are both countable and collectible. But even so, the level still poses little or no challenge to the player, because there's no way the level can be won or lost. Specifically, there's nothing for the player to *achieve*. This is why a time-limit is important for the game: it defines a win and loss condition. Namely: collecting all coins before the timer expires results in a win condition, and failing to achieve this results in a loss condition. Let's get started at creating a timer countdown for the level. To do this, create a new and empty game object by selecting *GameObject* > *Create Empty*, and rename this to *LevelTimer*. See Figure 2.25.



Renaming the Timer Object

REMEMBER. Empty game objects cannot be seen by the player because they have no mesh renderer component. They especially useful for creating functionality and behaviors that don't directly correspond to physical and visible entities, such as timers, managers and game logic controllers.

Next, create a new Script file named *Timer.cs* and add it to the *LevelTimer* Object in the scene. By doing this the Timer functionality will exist in the scene. Make sure, however, that the Timer script is added to one object, and no more than one. Otherwise, there will effectively be multiple, competing timers in the same scene. You can always search a scene to find all components of a specified type by using the Hierarchy Panel. To do this, click inside the Hierarchy Search box and type: *t:Timer*. Then press Enter on the keyboard to confirm the search. This searches the scene for all objects with a component attached of type Timer, and the results are displayed in the Hierarchy Panel. Specifically, the hierarchy panel is filtered to show only the matching objects. The prefix *t* in the search string indicates a search by type operation. See Figure 2.26.

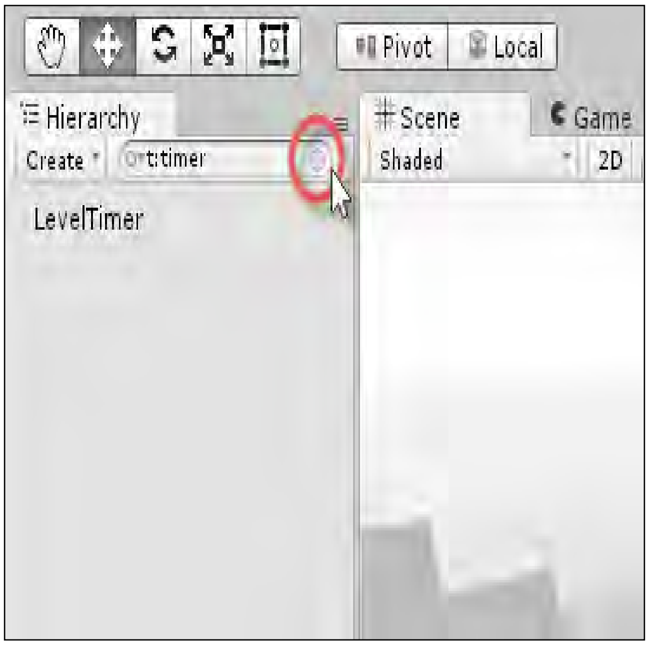


t:timer

LevelTimer

Searching for Objects with a component of matching type...

You can easily cancel a search and return the hierarchy panel back to its original state by clicking the small cross icon, aligned to the right-hand side of the search field. This button can be tricky to spot. See Figure 2.27.



Cancelling a type search

The timer script itself must also be coded if it's to be useful. The full source code for the Timer.cs file is given in Code Sample 2.6 below, and then comments follow. This source code is highly important if you've never scripted in Unity before. It demonstrates so many critical features. See the comments for a fuller explanation.

•

```
//------------------------
using UnityEngine;
using System.Collections;
//------------------------
public class Timer : MonoBehaviour
{
   //------------------------
   //Maximum time to complete level (in seconds)
   public float MaxTime = 60f;
   //------------------------
   //Countdown
```

```
      [SerializeField]
      private float CountDown = 0;
      //------------------------
      // Use this for initialization
      void Start ()
      {
            CountDown = MaxTime;
      }
      //------------------------
      // Update is called once per frame
      void Update ()
      {
            //Reduce time
            CountDown -= Time.deltaTime;

            //Restart level if time runs out
            if(CountDown <= 0)
            {
                  //Reset coin count
                  Coin.CoinCount=0;
                  Application.LoadLevel(Application.loadedLevel);
            }
      }
      //------------------------
   }
   //------------------------
```

Comments on Code Sample 2-6

- *OnTriggerEnter* is called once automatically by Unity each time the *FPSController* intersects the Coin collider
- In Unity, class variables declared as public (such as `public float MaxTime`) are displayed as editable fields inside the Object Inspector of the editor. This applies only to a range of supported data types, however, but it's a highly useful feature. It means developers can monitor and set public variables for classes directly from the inspector, as opposed to changing and recompiling code every time a change is needed. Private variables, in contrast, are hidden from the Inspector by default. However, you can force them to be visible, if needed, by using the `SerializeField` attribute. Private variables prefixed with this attribute, such as variable `CountDown`, will display in the Object Inspector just like a public variable, even though the variable's scope still remains private.

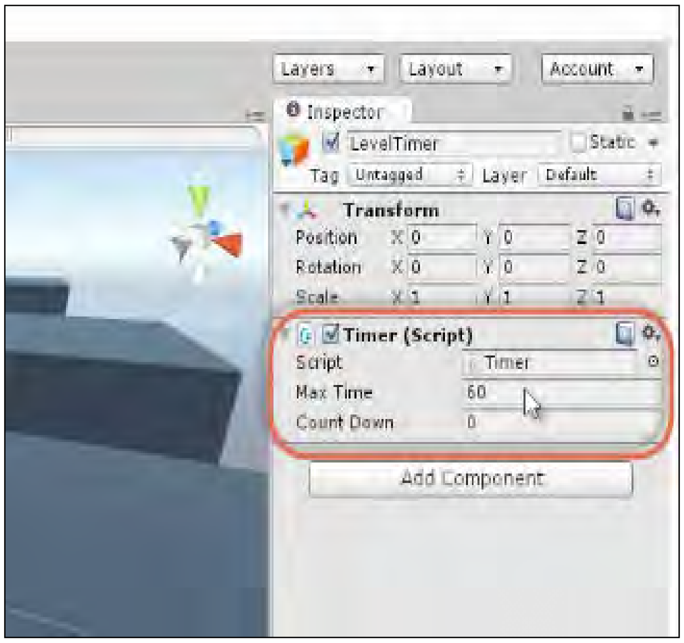
- The `Update` function is a Unity native Event supported for all classes derived from `MonoBehaviour`. Update is invoked automatically *once per frame* for *all active* GameObjects in the scene. This means that all active game objects are notified about frame change events. In short, *Update* is therefore called many times per second; the game FPS is a general indicator as to how many times on each second. The actual number of calls will vary in practice, from second to second. In any case, *Update* is especially useful for animating, updating and changing objects over time. In the case of a CountDown class, it'll be useful for keeping track of time as it passes away, second by second. More information on the Update function can be found at the online Unity documentation here: `https://unity3d.com/learn/tutorials/modules/beginner/scripting/update-and-fixedupdate`

In addition to the `Update` *function, called on each frame, Unity also supports two other related functions, namely* `FixedUpdate` *and* `LateUpdate`*.* `FixedUpdate` *is used when coding with Physics, as we'll see later, and is called a fixed number of times per frame.* `LateUpdate` *is called once per frame for each active object, but the* `LateUpdate` *call will always happen after every object has received an* `Update` *event. Thus it happens after the* `Update` *cycle; making it a late update. There are reasons for this late update, and we'll see them later in the book. More information on* `FixedUpdate` *can be found in the Online Unity Documentation here:* `http://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html`*. More information on the LateUpdate function can be found in the Online Unity Documentation here:* `http://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html`

- When scripting, the static *Time.deltaTime* variable is constantly available and is updated automatically by Unity. It always describes the amount of time (in seconds) that has passed since the previous frame ended. For example, if your game has a frame rate of 2 FPS (a very low frame rate!) then `deltaTime` will be 0.5. This is because, in each second, there would be two frames, and thus each frame would be half a second. `deltaTime` is useful because, if added over time, it tells you how much time in total has elapsed or passed since the game began. For this reason, deltaTime is used heavily inside the `Update` function for the Timer, to subtract the elapsed time from the countdown total. More information can be found on deltaTime at the online documentation here: `http://docs.unity3d.com/ScriptReference/Time-deltaTime.html`

- The static function `Application.LoadLevel` may be called anywhere in code to change the active scene at run time. Thus, this function is useful for moving the gamer from one level to another. It causes Unity to terminate the active, destroying all its contents, and to load in a new scene. It can also be used to restart the active scene, simply by loading the active level again. `Application.LoadLevel` is most appropriate for games with clearly defined levels that a separate from each other and have clearly defined beginnings and endings. It is not, however, suitable for large open-world games in which large sprawling environments stretch on, seemingly without any breakage or disconnection. More information on Application.LoadLevel can be found at the online Unity Documentation here: `http://docs.unity3d.com/ScriptReference/Application.LoadLevel.html`

After the timer script is created, select the LevelTimer object in the scene. From the Object Inspector, you can set the maximum time (in seconds) the player is allowed for completing the level. See Figure 2.28. I've set the total time to 60 seconds. This means all coins must be completed within 60 seconds from the level start. If the timer expires, the level is restarted.



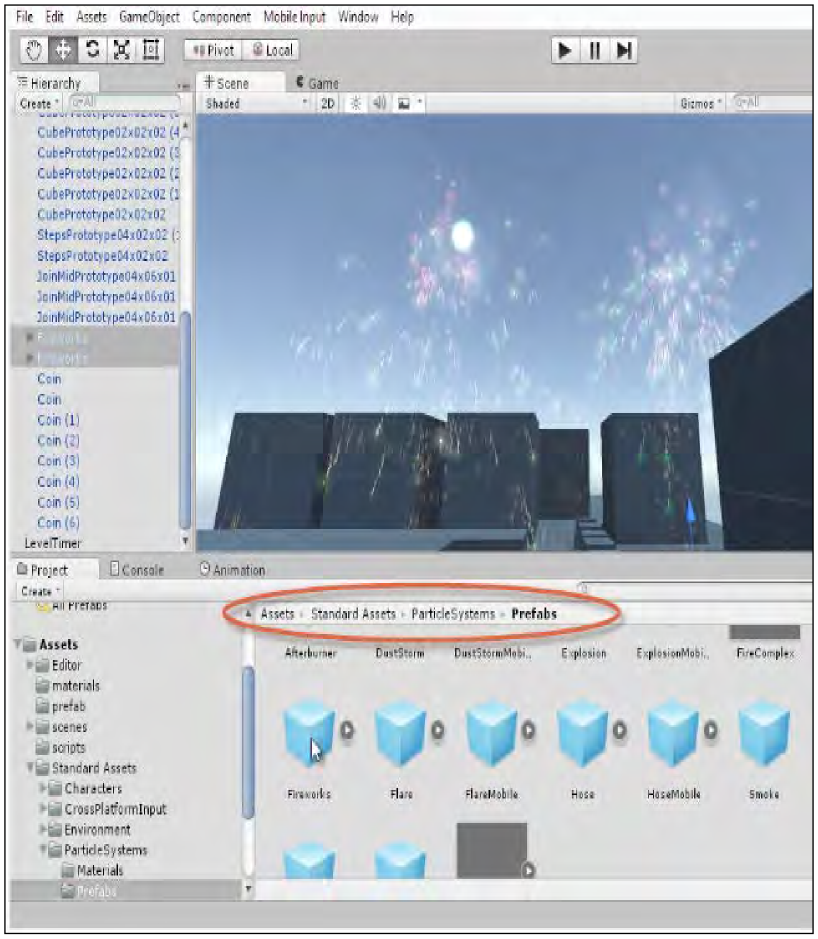
LevelTimer
Tag Untagged Layer Default
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Timer (Script)
Script Timer
Max Time 60
Count Down 0

Setting the level total time

Great work! You should now have a completed level with a countdown that works. You can collect coins, and the timer can expire. There is a further problem, however, which we'll address next. But overall, the game is taking shape.

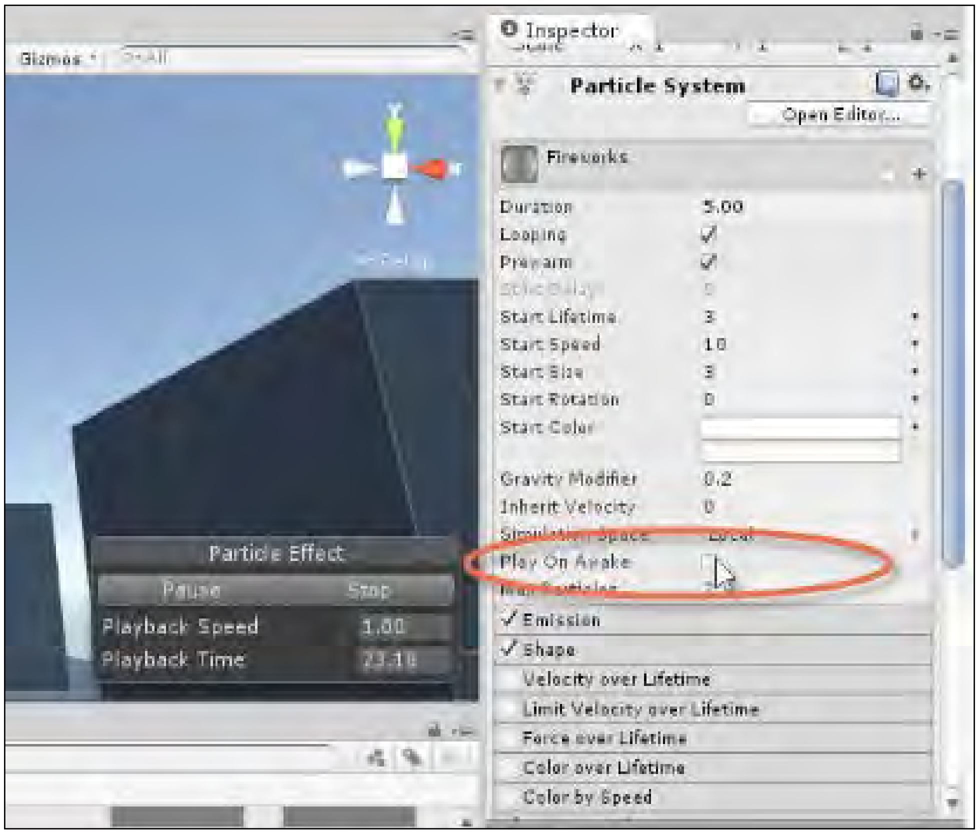
Celebrations and Fireworks!

The coin collection game is nearly finished. Coins can be collected and a timer expires, but the win condition itself is not truly handled. That is, when all coins are collected before time expiry, nothing actually happens to show the player they've won. The countdown still proceeds and even restarts the level as though the win condition hadn't been satisfied at all. Let's fix that now. Specifically, when the win scenario happens, we should delete the timer object to prevent further countdown, and show visual feedback to signify that the level has been completed. In this case, I'll add some fireworks! So, let's start by creating the fireworks. You can add these easily from the Unity 5 Particle System packages. Open the folder *Standard Assets* > *ParticleSystems* > *Prefabs*. Then drag and drop the Fireworks Particle System into the scene. Add a second, or even a third one if you want.



Adding two Fireworks Prefabs

By default, all firework particle systems will play when the level begins. You can test that by pressing *Play* on the toolbar. This is not the behavior we want. We only want the fireworks to play when the win condition has been satisfied. To disable playback on level start-up, select the Particle System object in the scene, and from the Object Inspector disable the *Play on Awake* check box, which can be found in the Particle System Component. See Figure 2.30.



Fireworks

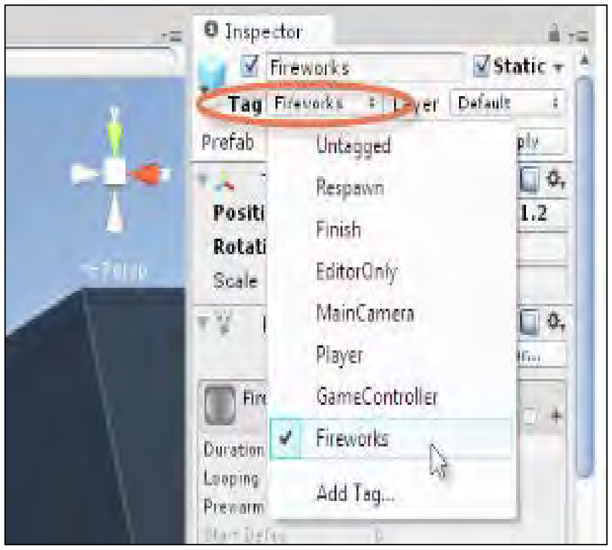
Duration	5.00
Looping	✓
Prewarm	✓
Start Delay	0
Start Lifetime	3
Start Speed	10
Start Size	3
Start Rotation	0
Start Color	
Gravity Modifier	0.2
Inherit Velocity	0
Simulation Space	Local
Play On Awake	

✓ Emission
✓ Shape
Velocity over Lifetime
Limit Velocity over Lifetime
Force over Lifetime
Color over Lifetime
Color by Speed

Playback Speed 1.00
Playback Time 23.18

Disabling Play on Awake

Disabling *Play on Awake* prevents particle systems playing automatically at level start-up. This is fine, but if they are ever to play at all something must manually start them at the right time. We can achieve this through code. Before resorting to a coding solution, however, we'll first mark all firework objects with an appropriate tag. The reason for this is that, in code, we'll want to search for all firework objects in the scene and trigger them to play when needed. To isolate the firework objects from all other objects, we'll use tags. So, let's create a new Firework tag and assign them to only the firework objects in the scene. Tags were created earlier in this chapter when configuring the player character for coin collisions. See Figure 2.31.



Tagging firework objects

With the firework objects now tagged, we can refine the *Coin.cs* script class to handle a win condition for the scene, as shown in Code Sample 2.7. Comments follow.

```
//-------------------------
using UnityEngine;
using System.Collections;
//-------------------------
public class Coin : MonoBehaviour
{
   //-------------------------
   public static int CoinCount = 0;
   //-------------------------
   // Use this for initialization
   void Awake ()
   {
         //Object created, increment coin count
         ++Coin.CoinCount;
   }
   //-------------------------
   void OnTriggerEnter(Collider Col)
   {
         //If player collected coin, then destroy object
         if(Col.CompareTag("Player"))
               Destroy(gameObject);
   }
```

```
    //------------------------
    void OnDestroy()
    {
          --Coin.CoinCount;

          //Check remaining coins
          if(Coin.CoinCount <= 0)
          {
                //Game is won. Collected all coins
                //Destroy Timer and launch fireworks
                GameObject Timer = GameObject.Find("LevelTimer");
                Destroy(Timer);

                GameObject[] FireworkSystems = GameObject.FindGameObj
ectsWithTag("Fireworks");
                foreach(GameObject GO in FireworkSystems)
                      GO.GetComponent<ParticleSystem>().Play();
          }
    }
    //------------------------
}
//------------------------
```

Comments on Code Sample 2-7

- The *OnDestroy* function is critical. It occurs when a coin is collected, and it features an *if* statement to determine when all coins are collected (the win scenario).
- When a win scenario happens, the function `GameObject.Find` is called to search the complete scene hierarchy for any active object named "LevelTimer". If found, the object is deleted. This happens to delete the timer and prevent any further count down when the level is won. If the scene contains multiple objects of matching name, then only the first object is returned. This is one reason why the scene should contain one and only one timer.

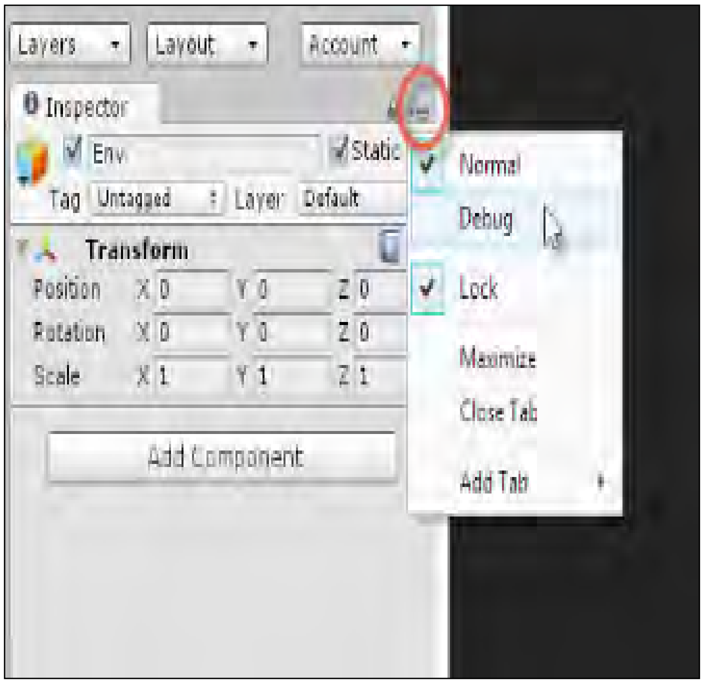
TIP. Avoid using the `GameObject.Find` *function wherever possible. It's slow for performance. Instead, use* `FindGameObjectsWithTag` *instead. It's been used here only to demonstrate its existence and purpose. Sometimes, you'll need to use it for finding a single, miscellaneous object that has no specific tag.*

- In addition to deleting the LevelTimer object, the OnDestroy function all finds all firework objects in the scene and initiates them. It finds all objects of a matching tag by using the `GameObject.FindGameObjectsWithTag` function. This function returns an array of all objects with the "Fireworks" tag, and the ParticleSystem is initiated for each object by calling the `Play` function.

As mentioned, each GameObject in Unity is really made from a collection of attached and related components. An object is the sum of its components. For example, a standard cube (created using GameObject > 3D Object > Cube) is made from a Transform Component, a Mesh Filter Component, a Mesh Renderer Component, and a Box Collider Component. These components together make the cube what it is and behave how it does. The `GetComponent` *function can be called in script to retrieve a reference to any specified component, giving you direct access to its public properties. The OnDestroy function in Code Sample 2.7 uses* `GetComponent` *to retrieve a reference to the ParticleSystem component attached to the object. GetComponent is a highly useful and important function. More information on GetComponent can be found at the online Unity Documentation here:* `http://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html`

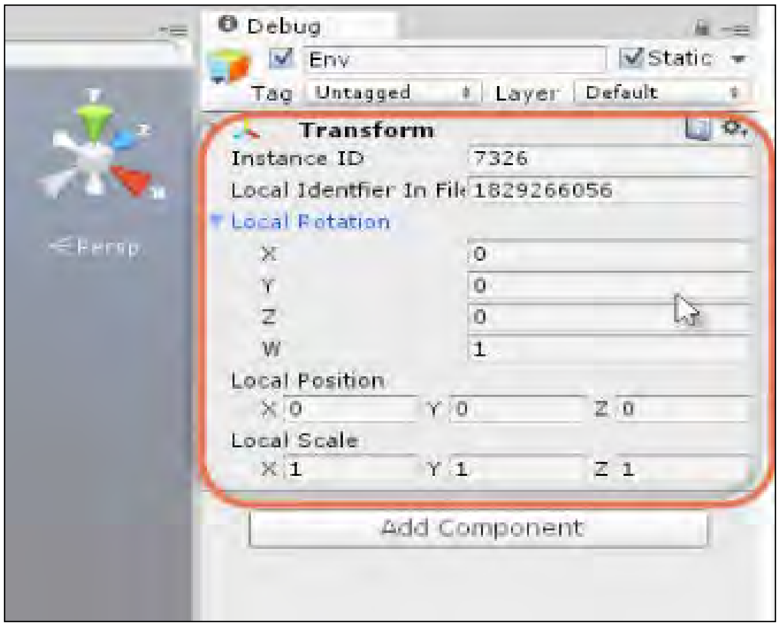
Play Testing

You've now completed your first game in Unity! It's time to take it for a test run, and then finally to build it. Testing in Unity firstly consists of pressing Play on the toolbar and simply playing your game to see that it works as intended, from the perspective of a gamer. In addition to playing, you can also enable Debugging mode from the Object Inspector to keep a watchful eye on all public and private variables during runtime; making sure no variable is assigned an unexpected value. To activate Debug mode, click the menu icon at the top-right corner of the Object Inspector, and from the context menu that appears, select the option *Debug*. See Figure 2.32.



Activating Debug Mode from the Object Inspector

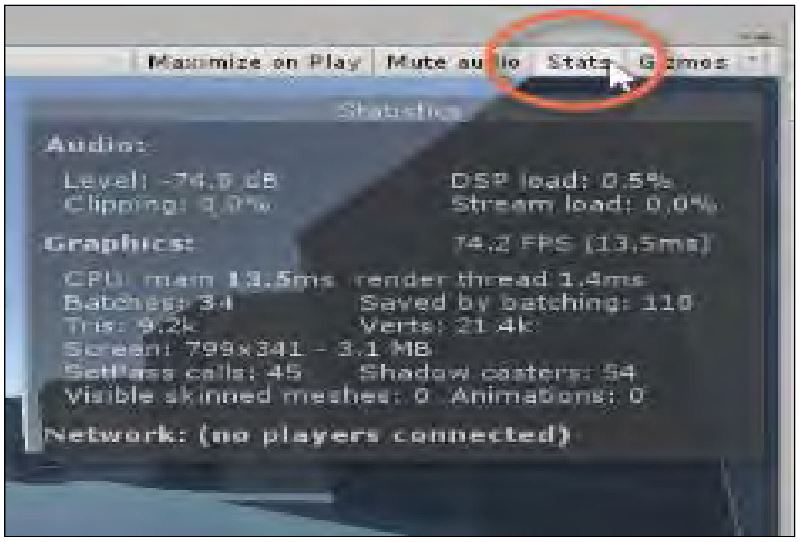
After activating debug mode, the appearance of some variables and components in the Object Inspector may change. Typically, you'll get a more detailed and accurate view of your variables; and you'll also be able to see most private variables. See Figure 2.33 for the Transform Component in Debug Mode.



```
Instance ID                   7326
Local Identfier In Fil        1829266056
Local Rotation
    X                         0
    Y                         0
    Z                         0
    W                         1
Local Position
    X 0          Y 0          Z 0
Local Scale
    X 1          Y 1          Z 1
```

Viewing the Transform Component in Debug Mode

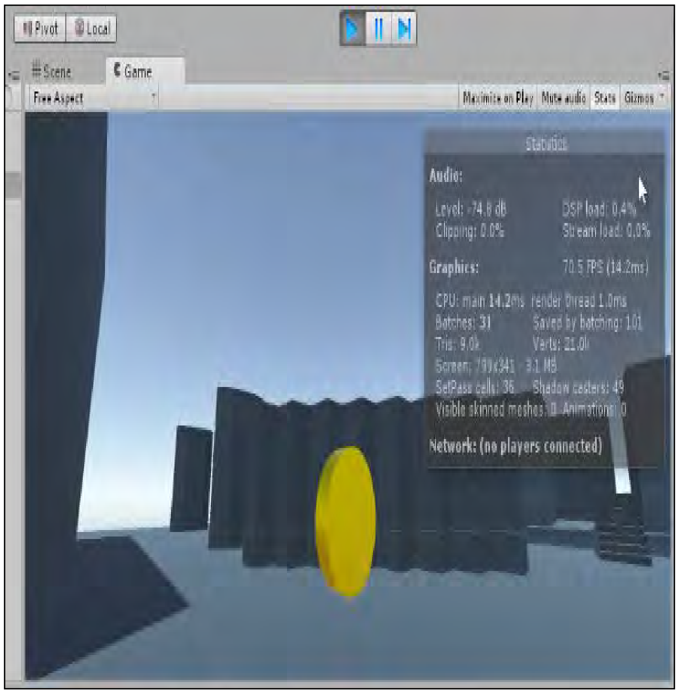
Another useful debugging tool at runtime is the Stats panel. This can be accessed from the Game tab, by clicking the *Stats* button from the toolbar. See Figure 2.34.



```
Statistics
Audio:
  Level: -74.9 dB                 DSP load: 0.5%
  Clipping: 0.0%                  Stream load: 0.0%
Graphics:                         74.2 FPS (13.5ms)
  CPU: main 13.5ms  render thread 1.4ms
  Batches: 34              Saved by batching: 110
  Tris: 9.2k               Verts: 21.4k
  Screen: 799x341 - 3.1 MB
  SetPass calls: 45        Shadow casters: 54
  Visible skinned meshes: 0  Animations: 0
Network: (no players connected)
```

Accessing the Stats panel from the Game Tab

The Stats Panel is only useful during *Play* mode. In this mode, it details the critical performance statistics for your game, such as Frame Rate (FPS) and memory usage. This lets you diagnose or determine whether any problems may be affecting your game. The FPS represents the total number of frames (ticks or cycles) per second that your game can sustain on average. There is no right or wrong or magical FPS per se; but higher values are better than lower ones. Higher values represent better performance, because it means your game can sustain more cycles in one second. If your FPS falls below 20 or 15, it's likely your game will appear choppy or 'laggy', as the performance weight of each cycle means it takes longer to process. Many variables can affect FPS, some internal and some external to your game. Internal factors include the number of lights in a scene, the vertex density of meshes, the number of instructions and complexity of code. Some external factors include the quality of your computer's hardware, the number of other applications and processes running at the same time, the amount of hard drive space, among others. In short, if your FPS is low, then it indicates a problem that needs attention. The solution to that problem varies depending on the context, and you'll need to use judgement, for example: are your meshes too complex? Do they have too many vertices? Are your textures too large? Are there too many sounds playing? See Figure 2.35 for the coin collection game up and running. The completed game can be found in the book companion files, in the *Chapter02/End* folder.

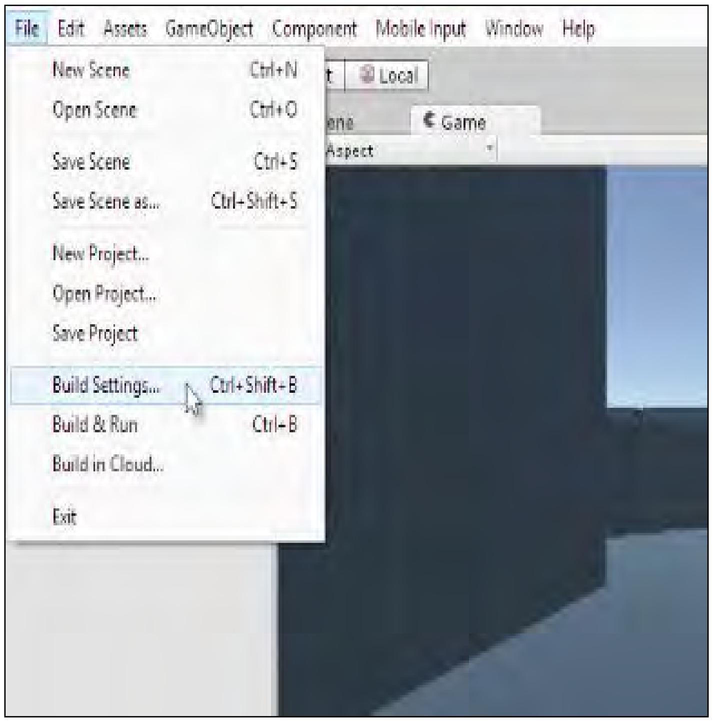


```
Statistics
Audio:
  Level: -74.8 dB            DSP load: 0.4%
  Clipping: 0.0%             Stream load: 0.0%
Graphics:                    70.5 FPS (14.2ms)
  CPU: main 14.2ms  render thread 1.0ms
  Batches: 31       Saved by batching: 101
  Tris: 9.0k        Verts: 21.0k
  Screen: 799x341 - 3.1 MB
  SetPass calls: 36  Shadow casters: 49
  Visible skinned meshes: 0  Animations: 0
Network: (no players connected)
```

Testing the Coin Collection Game...

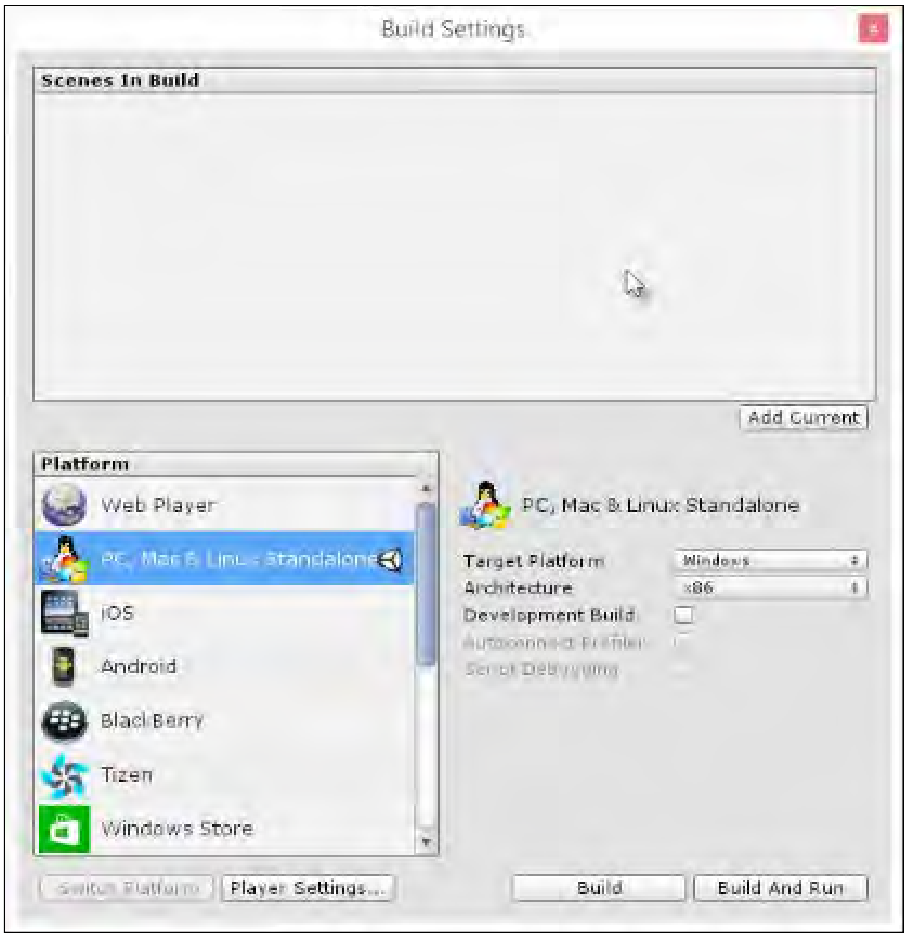
Building

So now it's time to **Build** the game! That is, to compile and package the game into a stand-alone and self-executing form, which the gamer can run and play without needing to use the Unity editor. Typically, when developing games you'll reach a decision about your target platform (such as Windows, iOS, Android etc.) during the design phase, and not at the end of development. It's often said that Unity is a 'develop once, deploy everywhere' tool. This slogan can conjure up the unfortunate image that, after a game is made, it'll work just as effortlessly on every platform supported by Unity as it does on the desktop. Unfortunately, things are not so simple: games that work well on desktop systems don't necessarily perform equally well on mobiles, and vice versa. This is due largely to the great differences in target hardware, and in the industry standards that hold between them. Because of these differences, I'll focus our attention here to the Windows and Mac Desktop platforms, ignoring mobiles and consoles and other platforms. To create a Build for Desktop platforms, select *File* > *Build Settings* from the File menu.



Accessing the Build Settings for the Project

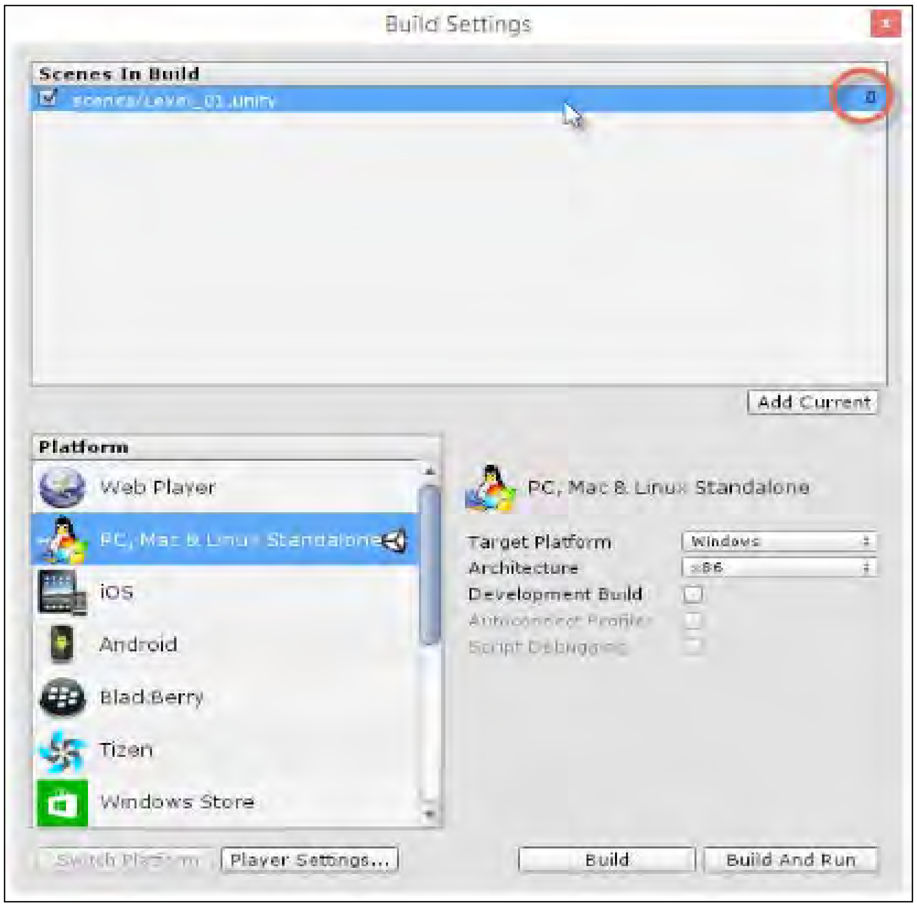
The Build Settings dialog then displays, and its interface consists of three main areas. The *Scenes in Build* list is a complete list of *all* scenes to be included in the build, regardless of whether the gamer will actually visit them in game. It represents the totality of all scenes that could ever be visited in the game. In short, if you want or need a scene in your game, then it needs to be in this list. Initially, the list is empty. See Figure 2.37.



Target Platform Windows
Architecture x86
Development Build
Autoconnect Profiler
Script Debugging

The Build Settings Dialog

You can easily add scenes to the list, simply by dragging and dropping the scene asset from the Project Panel into the *Scenes in Build* list. For the coin collection game, I'll drag and drop the *Level_01* scene into the list. As scenes are added, Unity automatically assigns them a number, depending on their order in the list. 0 represents the topmost item, 1 the next item, and so on. This number is important insofar as the 0 item is concerned. The topmost scene (Scene 0) will always be the starting the scene. That is, when the build runs, Unity automatically begins execution from Scene 0. Thus Scene 0 will typically be your Splash or Intro scene. See Figure 2.38.

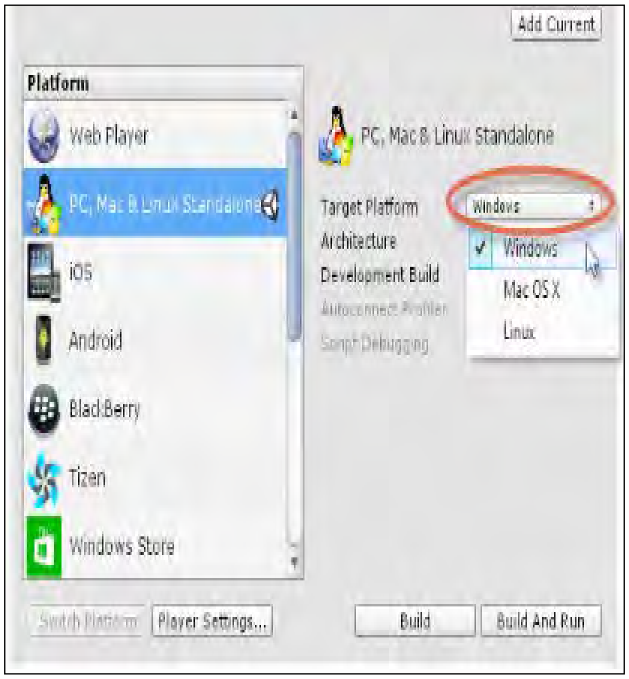

Build Settings

Scenes In Build
✓ scenes/Level_01.unity 0

Target Platform Windows
Architecture x86
Development Build ☐
Autoconnect Profiler ☐
Script Debugging ☐

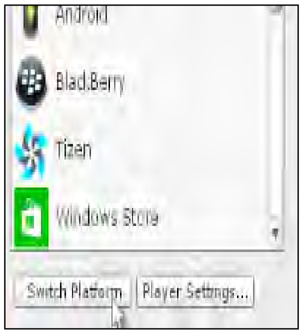
Adding a Level to the Build Settings Dialog

Next, be sure to select your target platform from the Platform list at the bottom-left side of the Build Settings dialog. For desktop platforms, choose *Pc, Mac & Linux Standalone*, which should be selected by default. And then from the options, set the *Target Platform* drop down list to either Windows, Linux or Mac, depending on your system. See Figure 2.39.



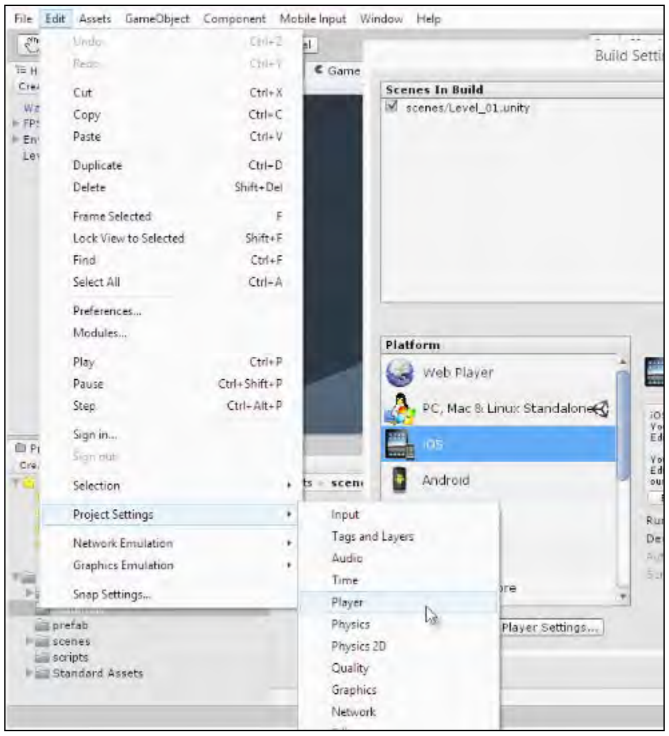
Choosing a Target Build Platform

If you've previously been testing your game for multiple platforms, or trying out other platforms, like Android and iOs, the button *Switch Platform* (at the bottom-left of the Build Settings dialog) might become activate when you select the Standalone option. If it does, click the *Switch Platform* button to confirm to Unity that you intended building for the selected platform. On clicking this, Unity may spend a few minutes configuring your assets for the selected platform.



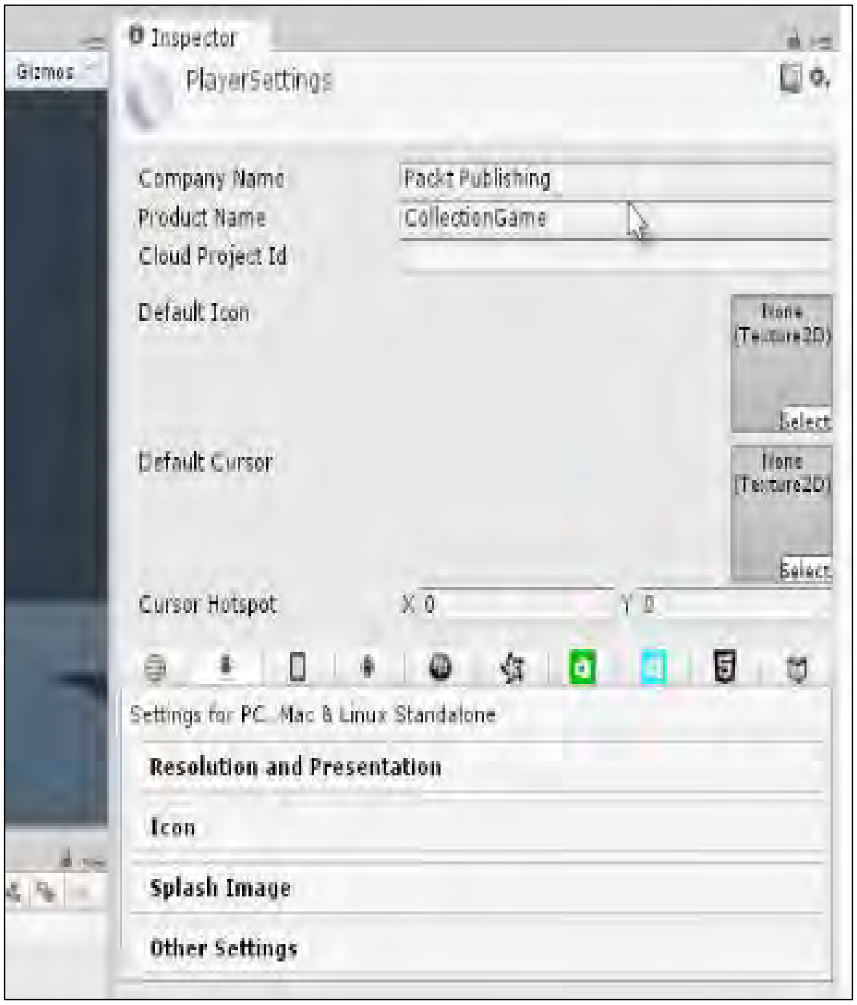
Switching platforms...

Before building for the first time, you'll probably want to view the *Player Settings* options to fine tune important build parameters, such as game resolution, quality settings, executable icon and information, among other settings. To access the Player Settings, you can simply click the *Player Settings* button from the Build Dialog. This displays the Player Settings inside the Object Inspector. The same settings can also be accessed via the application menu, by choosing *Edit* > *Project Settings* > *Player*. See Figure 2.41.



Accessing the Player Settings options

From the Player Settings options, set a *Company Name* and *Product Name*, as this information is baked and stored within the built executable. You can also specify an icon image for the executable, as well as a default mouse cursor, if one is required. For the collection game, however, these latter two settings will be left empty. See Figure 2.42.

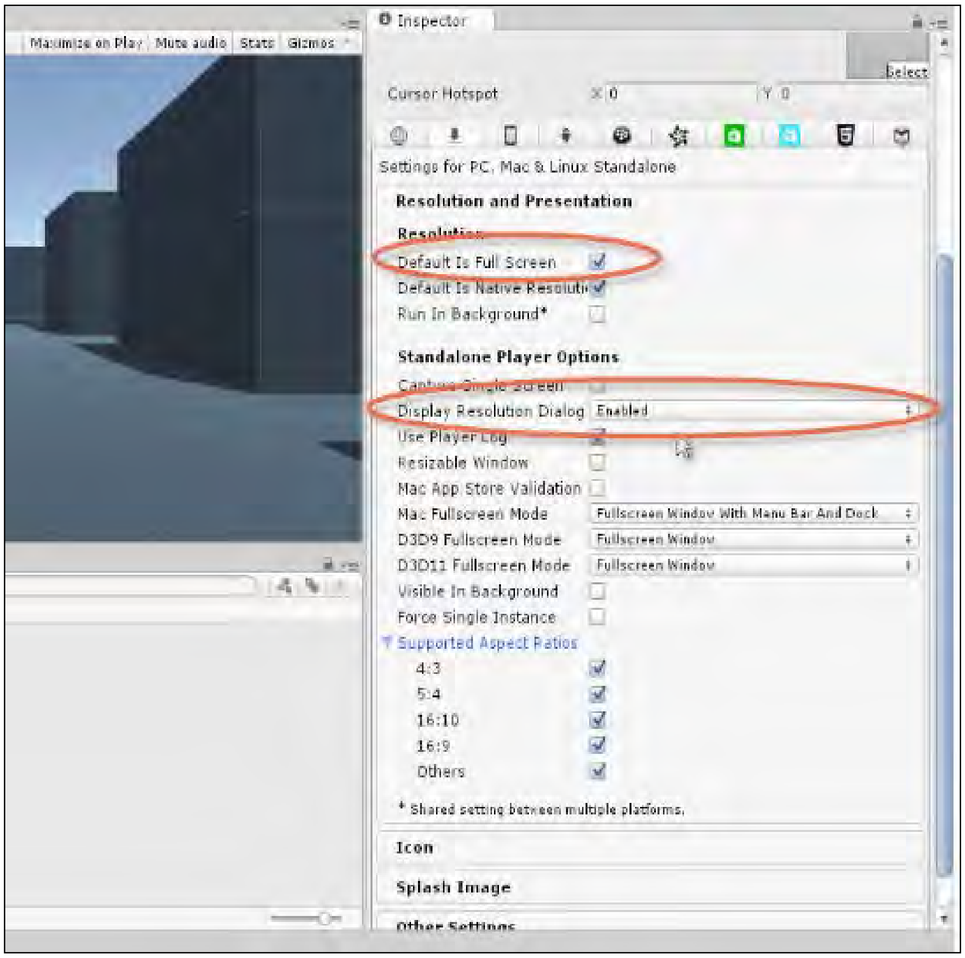


Company Name: Packt Publishing
Product Name: CollectionGame
Cloud Project Id
Default Icon: None (Texture2D)
Default Cursor: None (Texture2D)
Cursor Hotspot: X 0 Y 0

Settings for PC, Mac & Linux Standalone

Setting an Publisher and Product name...

The Resolution and Presentation tab is especially important, as it specifies the game screen size and whether a default splash screen (Resolution Dialog) should appear at application start-up. From this tab, ensure the option *Default is Full Screen* is enabled, meaning the game will run at the complete size of the system's screen, as opposed to in a smaller and movable window. In addition, enable the drop-down list *Display Resolution Dialog*. See Figure 2.43. When this is enabled, your application will display an options screen at start-up, allowing the user to select a target resolution and screen size, and to customize controls. For a final build, you'll probably want to disable this option, presenting the same settings through your own customized options screen in-game instead. But for test builds, the Resolution Dialog can be a great help. It lets you test your build easily at different sizes.



Cursor Hotspot X 0 Y 0

Settings for PC, Mac & Linux Standalone

Resolution and Presentation

Resolution

Default Is Full Screen ✓
Default Is Native Resoluti ✓
Run In Background* ☐

Standalone Player Options

Capture Single Screen ☐
Display Resolution Dialog Enabled
Use Player Log ✓
Resizable Window ☐
Mac App Store Validation ☐
Mac Fullscreen Mode Fullscreen Window With Menu Bar And Dock
D3D9 Fullscreen Mode Fullscreen Window
D3D11 Fullscreen Mode Fullscreen Window
Visible In Background ☐
Force Single Instance ☐

Supported Aspect Ratios

4:3 ✓
5:4 ✓
16:10 ✓
16:9 ✓
Others ✓

* Shared setting between multiple platforms.

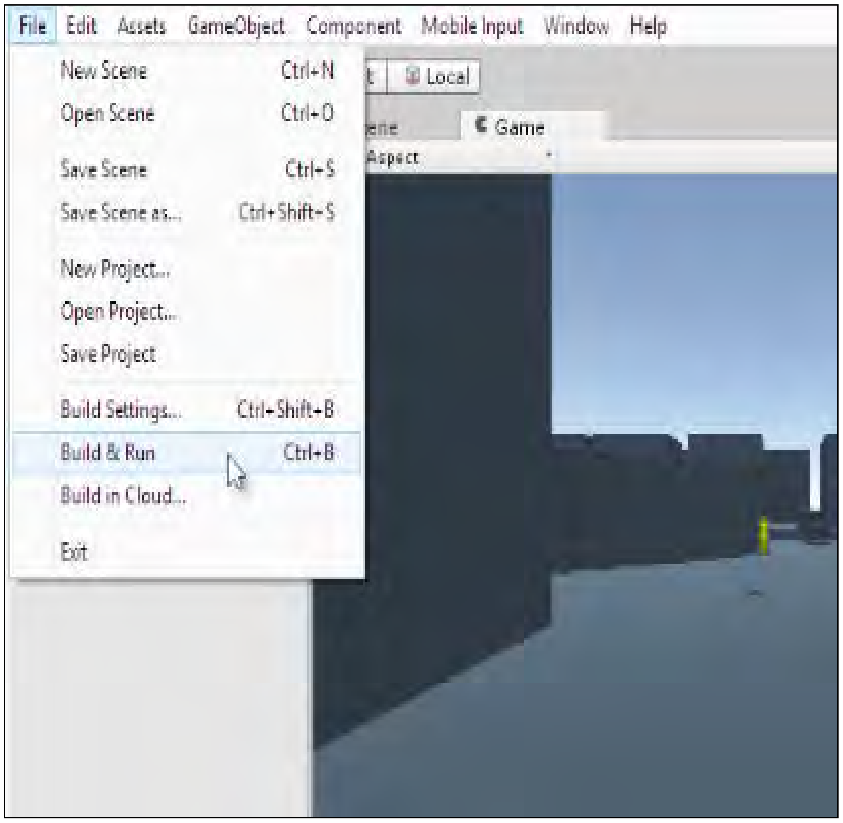
Icon

Splash Image

Other Settings

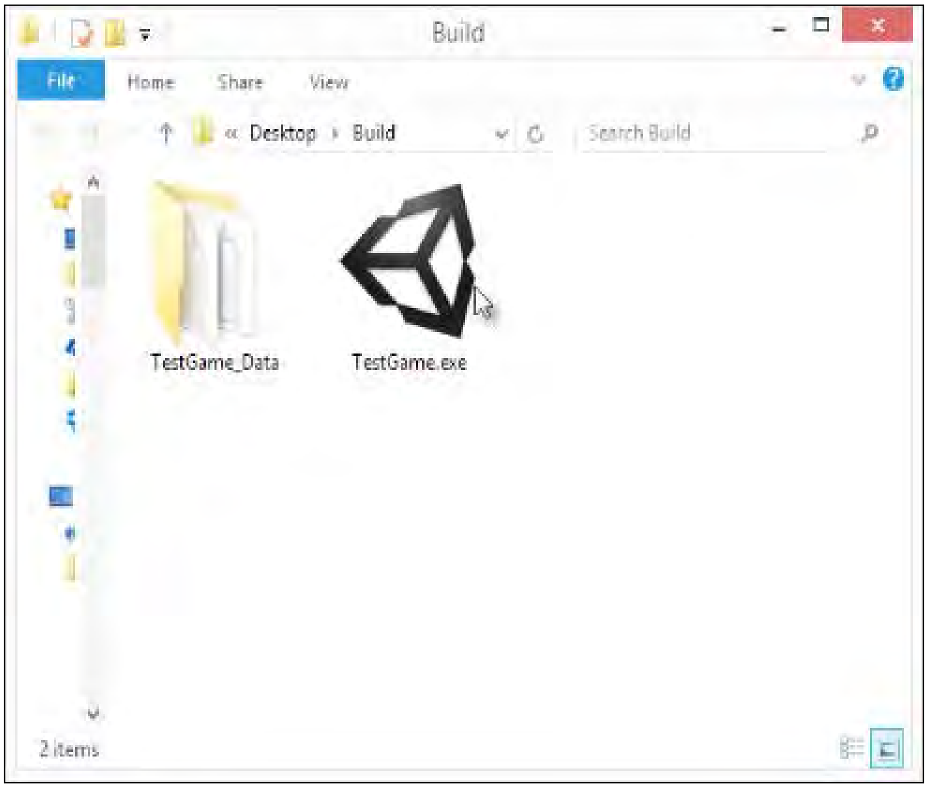
Enabling the Resolution Dialog

Now you're ready to make your first compiled Build. So click the *Build* button from the *Build Settings* Dialog, or else choose *File* > *Build and Run* from the application menu. When you do this, Unity presents you with a Save Dialog, asking you to specify a target location on your computer where the Build should be made. Select a location and choose *Save*, and the build process will complete. Occasionally, this process can generate errors, which are printed in red inside the Console Window. This can happen, for example, when you save to a read-only drive, or have insufficient hard drive space, or don't have the necessary administration privileges on your computer. But generally, the Build Process succeeds if your game runs properly in the Editor. See Figure 2.44.



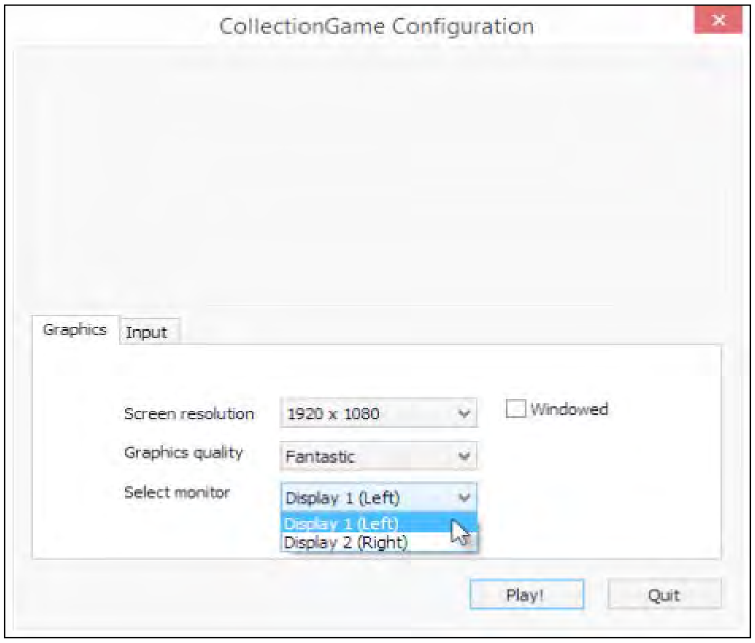
Building and Running a Game

After the Build is completed, Unity generates new files at your destination location. For Windows, it generates an executable file, and a Data folder. See Figure 2.45. Both are essential and interdependent. That is, if you want to distribute your game and have other people play it without needing to install Unity, then you'll need to send users both the executable file and the associated data folder and all its contents.



Unity builds several files

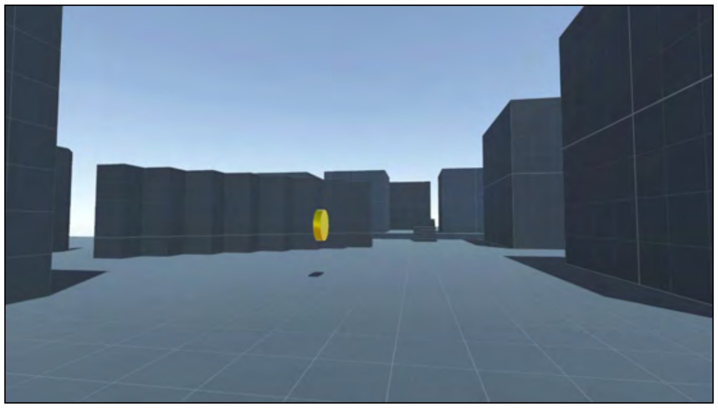
On running your game, the Resolution Dialog will show, assuming you enabled the *Show Resolution Dialog* option from the Player Settings. From here, users can select game resolution, quality, output monitor, and can configure player controls.



Screen resolution: 1920 x 1080
Windowed [ ]
Graphics quality: Fantastic
Select monitor: Display 1 (Left)
Display 1 (Left)
Display 2 (Right)

Preparing to run your game from the Resolution Dialog

On clicking the *Play!* Button, your game will run by default in full screen mode. Congratulations! Your game is now completed and built, and you can send it to your friends and family for play testing! See Figure 2.47.



Running the coin collection game in full screen mode

But wait! How do you exit your game when you're finished playing? There's no quit button or main menu option in game. For Windows, you just need to press *Alt*+*F4* on the keyboard. For Mac, you press *Cmd*+*Q*, and for Ubuntu it's *Ctrl*+*Q*.

Summary

Excellent work!On reaching this point you've completed the Coin Collection Game, as well as your first game in you Unity. In achieving this, you've seen a wide range of Unity features, including: level editing and design, prefabs, particle systems, meshes, components, script files, and build settings. That's a lot! Of course, there's a lot more to be said and explored for all these areas, but nevertheless, we've pulled them together to make a game. Next, we'll get stuck in with a different game altogether; and in doing this we'll see a creative reuse of the same features, as well as the introduction of completely new features. In short, we're going to move from the world of beginner level Unity development to intermediate...

3

Project B: The Space Shooter

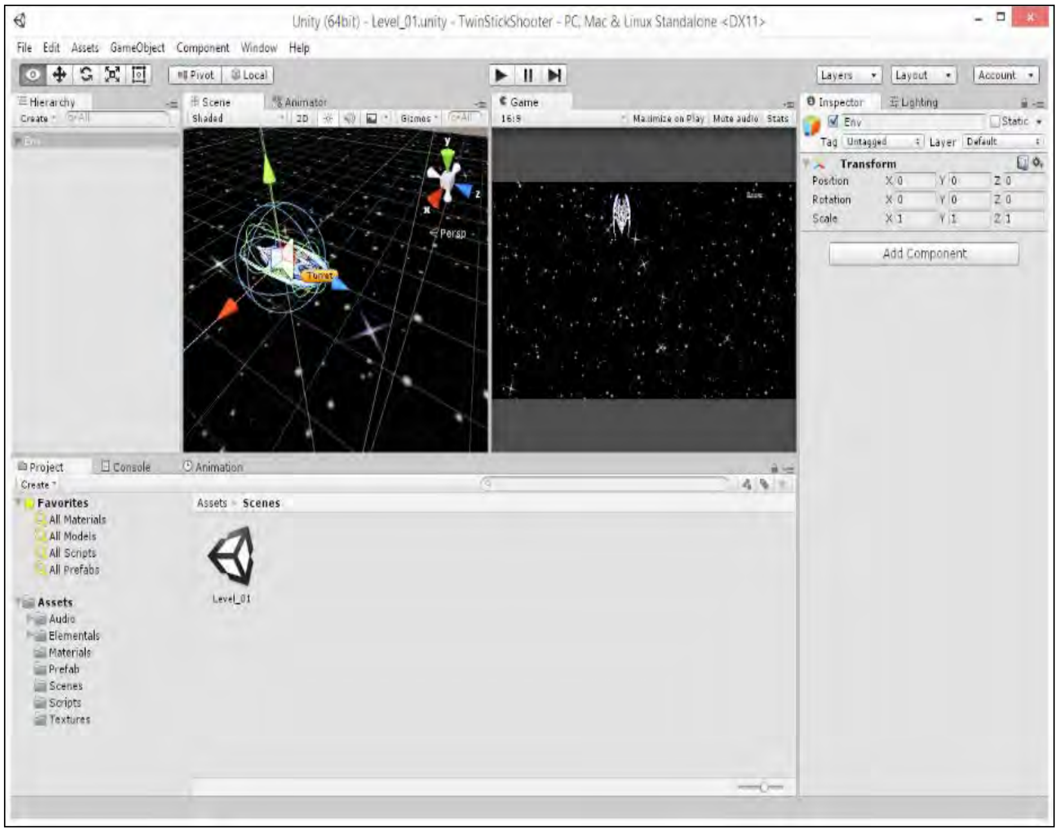
This chapter enters new territory now as we begin development work on our second game, which is a twin-stick space shooter. The twin-stick genre simply refers to any game in which the player input for motion spans two dimensions or axes, typically one axis for movement and one for rotation. Example twin-stick games include: *Zombies Ate My Neighbors* and *Geometry Wars*. Our game will rely heavily on coding in C#, as we'll see. The primary purpose of this is to demonstrate by example just how much can be achieved with Unity procedurally (that is, via script), even without using the editor and level building tools. We'll still use those tools to some extent, but not as much here, and that's a deliberate and not an incidental move. Consequently, this chapter assumes you not only completed the game project created in the previous two chapters, but have a good, basic knowledge of C# scripting generally, though not necessarily inside Unity. So let's roll up our sleeves, if we any, and get stuck in making a twin-stick shooter. This chapter covers the following important topics, as well as others:

1. Spawning and Prefabs
2. Twin-Stick Controls and Axial Movement
3. Player Controllers and Shooting Mechanics
4. Basic Enemy Movement and AI

Remember to see the game created here, and its related work, in abstract terms; that is, as general tools and concepts with multiple applications. For your own projects, you may not want to make a twin-stick shooter, and that's fine. I cannot possibly know every kind of game you want to make. But it's important to see the ideas and tools used here as being transferrable; as being the kind of things you can creative use differently for your own games. Being able to see this is very important when working with Unity.

Looking Ahead – The Completed Project

Before getting stuck in with the Twin Stick Shooter game, let's see what the completed project looks like and how it works. See Figure 3.1. The game to be created will contain one scene only. In that scene the player controls a space ship that can shoot oncoming enemies. The directional keyboard arrows, and WASD, move the space ship around the level, and it will always turn to face the mouse pointer. And clicking the left mouse button will fire ammo. 00000000000



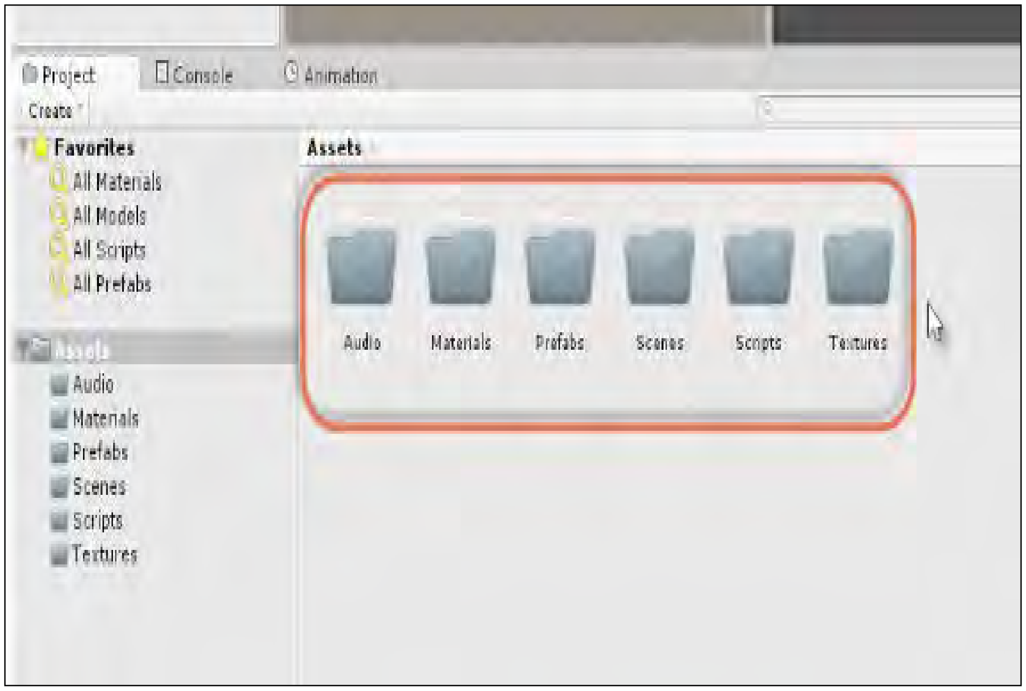
The completed twin stick shooter game

The completed 'Twin Stick Shooter' project, as discussed in this chapter and the next, can be found in the book companion files, inside the *Chapter03/TwinStickShooter* folder.

Most assets for this game (including sound and textures) were sourced from the freely accessible site `OpenGameArt.org` Here you can find many game assets available through the public domain, or creative commons licenses, or other licenses.

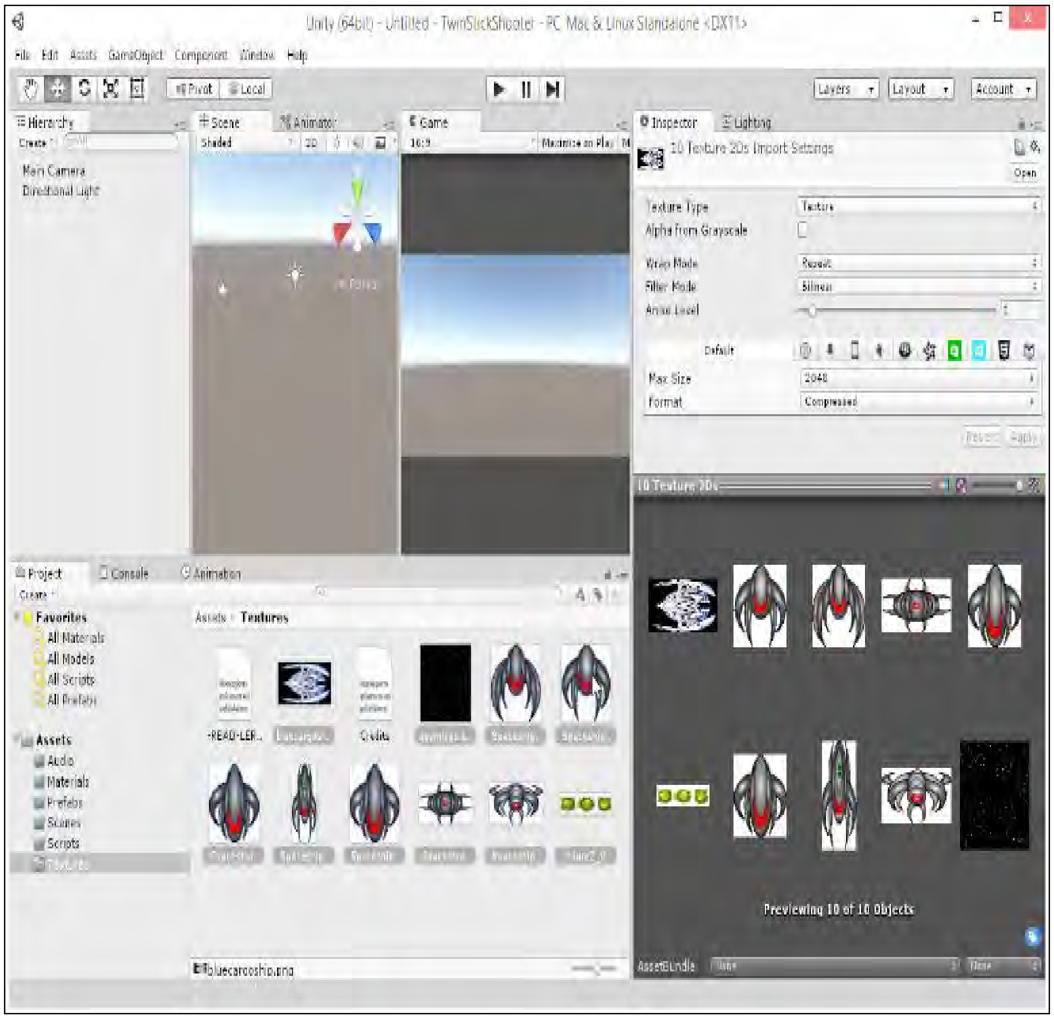
Getting Started with a Space Shooter

To get started, create a blank Unity 3D project without any packages or specific assets. Details about creating new projects is included in Chapter 1. We'll be coding everything from scratch this time around. Once a project is generated, create some basic folders to structure and organize the project assets from the outset. This is very important for keeping track of your files as you work. Create folders for Textures, Scenes, Materials, Audio, Prefabs, and Scripts. See Figure 3.2.



Create folders for structure and organization

Next, our game will depend on some graphical and audio assets. These are included in the book companion files in the *Chapter03/Assets* folder, but can also be downloaded online from `OpenGameArt.org`. Let's start with textures for the player space ship, enemy space ships, and star-field background. Drag and drop the textures from Windows Explorer or Finder into the Unity Project Panel, inside the *Textures* folder. Unity imports and configures the textures automatically. See Figure 3.3.



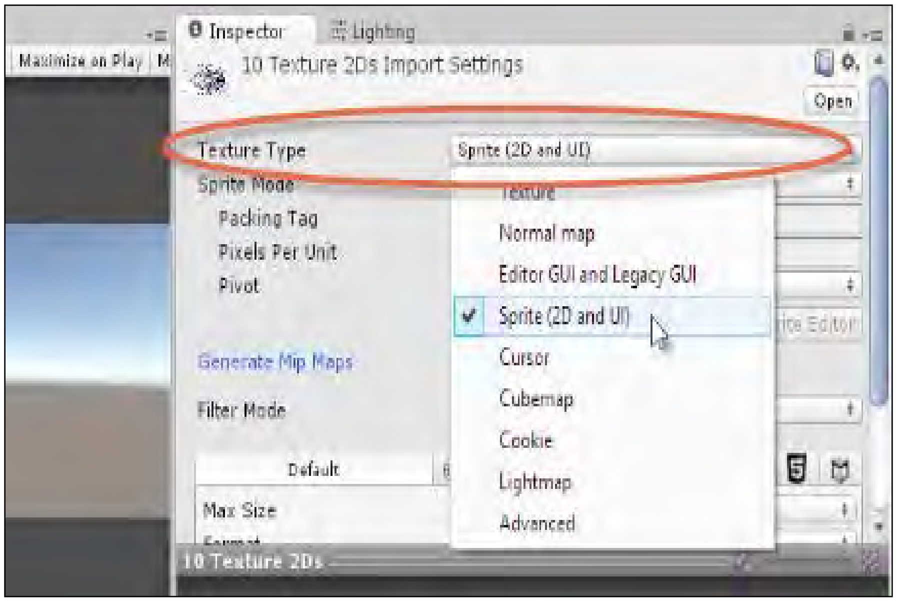
10 Texture 2Ds Import Settings

Texture Type: Texture
Alpha from Grayscale: ☐
Wrap Mode: Repeat
Filter Mode: Bilinear
Aniso Level: 1
Max Size: 2048
Format: Compressed

Importing Texture assets for the space ship, enemies, star background and ammo

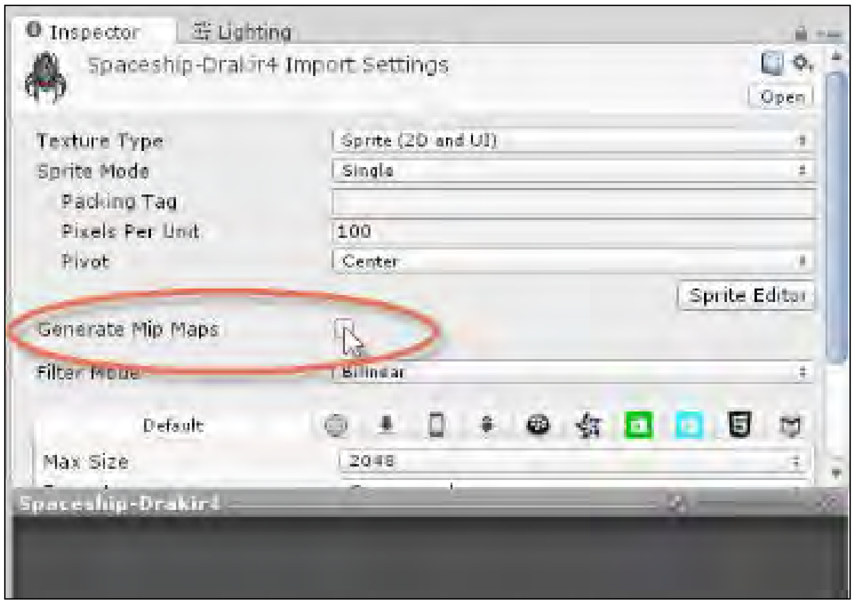
NOTE. Use of the provided assets is optional. You can create your own, if you prefer. Just drag and drop your own textures in place of the included assets, and you can still follow along with the tutorial just fine…

By default Unity imports image files as regular textures for use on 3D objects, and it assumes their pixel dimensions are a power-2 size (4, 8, 16, 32, 64, 128, 256 etc.). If the size is not actually one of these, then Unity will up-scale or down-scale the texture to the nearest valid size. This is not appropriate behavior however for a 2D top-down space shooter game, in which imported textures should appear at their native (imported) size, without any scaling or automatic adjustment. To fix this, select all the imported textures and, from the Object Inspector, change their *Texture Type* from *Texture* to *Sprite (2D and UI)*. Once changed, click the *Apply* button to update the settings, and the textures will retain their imported dimensions. See Figure 3.4.



Changing the Texture Type for imported Textures

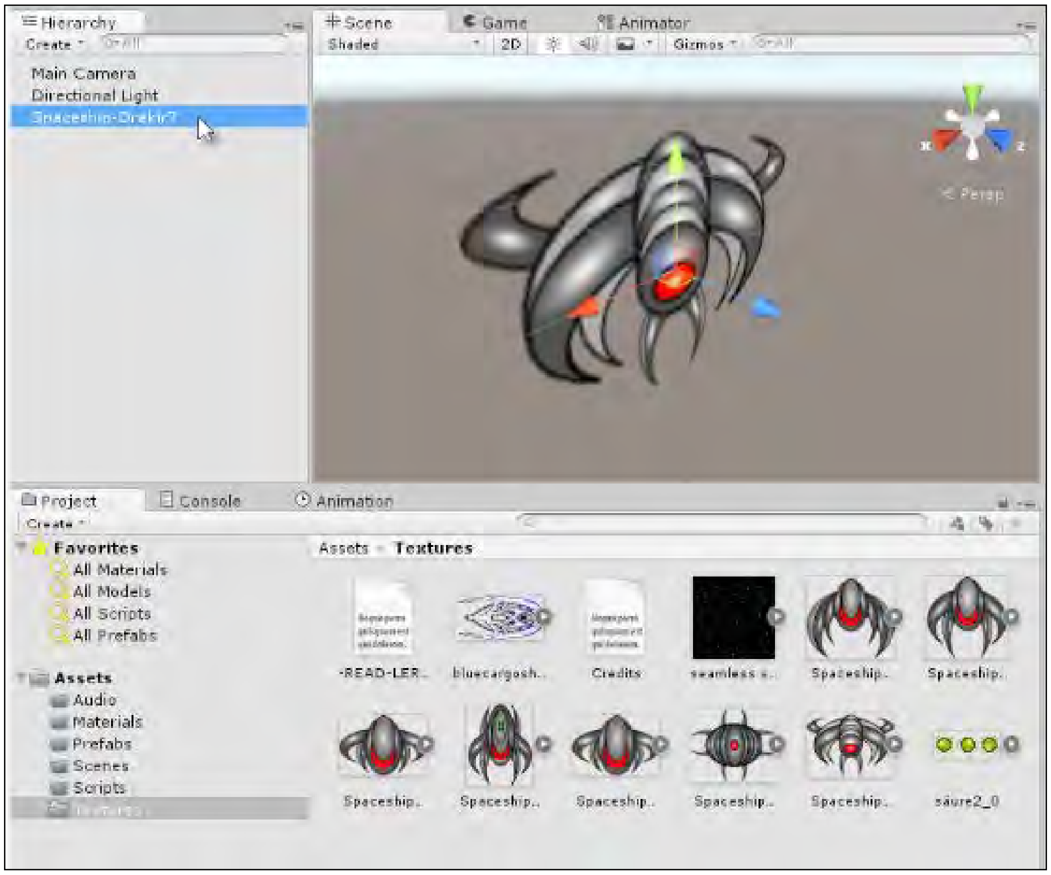
After changing the *Texture Type* setting to *Sprite*, also remove the check mark from the box *Generate Mip Maps*, if this box is enabled. This will prevent Unity from automatically downgrading the quality of textures based on their distance from the camera in the scene. This ensures your textures retain their highest quality. More information 2D Texture settings, and Mip Maps, can be found at the online Unity documentation, available here: `http://docs.unity3d.com/Manual/class-TextureImporter.html`. See Figure 3.5.



Texture Type: Sprite (2D and UI)
Sprite Mode: Single
Packing Tag:
Pixels Per Unit: 100
Pivot: Center
Generate Mip Maps:
Filter Mode: Bilinear
Max Size: 2048

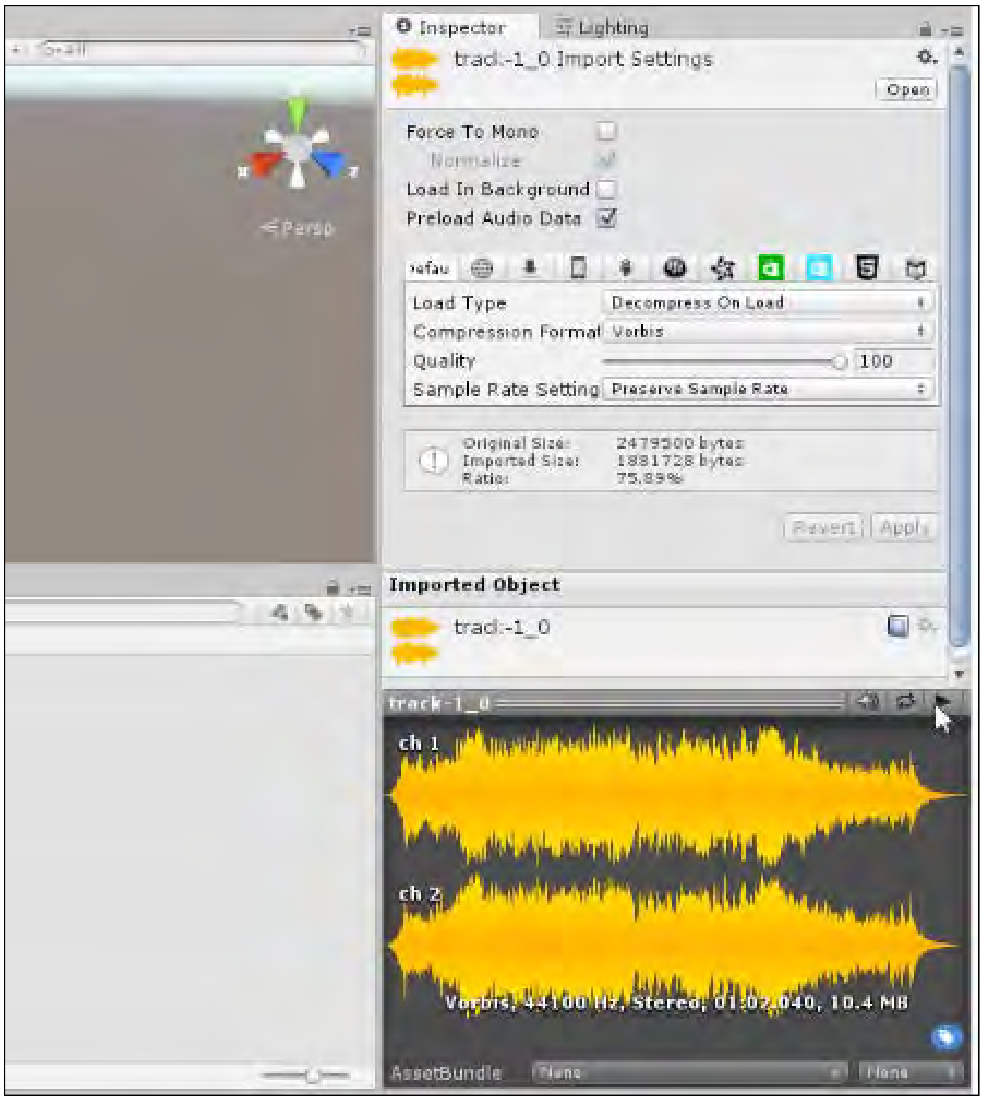
Removing MipMapping from Imported Textures

Now you can easily drag and drop your textures into the scene, adding them as sprite objects. You *can't* drag and drop them from the project panel into the viewport, *but* you can drag and drop them from the Project Panel into the Hierarchy Panel. When you do this, the texture will automatically be added as a sprite object in the scene. We'll make frequent use of this feature as we work for creating space ship objects. See Figure 3.6.



Adding Sprites to the Scene...

Next, let's import music and sound effects, which are also included in the book companion files in the folder *Chapter03/Assets/Audio*. These assets were downloaded from `OpenGameArt.org`. To import the audio, simply drag and drop the files from Windows Explorer or Mac Finder into the Project Panel. When you do this, Unity automatically imports and configures the assets. You can give the audio a test from within the Unity Editor, by pressing *Play* on the preview tool bar from the Object Inspector. See Figure 3.7.



track-1_0 Import Settings

Force To Mono ☐
Normalize ✓
Load In Background ☐
Preload Audio Data ✓

Load Type: Decompress On Load
Compression Format: Vorbis
Quality: 100
Sample Rate Setting: Preserve Sample Rate

Original Size: 2479500 bytes
Imported Size: 1881728 bytes
Ratio: 75.89%

Imported Object

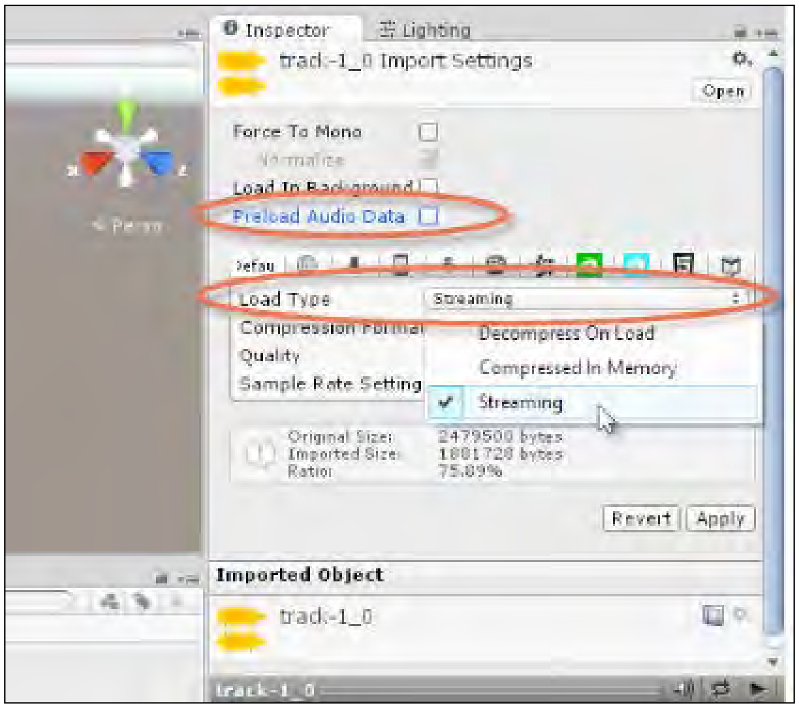
track-1_0

Vorbis, 44100 Hz, Stereo, 01:02.040, 10.4 MB

AssetBundle: None None

Previewing Audio from the Object Inspector

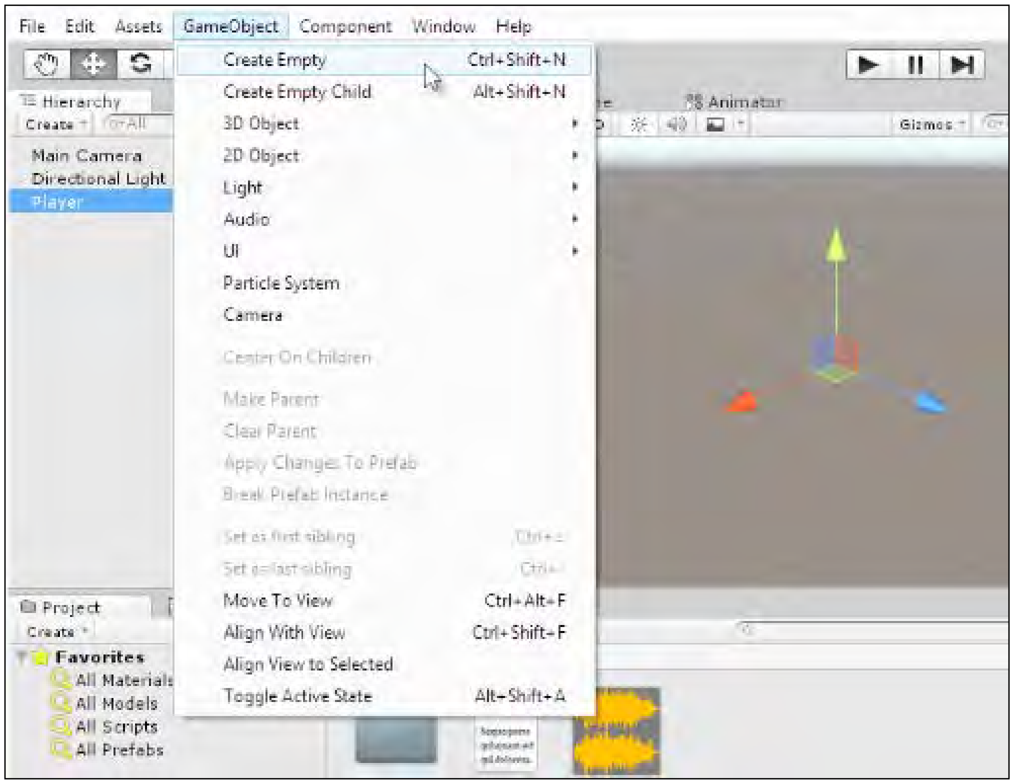
As with texture files, Unity imports audio files using a set of default parameters. These parameters are typically suitable for short sound effects like footsteps, gun shots and explosions, but for longer tracks like music they can be problematic, causing long level loading times. To fix this, select the music track in the Project Panel, and from the Object Inspector disable the check box *Preload Audio Data*. And for the *Load Type* drop down box, select the option *Streaming*. This ensures the music track is streamed as opposed to loaded whole into memory at level start up. See Figure 3.8.



Configuring Music Tracks for Streaming

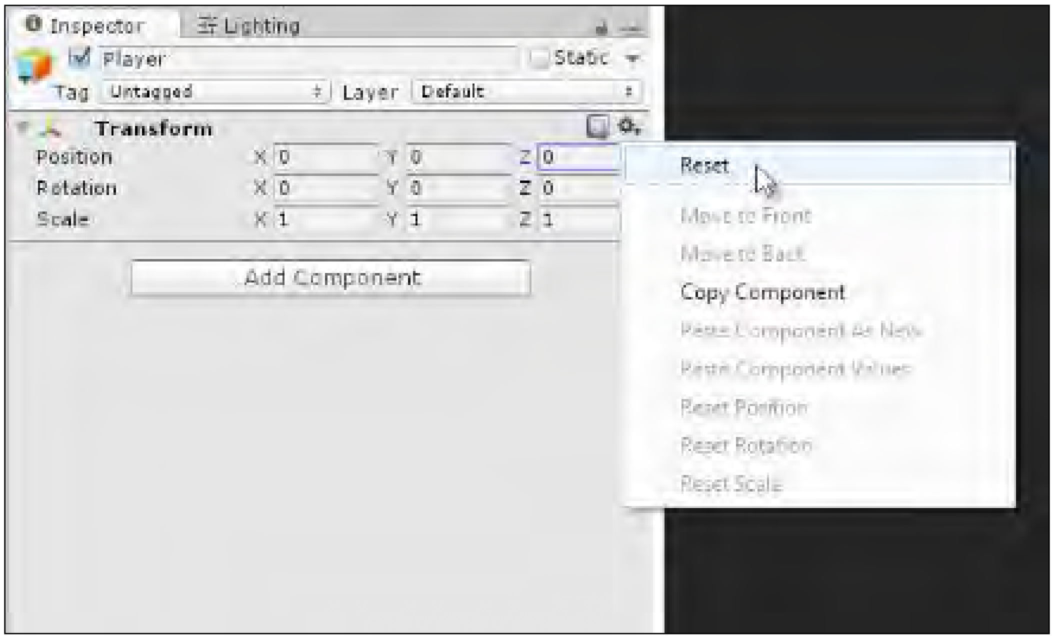
Creating a Player Object

We've now imported most assets for the twin stick shooter, and we're ready to create a player space ship object. That is, the object which the player will control and move around. Creating this might seem a trivial matter of simply dragging and dropping the relevant player sprite from the Project Panel into the Scene, but things are not so simple. The player is a complex object with many different behaviors, as we'll see. For this reason, much more care needs to be taken about creating the player. To get started, create an Empty Game Object in the scene by selecting *Game Object* > *Create Empty* from the application menu, and name the object *Player*. See Figure 3.9.



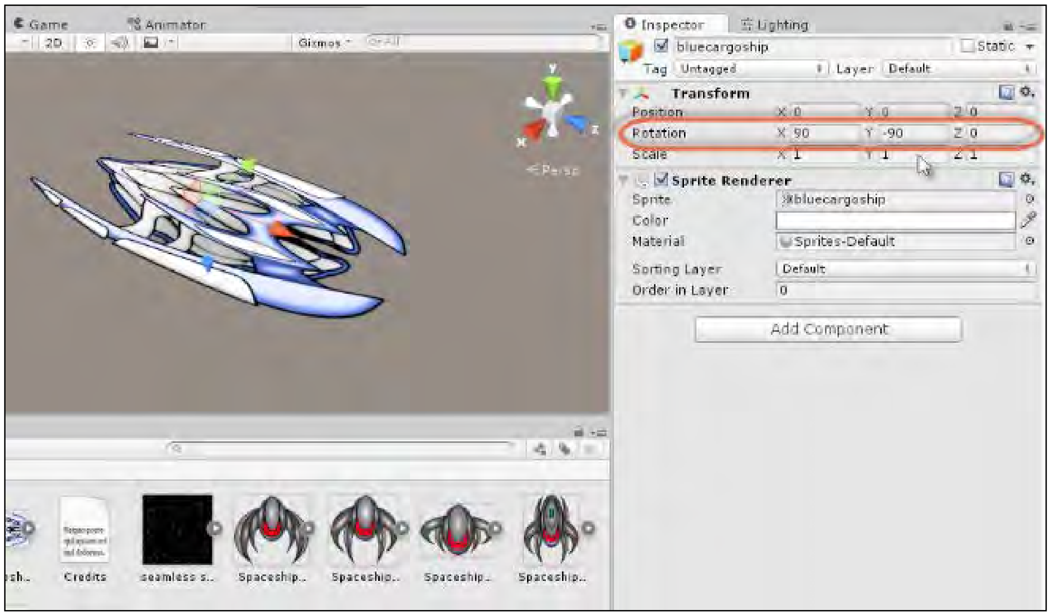
Starting to create the player

The newly created object may or may not be centered at the world origin of (0, 0, 0) and its rotation properties may not be consistently 0 across X, Y and Z. To ensure a completely zeroed transform you *could* manually set the values to 0, by entering them directly into the Transform component for the object inside the Object Inspector. However, you can set them all to 0 automatically, by clicking the cog icon, at the top-left corner of the Transform component, and selecting Reset from the context menu. See Figure 3.10.



Resetting the Transform Component...

Next, drag and drop the Player drop ship sprite (inside the Textures folder) from the Project Panel into the Hierarchy Panel, making it a child of the empty player object. Then rotate the drop ship sprite by 90 degrees in X, and -90 degrees in Y. This makes the sprite oriented in the direction of its parent's forward vector, and also flattened onto the ground plane. The game camera will take a top-down view. See Figure 3.11.



bluecargoship
Tag Untagged Layer Default

Transform

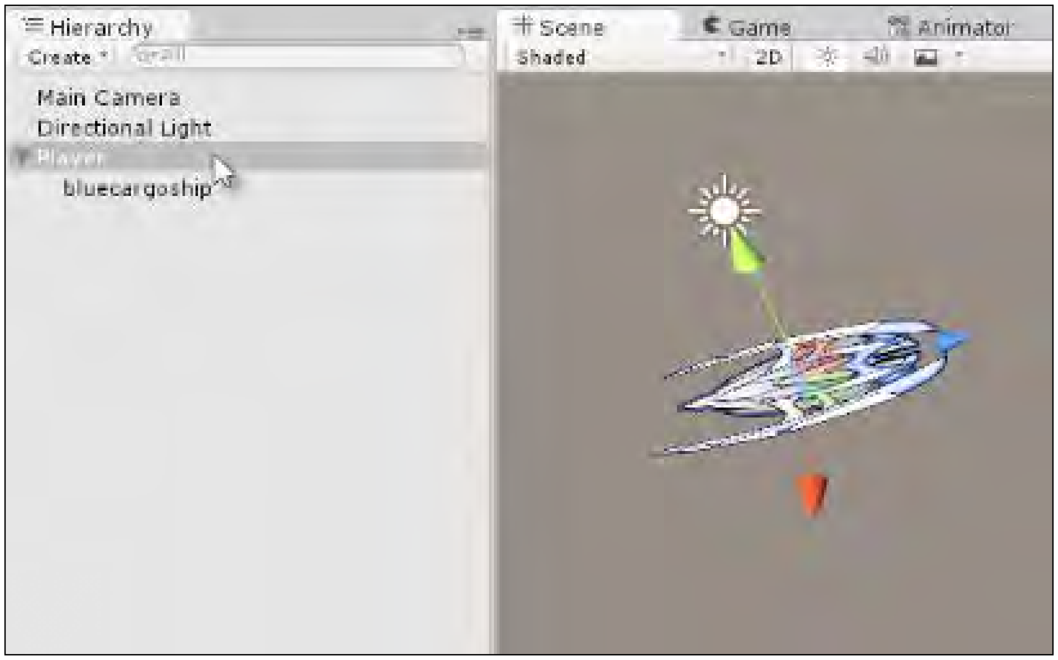
Position	X 0	Y 0	Z 0
Rotation	X 90	Y -90	Z 0
Scale	X 1	Y 1	Z 1

Sprite Renderer

Sprite	bluecargoship
Color	
Material	Sprites-Default
Sorting Layer	Default
Order in Layer	0

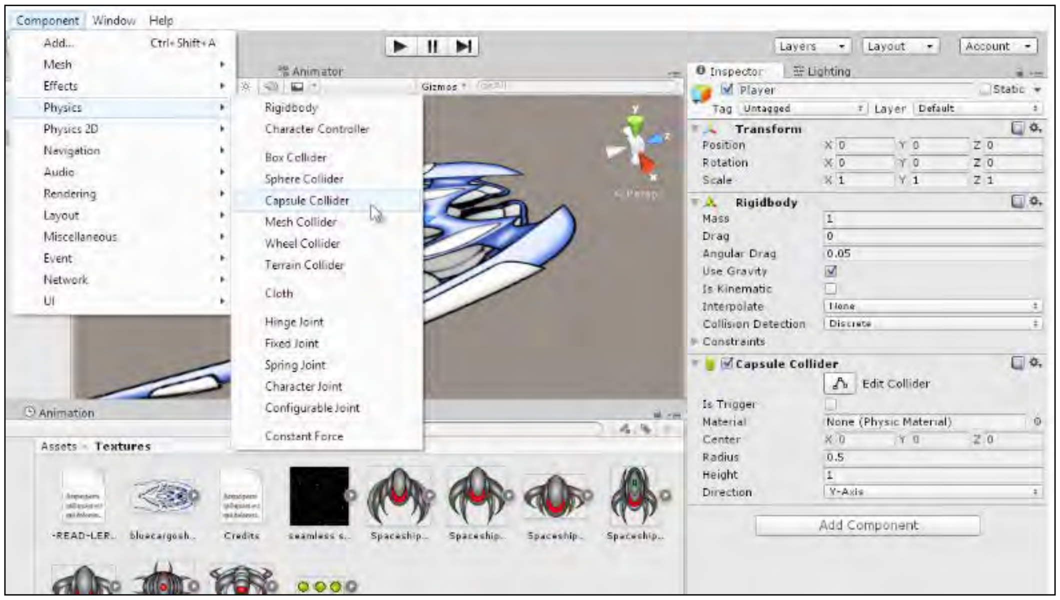
Aligning the Player Ship

You can confirm the ship sprite has been aligned correctly in relation to its parent, by selecting the Player object and viewing the blue forward vector arrow. The front of the ship sprite and the blue forward vector should be pointing in the same direction. If they're not, then continue to rotate the sprite by 90 degrees until they're in alignment. This will be important later when coding player movement, for making the ship travel in the direction it's looking. See Figure 3.12.



The blue arrow is called the 'Forward Vector'

Next, the player object should react to physics- that is, the player object is solid and effected by physical forces. It must collide with other solids, and also take damage from enemy ammo when hit. To facilitate this, two additional components should be added to the player object; specifically a *RigidBody* and a *Collider*. To do this, select the Player Object (not the sprite object), and choose *Component* > *Physics* > *Rigidbody* from the application menu. And then choose *Component* > *Physics* > *Capsule Collider* from the menu. This adds both a Rigidbody and a Collider. See Figure 3.13.



Player Static
Tag Untagged Layer Default

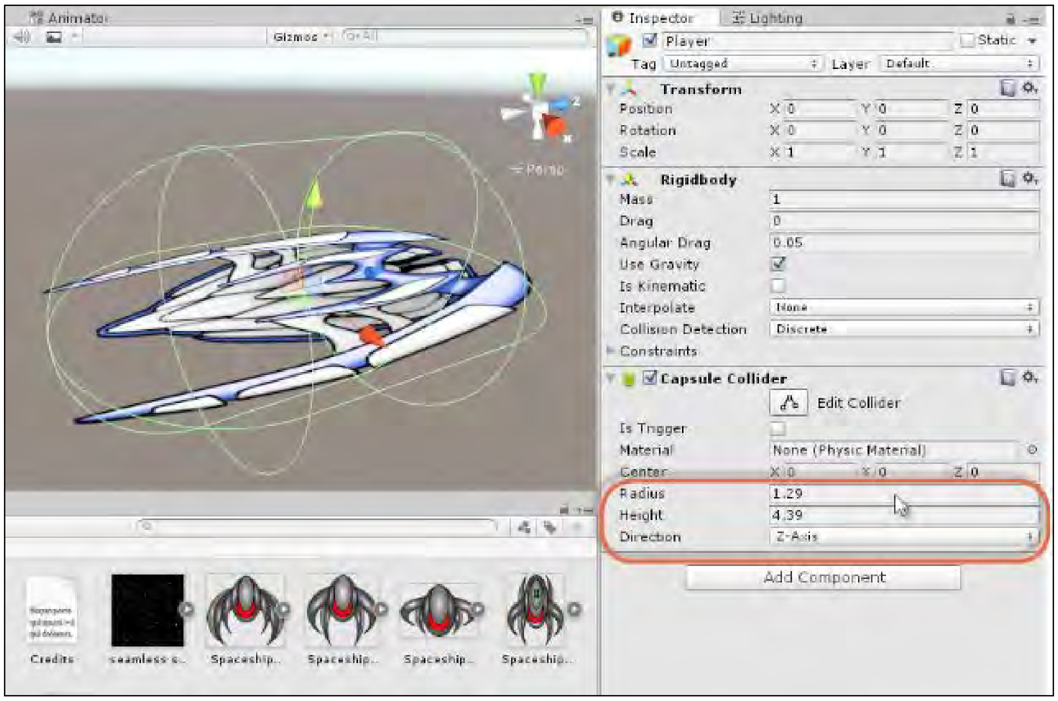
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Rigidbody
Mass 1
Drag 0
Angular Drag 0.05
Use Gravity ✓
Is Kinematic
Interpolate None
Collision Detection Discrete
Constraints

Capsule Collider
Edit Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 0.5
Height 1
Direction Y-Axis

Adding a Rigidbody and Capsule Collider to the Player Object

The Collider Component is used to approximate the volume of the object, and the Rigibody Component uses the Collider to determine how physical forces should be applied realistically. Let's adjust the Capsule Collider a little, because the default settings typically do not match up with the Player Sprite as intended. Specifically, adjust the *Direction*, *Radius* and *Height* values until Capsule encompasses the Player Sprite and represents the volume of the player. See Figure 3.14.



Player ✓ Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

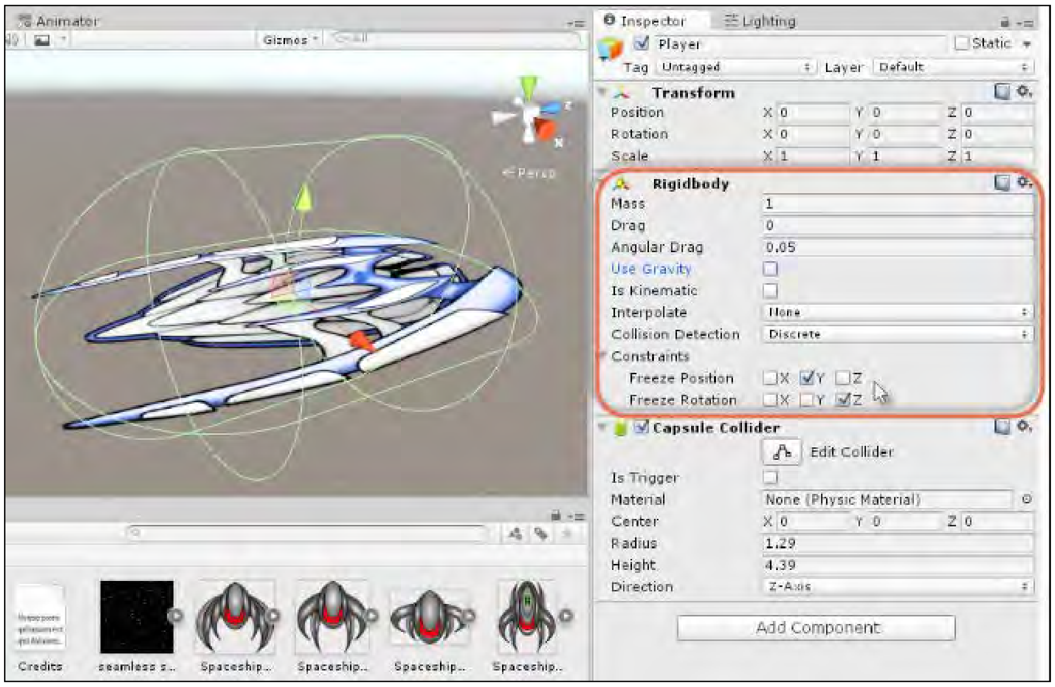
Rigidbody
Mass 1
Drag 0
Angular Drag 0.05
Use Gravity ✓
Is Kinematic
Interpolate None
Collision Detection Discrete
Constraints

✓ Capsule Collider
Edit Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 1.29
Height 4.39
Direction Z-Axis

Add Component

Adjusting the Space Ship Capsule Collider

By default, the Rigidbody component is configured to approximate objects that are affected by gravity and which fall to the ground, bumping into and reacting to other solids in the scene. This is not appropriate for a space ship that flies around. Consequently, the Rigidbody should be adjusted. Specifically: remove the *Use Gravity* check mark, to prevent the object from falling to the ground. And also enable the *Freeze Position Y* check box, and the *Freeze Rotation Z* check box, to prevent the space ship moving and rotating around axes that are undesirable in a 2D top-down game. See Figure 3.15.



Player ☐ Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Rigidbody
Mass 1
Drag 0
Angular Drag 0.05
Use Gravity ☐
Is Kinematic ☐
Interpolate None
Collision Detection Discrete
Constraints
Freeze Position ☐X ✓Y ☐Z
Freeze Rotation ☐X ☐Y ✓Z

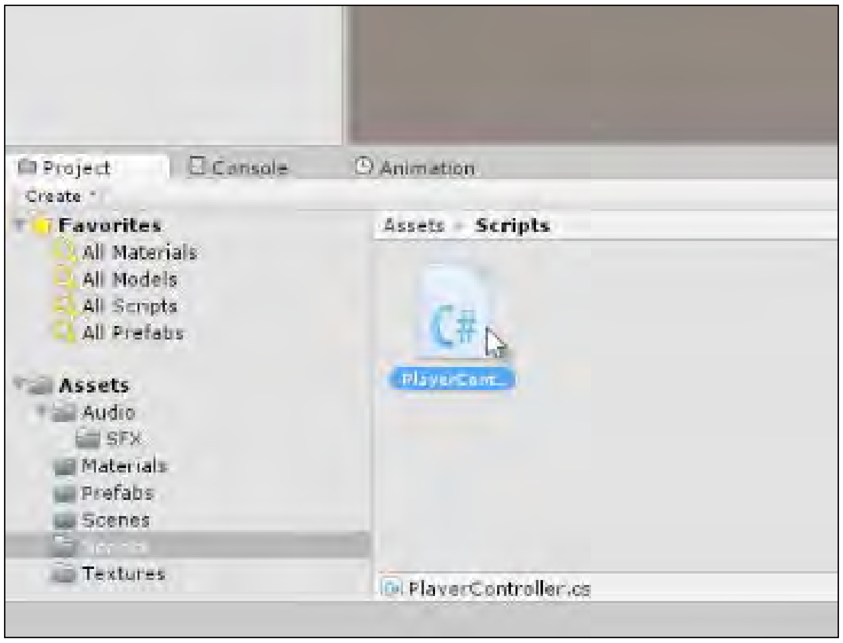
✓ Capsule Collider
Is Trigger ☐
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 1.29
Height 4.39
Direction Z-Axis

Configuring the Rigidbody Component for the Player Space Ship

Excellent work! We've now successfully configured the Player Space Ship object. Of course, it still doesn't move or do anything specific in-game. That's simply because we haven't added any code yet. That's something we'll turn to next; making the player object respond to user input.

Player Input

The Player Object is now created in the scene, configured both with a Rigidbody and Collider component. However, this object doesn't respond to player controls. In a twin stick shooter, the player provides input on two axes, and can typically shoot a weapon. This often means that keyboard WASD buttons guide player movement up, down, left and right. In addition, mouse movement controls the direction in which the player is looking and aiming; and the left mouse button typically fires a weapon. This is the control scheme required for our game. To implement this, we'll need to create a *PlayerController* script file. Right-click inside the *Scripts* folder of the Project Panel, and create a new C# script file named *PlayerController.cs*. See Figure 3.16.



Creating a Player Controller C# Script File

Inside the *PlayerController.cs* script file the following code (as shown in Code Sample 3.1) should be featured. Comments follow this sample.

```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class PlayerController : MonoBehaviour
{
    //-----------------------------
    private Rigidbody ThisBody = null;
    private Transform ThisTransform = null;

    public bool MouseLook = true;
    public string HorzAxis = "Horizontal";
    public string VertAxis = "Vertical";
    public string FireAxis = "Fire1";
    public float MaxSpeed = 5f;

    //-----------------------------
    // Use this for initialization
    void Awake ()
    {
          ThisBody = GetComponent<Rigidbody>();
          ThisTransform = GetComponent<Transform>();
    }
    //-----------------------------
    // Update is called once per frame
    void FixedUpdate ()
    {
          //Update movement
          float Horz = Input.GetAxis(HorzAxis);
          float Vert = Input.GetAxis(VertAxis);
          Vector3 MoveDirection = new Vector3(Horz, 0.0f, Vert);
          ThisBody.AddForce(MoveDirection.normalized * MaxSpeed);

          //Clamp speed
          ThisBody.velocity = new Vector3(Mathf.Clamp(ThisBody.
velocity.x, -MaxSpeed, MaxSpeed),
                                          Mathf.Clamp(ThisBody.
velocity.y, -MaxSpeed, MaxSpeed),
                                          Mathf.Clamp(ThisBody.
velocity.z, -MaxSpeed, MaxSpeed));

          //Should look with mouse?
          if(MouseLook)
          {
                //Update rotation - turn to face mouse pointer
                Vector3 MousePosWorld = Camera.main.
ScreenToWorldPoint(new Vector3(Input.mousePosition.x, Input.
mousePosition.y, 0.0f));
                MousePosWorld = new Vector3(MousePosWorld.x, 0.0f,
MousePosWorld.z);
                //Get direction to cursor
                Vector3 LookDirection = MousePosWorld -
ThisTransform.position;
```

```
                //FixedUpdate rotation

                ThisTransform.localRotation = Quaternion.
LookRotation(LookDirection.normalized,Vector3.up);
        }

    }
}
//------------------------------
```

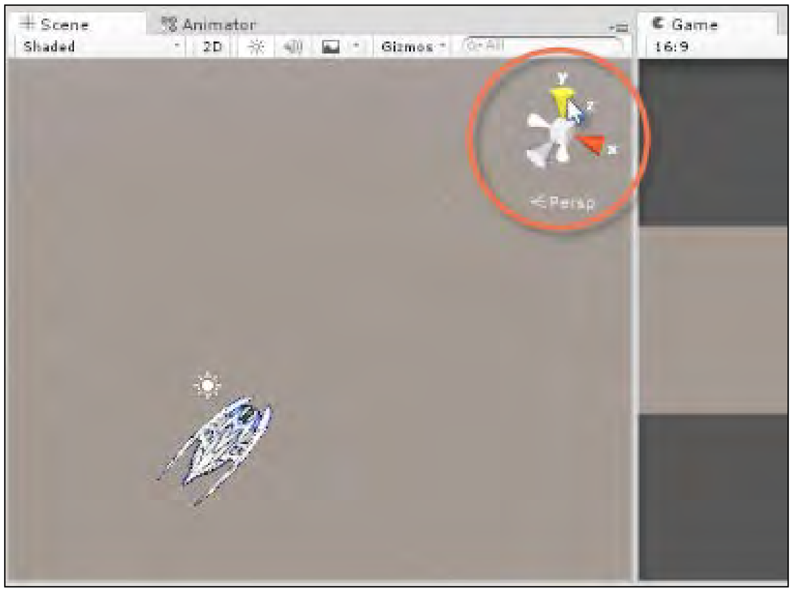
Comments on Code Sample 3-1

- The PlayerController class should be attached to the Player object in the scene. Overall it accepts input from the player, and will control movement of the space ship
- The *Awake* function is called once when the object is created at the level start. During this function, two components are retrieved, namely the Transform component for controller player rotation, and the Rigidbody Component for controller Player Movement. The *Transform* component can be used to control player movement through the *Position* property, but this ignores collisions and solid objects. The *Rigidbody* component, in contrast, prevents the player object from passing through other solids.
- The *FixedUpdate* function is called once on each update of the physics system, which is a fixed number of times per second. *FixedUpdate* differs from *Update*, which is called once per frame and can vary on a per second basis as the frame rate fluctuates. If you ever need to control an object through the physics system, by using components like Rigidbody, then you should always do so in *FixedUpdate* and not *Update*. This is a Unity convention that you should remember for best results.
- The *Input.GetAxis* function is called on each *FixedUpdate* to read axial input data from an input device, like the keyboard or gamepad. This function reads from two named axes, *Horizontal* (left-right) and *Vertical* (up-down). These work in a normalized space of -1 to 1. This means that, when the left key is pressed and held down, the Horizontal axis returns -1, and when the right key is being pressed and held down, the Horizontal axis returns 1. A value of 0 indicates either that no relevant key is being pressed, or both left and right are being pressed together, cancelling each other. A Similar principle applies for the vertical axis. Up refers to 1, down to -1, and no key-press relates to 0. More information on the *GetAxis* function can be found online at the Unity documentation here: `http://docs.unity3d.com/ScriptReference/Input.GetAxis.html`

- The *Rigidbody.AddForce* function is used to apply a physical force to the player object, moving it in a specific direction. *AddForce* encodes a velocity: moving the object in a specific direction by a specific strength. The direction is encoded inside the *MoveDirection* vector, which is based on player input from both the *Horizontal* and *Vertical* axes. This direction is multiplied by our maximum speed to ensure the object travels as fast as needed. For more information on *AddForce*, see the online Unity documentation here: `http://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html`
- The *Camera.ScreenToWorldPoint* function is used to convert the screen position of the mouse cursor within the game window into a position inside the game world, giving the player a target destination to look at. This code is responsible for making the player always look at the mouse cursor. However, as we'll see soon, some further tweaking is required to make this code work properly. For more information on *ScreenToWorldPoint*, see the Unity online documentation here: `http://docs.unity3d.com/ScriptReference/Camera.ScreenToWorldPoint.html`

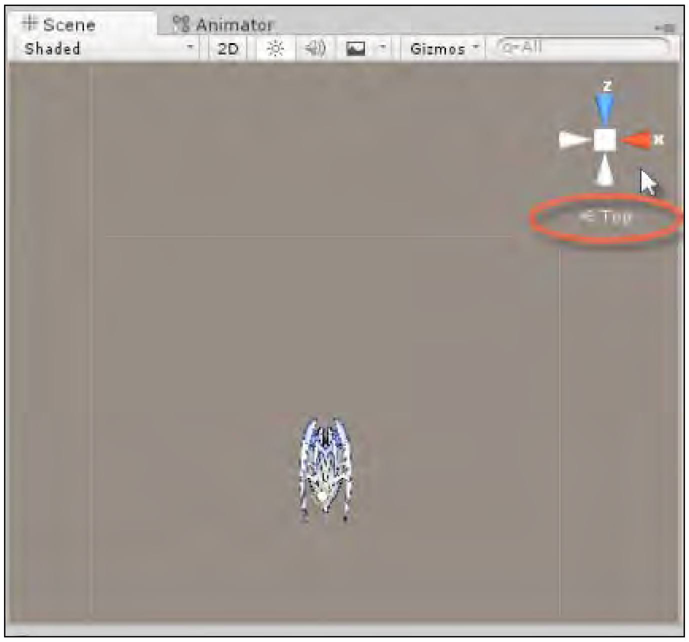
Configuring the Game Camera

Code Sample 3.1, as given in the preceding section, allows you to control the player object, but there are some problems. One of them is that the player doesn't seem to face the position of the mouse cursor, even though our code is designed to achieve this behavior. The reason is because the camera, by default, is not configured as it needs to be for a top-down 2D game. We'll fix that in this section. To get started, the scene camera should have a top-down view of the scene. To achieve this, switch the scene viewport to a top-down 2D view by clicking the ViewCube; the up arrow in the top-right hand corner of the scene viewport. This switches your viewport to a top view. See Figure 3.17.



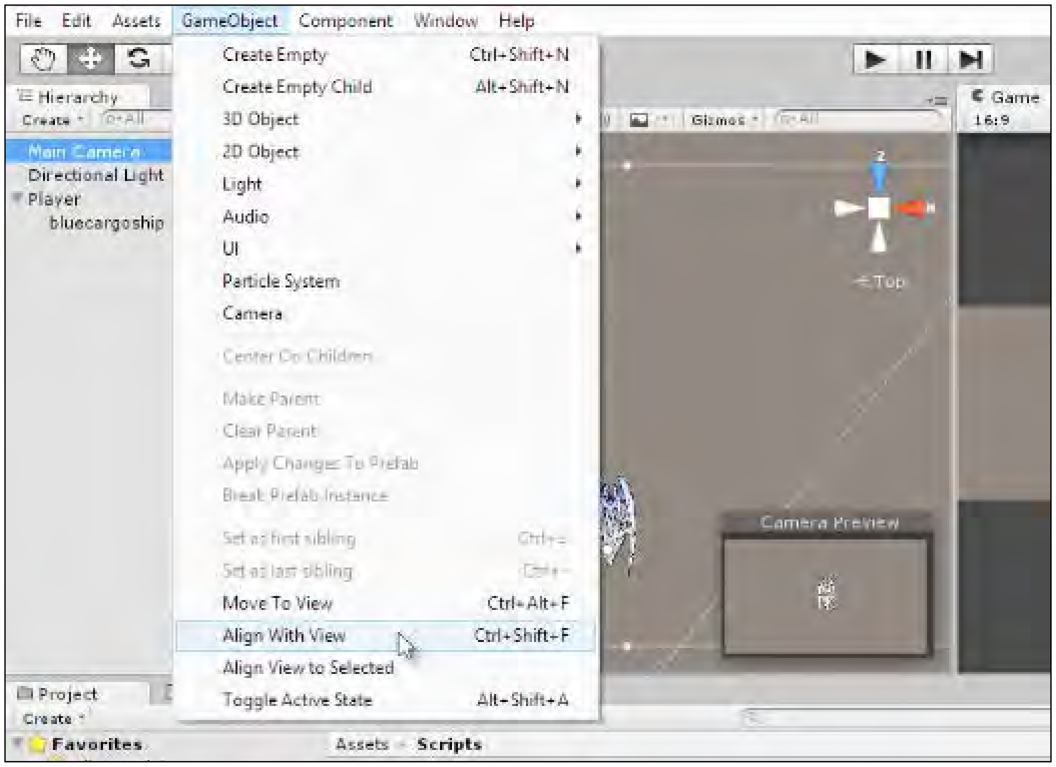
The viewcube can changes viewport perspective...

You can confirm the viewport is in a top view because the viewcube will list *Top*, as the current view. See Figure 3.18.



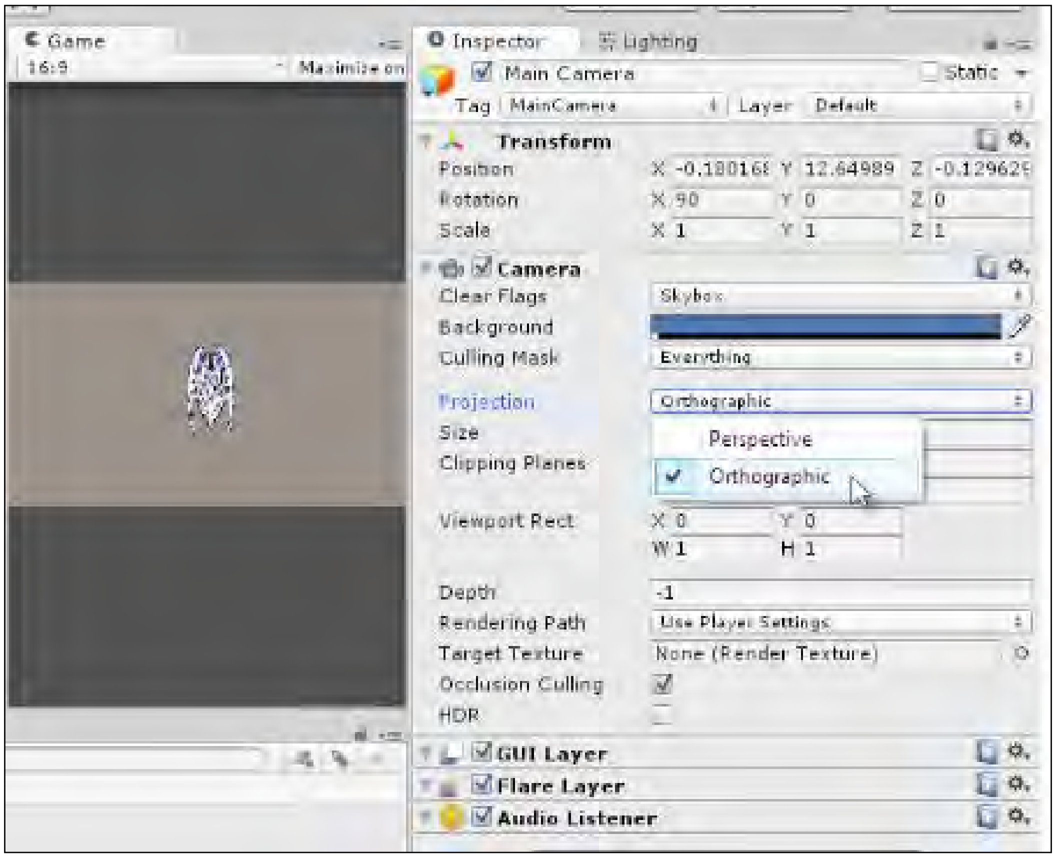
Top View in the Scene Viewport

From here, you can have the scene camera conform to the viewport camera exactly, giving you an instant top-down view for your game. To do this, select the Camera in the scene (or from the Hierarchy Panel), and then choose *GameObject* > *Align with View* from the application menu. See Figure 3.19.



Aligning the camera to the scene viewport

This makes your game look much better than before, but there's still a problem. When the game is running, the space ship still doesn't look at the mouse cursor as intended. This is because the camera is a *Perspective* camera, and the conversion between a screen point and world point is leading to unexpected results. We can fix this by changing the camera into an Orthographic camera, which is a truly 2D camera that allows no perspective distortion. To do this, select the Camera in the scene, and from the Object Inspector change the Projection setting from *Perspective* to *Orthographic*.



Main Camera
Tag MainCamera
Layer Default

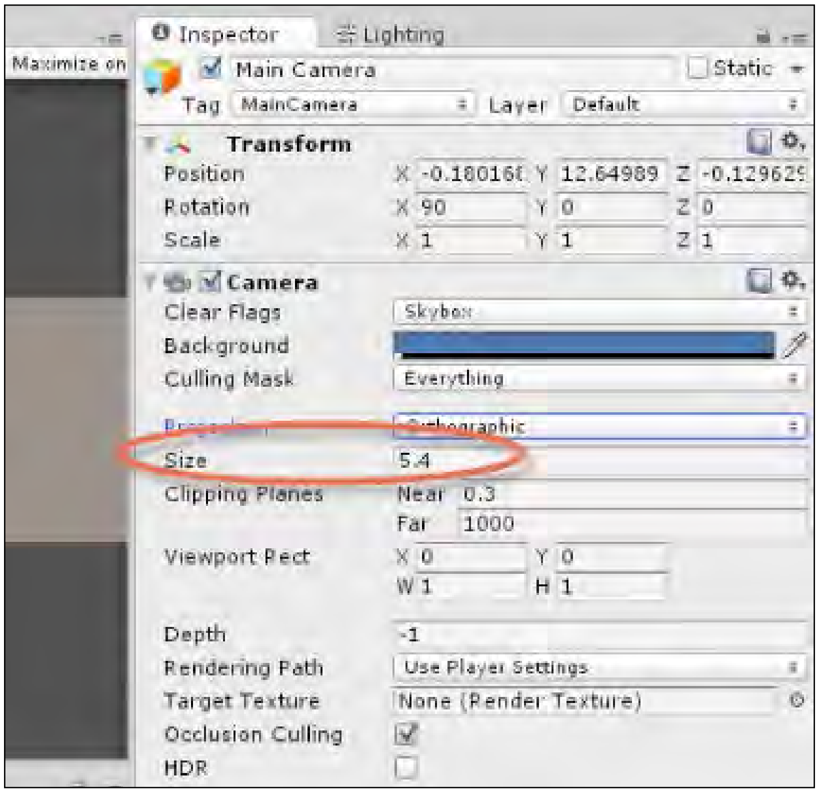
Transform
Position X -0.180168 Y 12.64989 Z -0.129629
Rotation X 90 Y 0 Z 0
Scale X 1 Y 1 Z 1

Camera
Clear Flags Skybox
Background
Culling Mask Everything
Projection Orthographic
Size
Clipping Planes
Perspective
✓ Orthographic
Viewport Rect X 0 Y 0
W 1 H 1
Depth -1
Rendering Path Use Player Settings
Target Texture None (Render Texture)
Occlusion Culling ✓
HDR

GUI Layer
Flare Layer
Audio Listener

Changing the Camera to Orthographic Mode

Every Orthographic camera has a *Size* field in the Object Inspector, which is not present for Perspective Cameras. This field controls how many units in the world view corresponds to pixels on the screen. We want a 1:1 ratio or relationship between world units to pixels, to ensure our textures appear at the correct size and that cursor movement has the intended effect. The target resolution for our game will be Full HD, which is 1920x1080, and this has an Aspect Ratio of 16:9. For this resolution, the Orthographic Size should be *5.4*. The reasons for that value are beyond the scope of this book, but the formula to arrive at it is *Screen Height (in pixels) / 2 / 100*. Therefore: *1080 / 2 / 100* = *5.4*. See Figure 3.21.



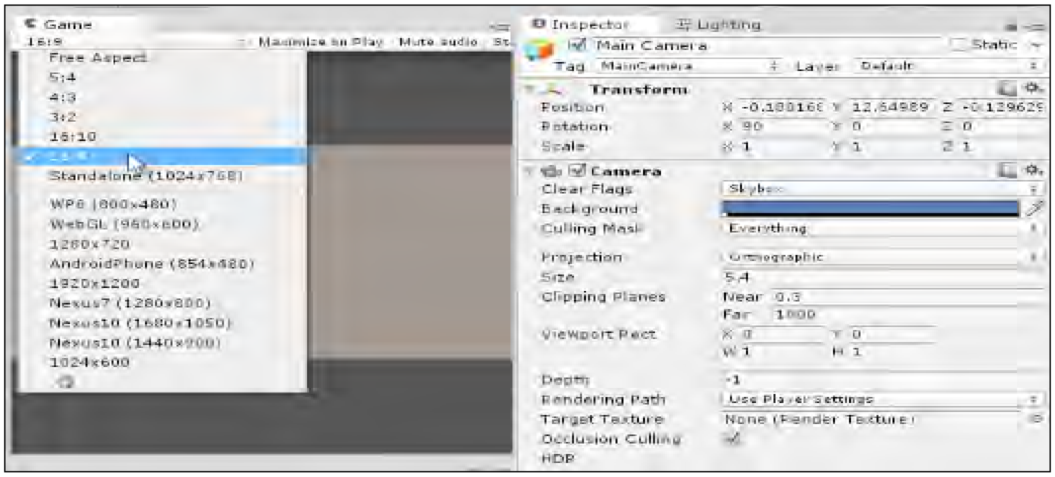
Main Camera Static
Tag MainCamera Layer Default

Transform
Position X -0.18016 Y 12.64989 Z -0.12962
Rotation X 90 Y 0 Z 0
Scale X 1 Y 1 Z 1

Camera
Clear Flags Skybox
Background
Culling Mask Everything
Projection Orthographic
Size 5.4
Clipping Planes Near 0.3
Far 1000
Viewport Rect X 0 Y 0
W 1 H 1
Depth -1
Rendering Path Use Player Settings
Target Texture None (Render Texture)
Occlusion Culling ✓
HDR ☐

Changing Orthographic Size for a 1:1 Pixel-To-Screen Ratio

Finally, make sure your Game tab view is configured to display the game at 16:9 aspect ratio. If it isn't, click on the Aspect drop-down list at the top-left corner of the Game view, and choose the 16:9 option. See Figure 3.22.



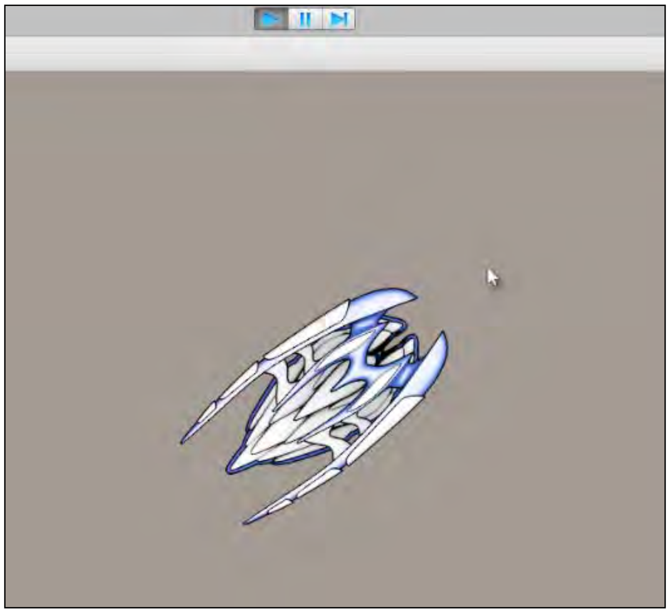
Main Camera Static
Tag MainCamera Layer Default

Transform
Position X -0.180168 Y 12.64989 Z -0.129629
Rotation X 90 Y 0 Z 0
Scale X 1 Y 1 Z 1

Camera
Clear Flags Skybox
Background
Culling Mask Everything
Projection Orthographic
Size 5.4
Clipping Planes Near 0.3
Far 1000
Viewport Rect X 0 Y 0
W 1 H 1
Depth -1
Rendering Path Use Player Settings
Target Texture None (Render Texture)
Occlusion Culling ✓
HDR

Displaying the game at a 16:9 Aspect Ratio

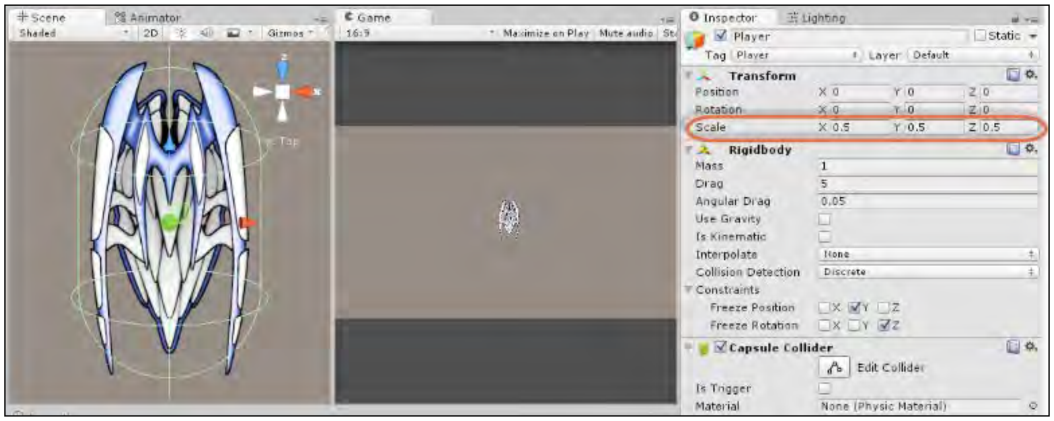
Now try running the game, and you have a player space ship that moves based on WASD input, and also turns to face the mouse cursor. Great work! See Figure 3.23. The game is really taking shape. But, there's lots more work to do.



Turning to face the cursor!

Bounds Locking

On previewing the game thus far the space ship probably looks too large. We can fix this easily by just changing the scale of the player object. I've used a value of 0.5 for the X, Y and Z axes. See Figure 3.24. But, even with a more sensible scale, a problem remains. Specifically, it's possible to move the player outside the boundaries of the screen, without limit. This means the player can fly off into the distance, out of view, and never be seen again. Instead, the camera should remain still and the player movement should be limited to the camera view or bounds so it never exits view.



Player [✓] Static [ ]
Tag Player Layer Default
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 0.5 Y 0.5 Z 0.5
Mass 1
Drag 5
Angular Drag 0.05
Use Gravity [ ]
Is Kinematic [ ]
Interpolate None
Collision Detection Discrete
Constraints
Freeze Position [ ] X [✓] Y [ ] Z
Freeze Rotation [ ] X [ ] Y [✓] Z
Is Trigger [ ]
Material None (Physic Material)

Rescaling the Player

There are different ways to achieve bounds locking, most of which involve scripting. One way is to simply clamp the positional values of the player object between a specified range; a minimum and maximum. Consider code sample 3.2 for a new C# class called *BoundsLock*. This script file should be attached to the player.

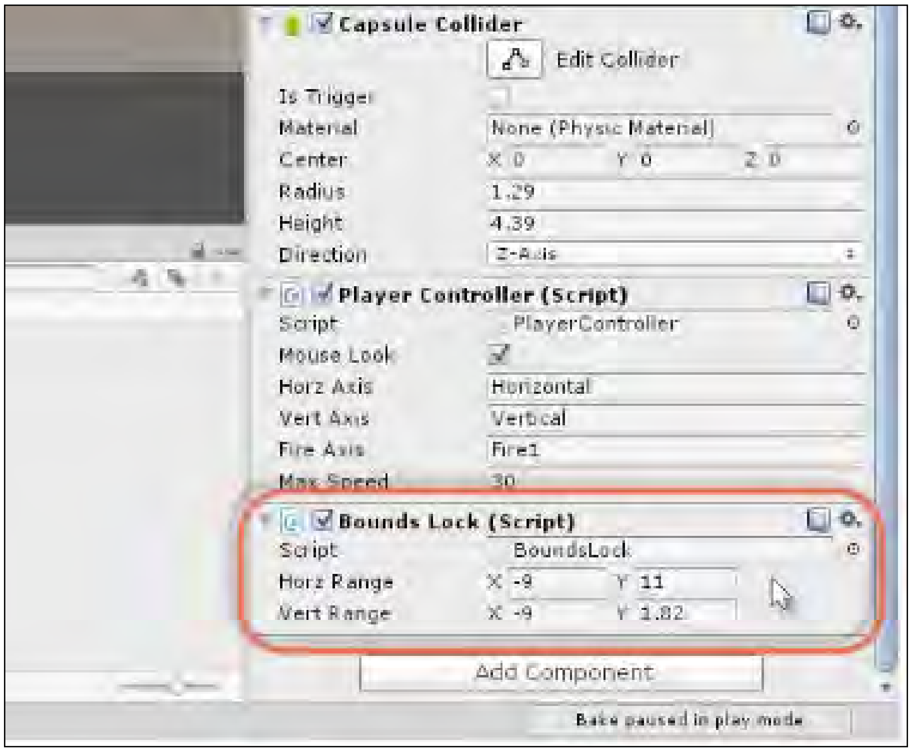
```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class BoundsLock : MonoBehaviour
{
   //------------------------------
   private Transform ThisTransform = null;
   public Vector2 HorzRange = Vector2.zero;
   public Vector2 VertRange = Vector2.zero;
   //------------------------------
```

```
    // Use this for initialization
    void Awake ()
    {
        ThisTransform = GetComponent<Transform>();
    }
    //------------------------------
    // Update is called once per frame
    void LateUpdate ()
    {
        //Clamp position
        ThisTransform.position = new Vector3(Mathf.
Clamp(ThisTransform.position.x, HorzRange.x, HorzRange.y),
                                             ThisTransform.
position.y,
                                             Mathf.
Clamp(ThisTransform.position.z, VertRange.x, VertRange.y));
    }
    //------------------------------
}
//------------------------------
```

Comments on Code Sample 3-2

- The *LateUpdate* function is always called after all *FixedUpdate* and *Update* calls, allowing an object to modify its position before it's rendered to the screen.
- The *Mathf.Clamp* function ensures a specified value is capped between a minimum and maximum range.

To use the BoundsLock script, simply drag and drop the file onto the Player Object and specify minimum and maximum values for its position. These values are specified in world position coordinates, and can be determined by temporarily moving the player object to the camera extremes and recording its position from the Transform component.



Is Trigger
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 1.29
Height 4.39
Direction Z-Axis

Player Controller (Script)
Script PlayerController
Mouse Look ✓
Horz Axis Horizontal
Vert Axis Vertical
Fire Axis Fire1
Max Speed 30

Bounds Lock (Script)
Script BoundsLock
Horz Range X -9 Y 11
Vert Range X -9 Y 1.82

Setting Bounds Lock

Now take the game for a test run by pressing Play on the tool bar. The player space ship should remain in view and be unable to move off screen. Splendid!

Health

Both the player space ship and the enemies need health. Health is a measure of a character's presence and legitimacy in the scene; typically scored as a value between 0-100. 0 means death, and 100 means full health. Now, although health is in many respects specific to each instance (the player has a unique health bar, and each enemy has theirs) there are nevertheless so many things in common, in terms of behavior, between player and enemy health that it makes sense to code health as a separate component and class that can be attached to all objects that need health. Consider Code Sample 3.3, which should be attached to the player, and all enemies or objects that need health. Comments follow.

```
using UnityEngine;
using System.Collections;
//------------------------------
public class Health : MonoBehaviour
{
   public GameObject DeathParticlesPrefab = null;
   private Transform ThisTransform = null;
   public bool ShouldDestroyOnDeath = true;
   //------------------------------
   void Start()
   {
         ThisTransform = GetComponent<Transform>();
   }
   //------------------------------
   public float HealthPoints
   {
         get
         {
               return _HealthPoints;
         }

         set
         {
               _HealthPoints = value;

               if(_HealthPoints <= 0)
               {
                     SendMessage("Die", SendMessageOptions.
DontRequireReceiver);

                     if(DeathParticlesPrefab != null)
```

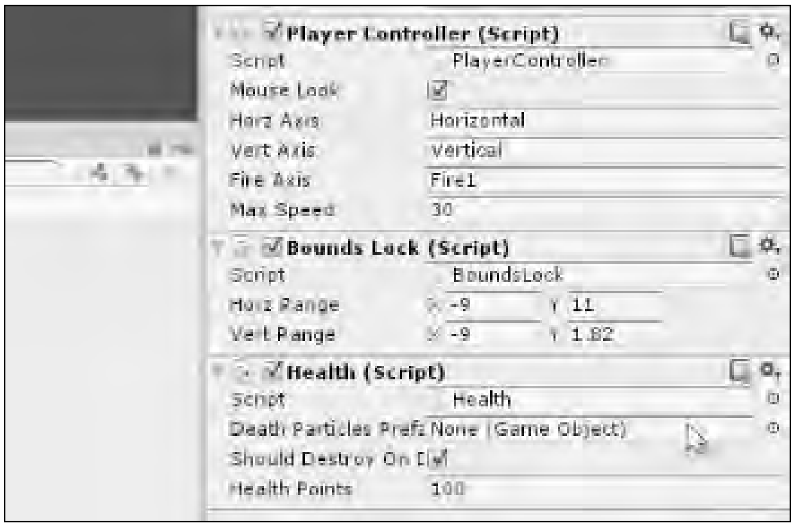
```
                              Instantiate(DeathParticlesPrefab,
ThisTransform.position, ThisTransform.rotation);

                          if(ShouldDestroyOnDeath)Destroy(gameObject);
                    }
              }
        }
        //----------------------------
        [SerializeField]
        private float _HealthPoints = 100f;
    }
    //----------------------------
```

Comments on Code Sample 3-3

- The health class maintains object health through a private variable `_HealthPoints`, which is accessed through a **C# Property** `HealthPoints`. This property features both Get and Set accessor, to return and set the health variable.
- The `_HealthPoints` variable is declared as a `SerializedField`, allowing its value to be visible in the Object Inspector.
- The Health class is an example of **Event Driven Programming**. This is because the class *could* have continually checked the status of object health during an `Update` function; checking to see if the object had died by its health falling below 0. But instead, the check for death is made during the C# Property `Set` method. This makes sense because `Set` is the only place where health will ever change. This means Unity is saved from a lot of work each frame. That's a great performance saving!
- This class uses the `SendMessage` function. This function lets you call any other *public function* on *any component* attached to the object by specifying the function name as a string. In this case, a function called *Die* will be executed on *every* component attached to the object (if such a function exists). If no function of matching name exists, then nothing happens for that component. This is a quick and easy way to run customized behavior on an object in a type-agnostic way without using any polymorphism. The disadvantage is that *SendMessage* internally uses a process called *Reflection*, which is slow and performance prohibitive. For this reason, *SendMessage* should be used only infrequently for death events and similar events, but not frequently, such as every frame. More information on *SendMessage* can be found at the online Unity documentation here: `http://docs.unity3d.com/ScriptReference/GameObject.SendMessage.html`

- When health falls below 0, triggering a death condition, the code will instantiate a death particle system, to show an effect on death, if a particle system is specified (more on this shortly).

When the Health script is attached to the player space ship it appears as a component in the Object Inspector. It contains a field for a Death Particle System. This is an optional field (it can be null), specifying a particle system to be spawned when the object dies. This lets you easily create explosion or blood splatter effects when objects die. See Figure 3.26.

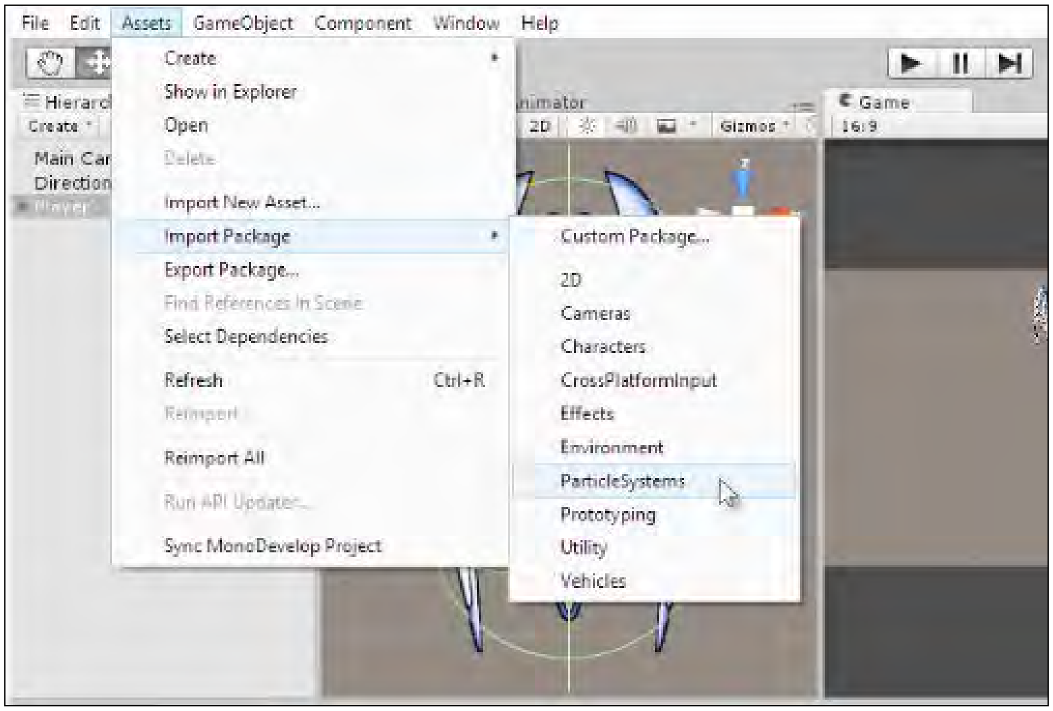

Player Controller (Script)
Script PlayerController
Mouse Look ✓
Horz Axis Horizontal
Vert Axis Vertical
Fire Axis Fire1
Max Speed 30
Bounds Lock (Script)
Script BoundsLock
Horz Range X -9 Y 11
Vert Range X -9 Y 1.82
Health (Script)
Script Health
Death Particles Prefa None (Game Object)
Should Destroy On D ✓
Health Points 100

Attaching the Health Script

Death and Particles

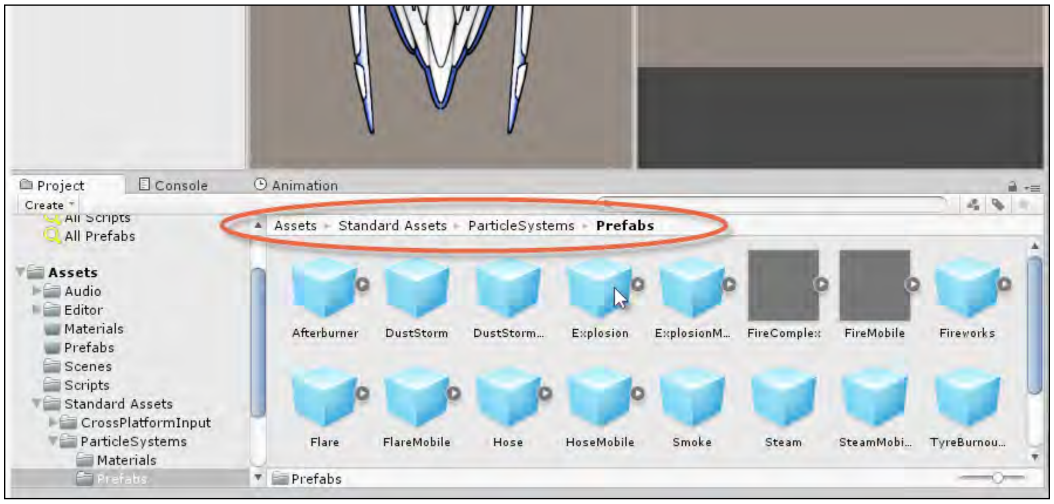
In this twin stick shooter game both the player and enemies are space ships. When they're destroyed they should explode in a fiery ball. This is really the only kind of effect that would be believable. To achieve explosions we can use a Particle System. This simply refers to a special kind of object that features two main parts, namely a Hose (or **Emitter**) and **Particles**. The Emitter refers to the part which spawns or generates new Particles into the world, and the Particles are many small objects or pieces that, once spawned, move and travel along their own trajectories. In short, Particle Systems are ideal for creating rain, snow, fog, sparkles, and explosions.

We can create our own Particle Systems from scratch, using the menu option *GameObject* > *Particle System*, or we can use any pre-made particle system included with Unity. Let's use some of the pre-made particle systems. To do this, import the Particle System package into the Project, by selecting *Assets* > *Import Package* > *Particle Systems* from the application menu. See Figure 3.27.



Importing Particle Systems into the Project

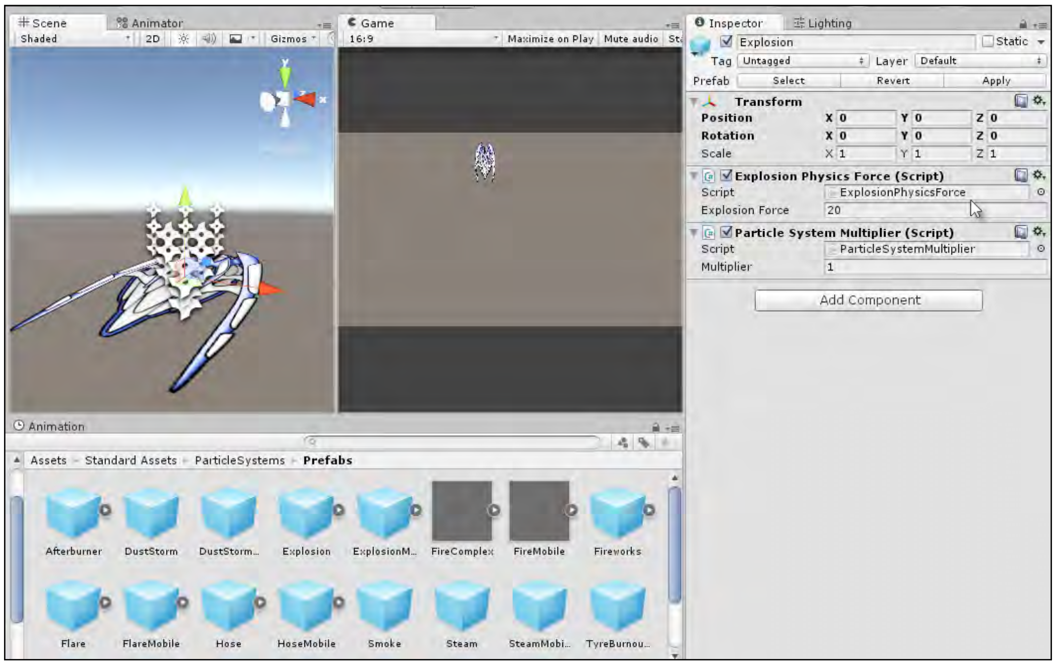
After the Import Dialog appears, leave all settings at their defaults, and simply click *Import* to import the complete package, including all particle systems. The Particle Systems will be added to the Project Panel, in the folder *Standard Assets* > *Particle Systems* > *Prefabs*. See Figure 3.28. You can test each of the Particle Systems by simply dragging and dropping each Prefab into the scene. Note, you can only preview a Particle System in the Scene viewport while it is selected.



Particles Systems imported into the Project Panel

Notice from Figure 3.28 above that an Explosion system is included among the default assets, which is great news! To test, we can just drag and drop the explosion into the scene, press *Play* on the tool bar, and see the explosion in action. Good. We're almost done, but there's still a bit more work. We've now seen that an appropriate Particle System is available, and we could just drag and drop that system into the *Death Particle System* slot in the *Health* Component, in the Object Inspector. That will work technically: when a player or enemy dies, the explosion system will be spawned, creating an explosion effect. But, the particle system will never be destroyed! This is problematic because, on each enemy death a new particle system will be spawned. And this raises the possibility that, after many deaths, the scene will be full of disused particle systems. We don't want that: it's bad for performance and memory usage to have a scene full of unused objects lingering around.

To fix this, we'll modify the explosion system slightly, creating a new and modified prefab that'll suit our needs. To create this, drag and drop the existing explosion system anywhere into the scene, and position it at the world origin. See Figure 3.29.



Explosion
Static
Tag Untagged
Layer Default
Prefab Select Revert Apply

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Explosion Physics Force (Script)
Script ExplosionPhysicsForce
Explosion Force 20

Particle System Multiplier (Script)
Script ParticleSystemMultiplier
Multiplier 1

Adding an Explosion System to the Scene for Modification

Next, we must refine the particle system to destroy itself soon after instantiation. By making a prefab from this arrangement, each and every generated explosion will eventually destroy itself. To make an object destroy itself after a specified interval, we'll create a new C# Script. I'll name this script *TimeDestroy.cs*. See the following code in Sample 3.4.

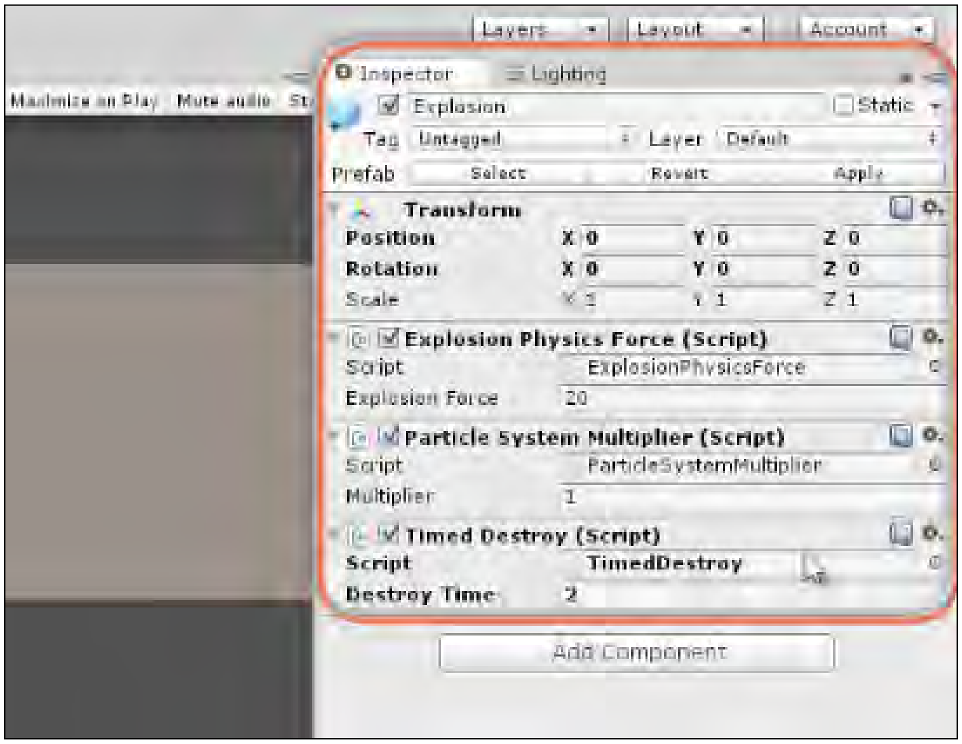
```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class TimedDestroy : MonoBehaviour
{
   public float DestroyTime = 2f;
```

```
    //-----------------------------
    // Use this for initialization
    void Start ()
    {
          Invoke("Die", DestroyTime);
    }

    // Update is called once per frame
    void Die ()
    {
          Destroy(gameObject);
    }
    //-----------------------------
}
//------------------------------
```

Comments on Code Sample 3-4

- The *TimeDestroy* class simply destroys the object to which it's attached after a specified interval (`DestroyTime`) has elapsed.
- The function `Invoke` is called inside the `Start` event. Invoke will execute a function of the specified name once and only once, after a specified interval has elapsed. The interval is measured in seconds.
- Like `SendMessage`, the Invoke function relies on *Reflection*. For this reason it should be used sparingly for best performance.
- The **`Die`** function will be executed by Invoke after a specified interval to destroy the Game Object (such as a Particle System).

Now drag and drop the *TimedDestroy* script file onto the explosion Particle System in the scene, and then press *Play* on the tool bar to test that the code works; that the object is destroyed after the specified interval, which can be adjusted from the Object Inspector. See Figure 3.30.



Explosion
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

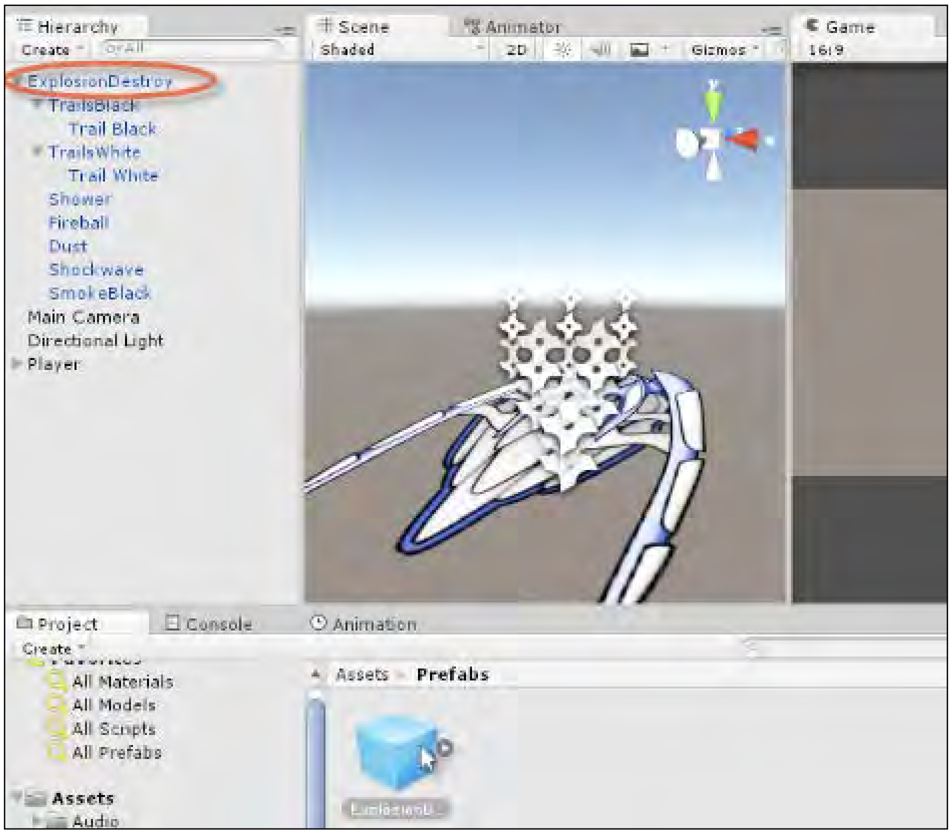
Explosion Physics Force (Script)
Script ExplosionPhysicsForce
Explosion Force 20

Particle System Multiplier (Script)
Script ParticleSystemMultiplier
Multiplier 1

Timed Destroy (Script)
Script TimedDestroy
Destroy Time 2

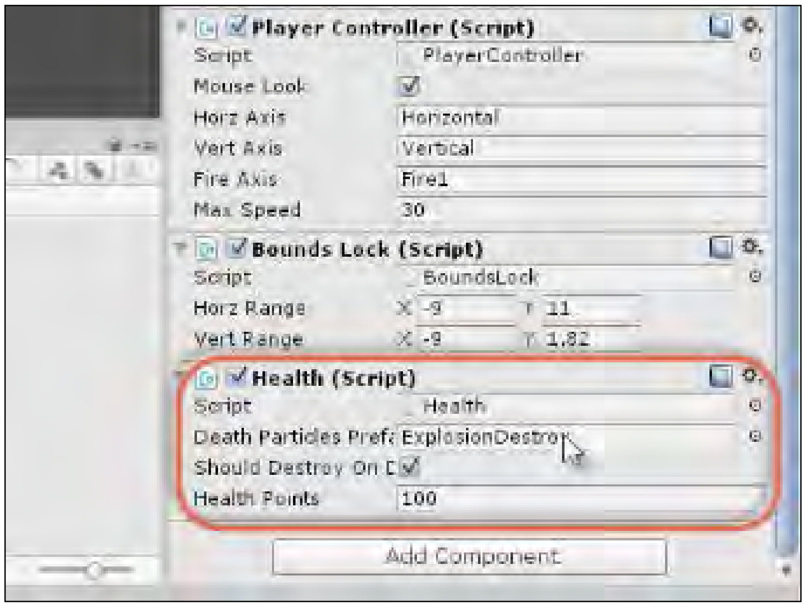
Adding a TimeDestroy script to an explosion Particle System

The *TimeDestroy* Script should remove the explosion particle system after the delay expires. So let's create a new and separate Prefab from this modified version. To do that rename the explosion system in the Hierarchy Panel *ExplosionDestroy*, and then drag and drop the system from the Hierarchy into the Project Panel, inside the *Prefabs* folder. Unity automatically creates a new Prefab, representing the modified particle system. See Figure 3.31.



Create a Timed Explosion Prefab

Now drag and drop the newly created prefab from the Project Panel into the *Death Particle System* slot on the *Health* Component for the Player, in the Object Inspector. This ensures the prefab is instantiated when the player dies. See Figure 3.32.



Player Controller (Script)
Script: PlayerController
Mouse Look: ✓
Horz Axis: Horizontal
Vert Axis: Vertical
Fire Axis: Fire1
Max Speed: 30

Bounds Lock (Script)
Script: BoundsLock
Horz Range: X -9 Y 11
Vert Range: X -9 Y 1.82

Health (Script)
Script: Health
Death Particles Prefa: ExplosionDestroy
Should Destroy On D: ✓
Health Points: 100

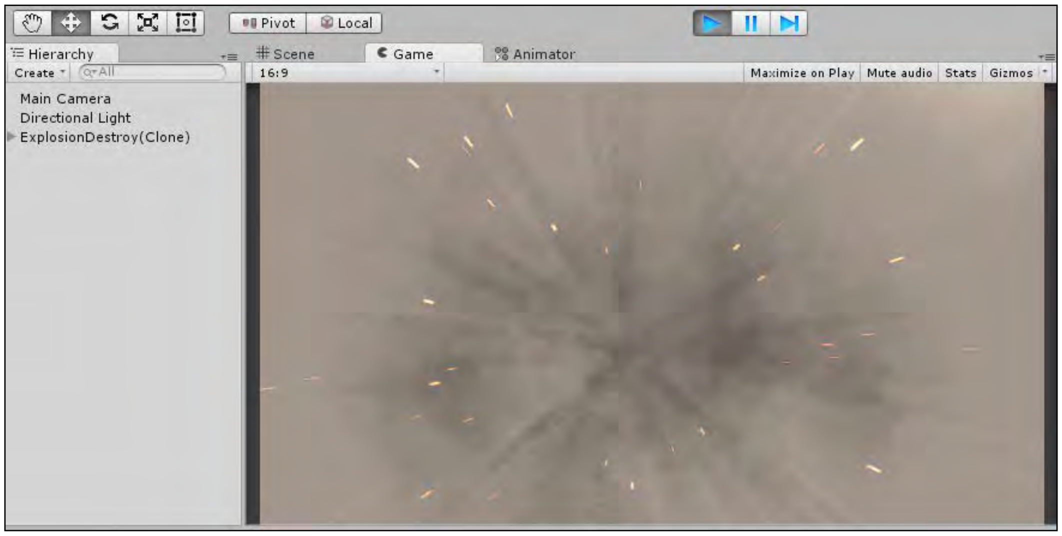
Configuring the Health Script

If you now run the game, you'll see that you cannot initiate a player death event to test the particle system generation. Nothing exists in the scene to destroy or damage the player, and you cannot manually set the Health Points to 0 from the Inspector in a way that is detected by the C# property set function. For now, however, we can insert some test death functionality into the health script that triggers an instant kill when the space bar is pressed. See Code Sample 3.5 for the modified Health Script.

```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class Health : MonoBehaviour
{
   public GameObject DeathParticlesPrefab = null;
   private Transform ThisTransform = null;
   public bool ShouldDestroyOnDeath = true;
   //------------------------------
   void Start()
   {
```

```
            ThisTransform = GetComponent<Transform>();
    }
    //------------------------------
    public float HealthPoints
    {
            get
            {
                    return _HealthPoints;
            }

            set
            {
                    _HealthPoints = value;

                    if(_HealthPoints <= 0)
                    {
                            SendMessage("Die", SendMessageOptions.
DontRequireReceiver);

                            if(DeathParticlesPrefab != null)
                                    Instantiate(DeathParticlesPrefab,
ThisTransform.position, ThisTransform.rotation);

                            if(ShouldDestroyOnDeath)Destroy(gameObject);
                    }
            }
    }
    //------------------------------
    void Update()
    {
            if(Input.GetKeyDown(KeyCode.Space))
                    HealthPoints = 0;
    }
    //------------------------------
    [SerializeField]
    private float _HealthPoints = 100f;
}
//------------------------------
```

On running the game now, with the modified health script, you can trigger an instant player death by pressing the space bar key on the keyboard. When you do this, the player object is destroyed and the particle system is generated until the timer destroys that too. Excellent work. We now have a playable, controllable player character that supports health and death functionality. Things are looking good. See Figure 3.33.



Trigger the Explosion Particle System...

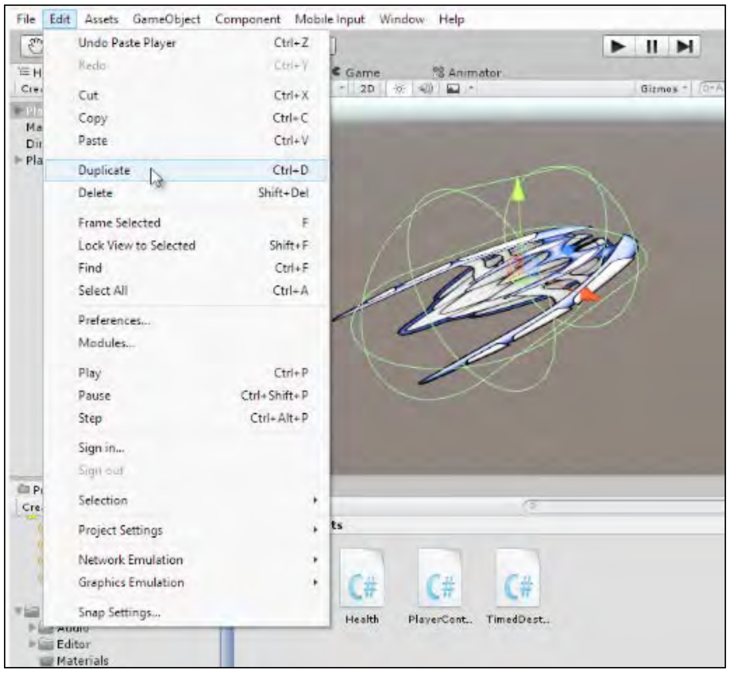
Enemies

The next step is to create something for the player to shoot and destroy, and which can also destroy us- namely enemy characters. These take the form of roaming space ships that will be spawned into the scene at regular intervals and will follow the player, drawing nearer and nearer. Essentially, each enemy represents a complex of multiple behaviors working together, and these should be implemented as separate scripts. Let's consider them in turn.

- Health
- Each enemy supports health functionality. They begin the scene with a specified amount of health and will be destroyed when that health falls below 0. We already have a health script created to handle this behavior.
- Movement

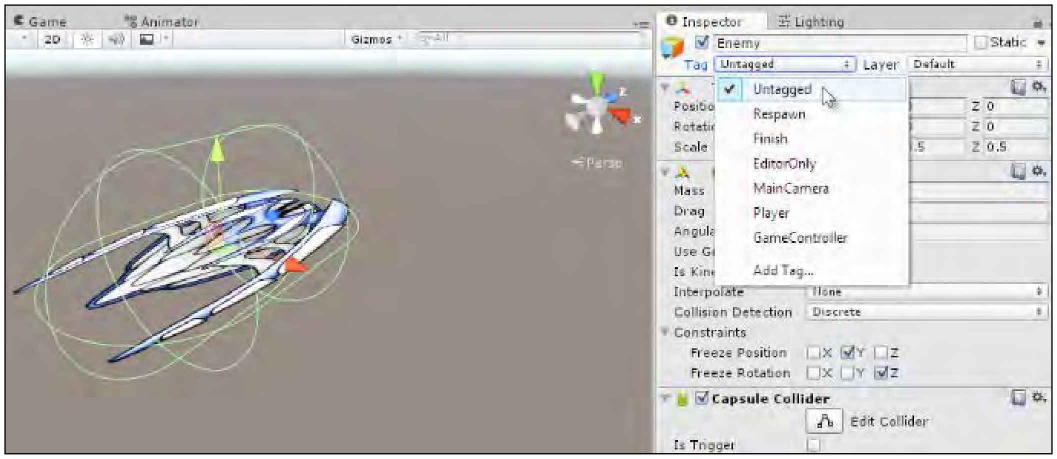
- Each enemy will constantly be in motion, travelling in a straight line along a forward trajectory. That is, each enemy will continually travel forwards in the direction it is looking.
- Turning
- Each enemy will rotate and turn towards the player, even when the player moves. This ensures the enemy always faces the player and, in combination with the movement functionality, will always be travelling towards the player.
- Scoring
- Each enemy rewards the player with a score value when destroyed. Thus, the death of an enemy will increase the player score.
- Damage
- Each enemy causes damage to the player on collision. Enemies cannot shoot, but will harm the player on proximity.

Now we've identified the range of behaviors applicable to an enemy, let's create an enemy in the scene. We'll make one specific enemy, create a prefab from that, and use it as the basis for instantiating many enemies. Start by selecting the player character in the scene and duplicate the object with *Ctrl* + *D*, or select *Edit* > *Duplicate* from the application menu. This initially creates a second player. See Figure 3.34.



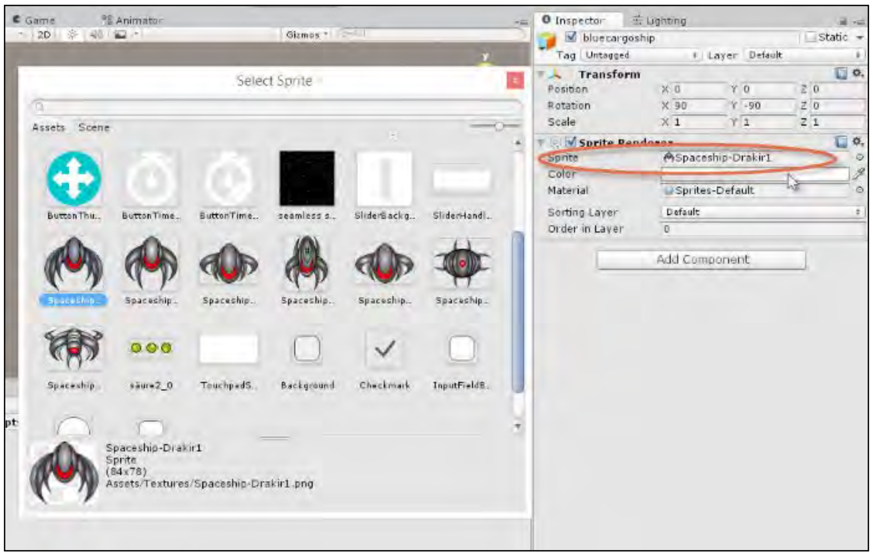
Duplicating the Player Object

Rename the object to *Enemy*, and ensure it is not tagged as *Player*; as there should be one and only one object in the scene with the Player tag- namely, the real player. In addition, temporarily disable the Player game object, allowing us to focus more clearly in the enemy object in the scene tab. See Figure 3.35.



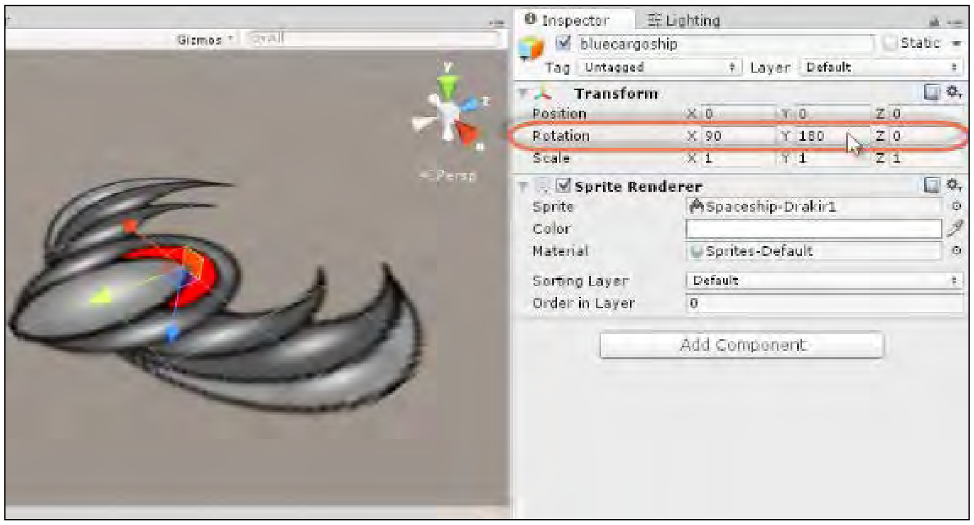
Removing a Player tag from the enemy, if applicable

Select the sprite child object of the duplicated enemy, and from the Object Inspector click inside the *Sprite* field of the *Sprite Renderer* component to pick a new sprite. Pick one of the darker imperial ships for the enemy character, and the sprite will update for the object in the viewport. See Figure 3.36.



Selecting a Sprite for the Sprite Renderer component

After changing the sprite to an enemy character you may need to adjust the rotation values to align the sprite to the parent forward vector; ensuring the sprite is looking in the same direction as the forward vector. See Figure 3.37.



bluecargoship
Static
Tag Untagged Layer Default

Transform

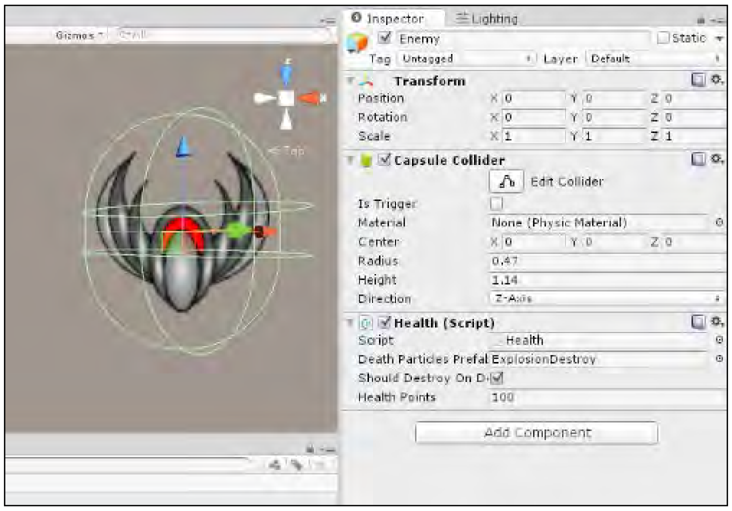
Position	X 0	Y 0	Z 0
Rotation	X 90	Y 180	Z 0
Scale	X 1	Y 1	Z 1

Sprite Renderer

Sprite	Spaceship-Drakir1
Color	
Material	Sprites-Default
Sorting Layer	Default
Order in Layer	0

Adjusting enemy sprite rotation...

Now select the parent object for the enemy, and remove the *RigidBody* component, and the *PlayerController* and *BoundsLock* Components, but keep the *Health* component as the enemy should support health. See Figure 3.38. In addition, feel free to resize the *CapsuleCollider* component to better approximate the enemy object.



Enemy
Static
Tag Untagged Layer Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Capsule Collider

Is Trigger	☐
Material	None (Physic Material)
Center	X 0 Y 0 Z 0
Radius	0.47
Height	1.14
Direction	Z-Axis

Health (Script)

Script	Health
Death Particles Prefal	ExplosionDestroy
Should Destroy On Di	✓
Health Points	100

Adjusting enemy sprite rotation...

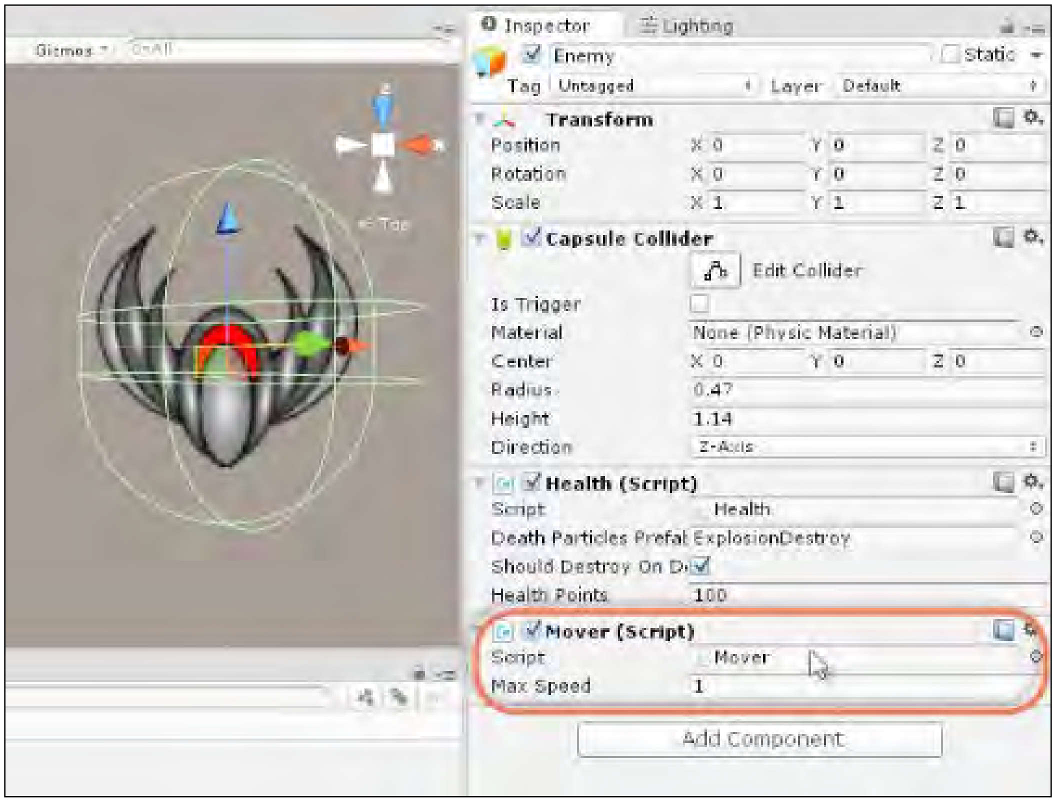
Let's start coding the enemy, focusing on movement. Specifically, the enemy should continually move in the forward direction at a specified speed. To achieve this, create a new script file, named *Mover.cs*. This should be attached to the enemy object. The code for this class is included in Code Sample 3.6.

```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class Mover : MonoBehaviour
{
   //------------------------------
   private Transform ThisTransform = null;
   public float MaxSpeed = 10f;
   //------------------------------
   // Use this for initialization
   void Awake ()
   {
         ThisTransform = GetComponent<Transform>();
   }
   //------------------------------
   // Update is called once per frame
   void Update ()
   {
         ThisTransform.position += ThisTransform.forward * MaxSpeed *
Time.deltaTime;
   }
   //------------------------------
}
//------------------------------
```

Comments on Code Sample 3-6

- The Mover Script moves an object at a specified speed (`MaxSpeed/Per Second`) along its forward vector. To do that, it uses the Transform component.
- The `Update` function is responsible for updating the position of the object. In short, it multiplies the forward vector by the object speed, and adds that onto its existing position to move the object further along its line of sight. The value `Time.deltaTime` is used to make the motion frame rate independent; moving the object per second, as opposed to per frame. More information on deltaTime can be found at the online Unity documentation here: `http://docs.unity3d.com/ScriptReference/Time-deltaTime.html`

Press *Play* on the tool bar to test run your code. It's always good practice to frequently test code like this. Your enemy may move too slow, or too fast. If so, stop playback to exit game mode, and select the enemy in the scene. From the Object Inspector, adjust the *Max Speed* value of the *Mover* component. See Figure 3.39.



Enemy ☐ Static
Tag Untagged Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Capsule Collider
Is Trigger ☐
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 0.47
Height 1.14
Direction Z-Axis
Health (Script)
Script Health
Death Particles Prefab ExplosionDestroy
Should Destroy On D ✓
Health Points 100
Mover (Script)
Script Mover
Max Speed 1

Adjusting enemy speed

In addition to moving in a straight line, the enemy should also continually turn to face the player wherever they move. To achieve this, we'll need another script file that works similarly to the player controller script. Whereas the player turns to face the cursor, the enemy turns to face the player. This functionality should be encoded in a new script file, called *ObjFace.cs*. This script should be attached to the enemy. See Code Sample 3.7.

```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class ObjFace : MonoBehaviour
```

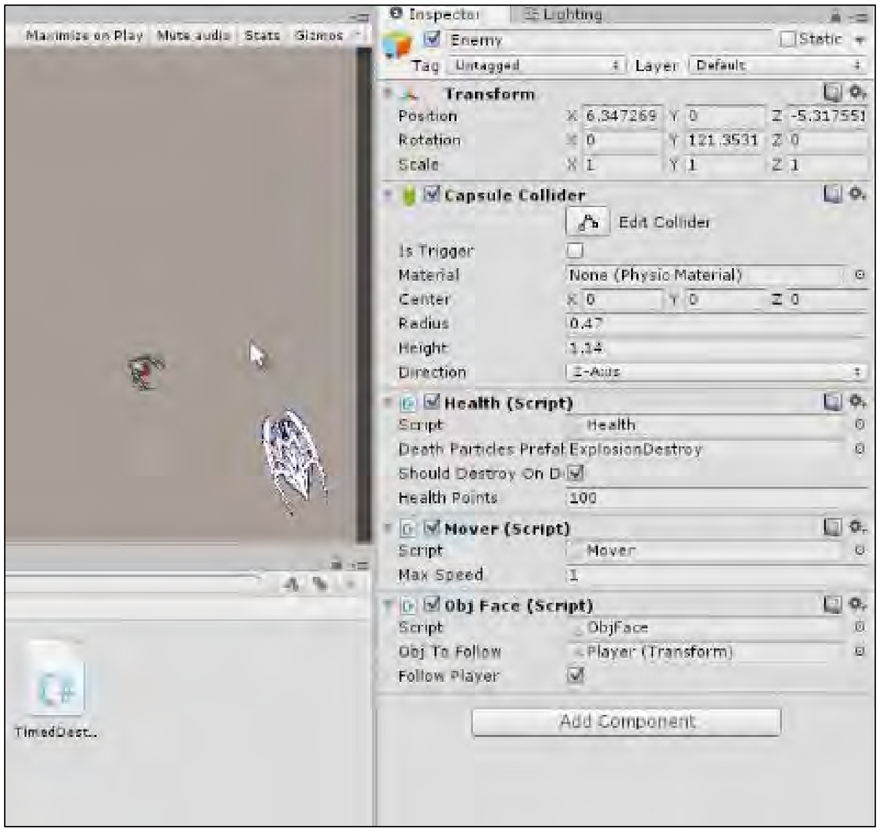
```
{
   //------------------------------
   public Transform ObjToFollow = null;
   public bool FollowPlayer = false;
   private Transform ThisTransform = null;
   //------------------------------
   // Use this for initialization
   void Awake ()
   {
         //Get local transform
         ThisTransform = GetComponent<Transform>();

         //Should face player?
         if(!FollowPlayer)return;

         //Get player transform
         GameObject PlayerObj = GameObject.FindGameObjectWithTag("Pla
yer");
         if(PlayerObj != null) ObjToFollow = PlayerObj.
GetComponent<Transform>();
   }
   //------------------------------
   // Update is called once per frame
   void Update ()
   {
         //Follow destination object
         if(ObjToFollow==null)return;

         //Get direction to follow object
         Vector3 DirToObject = ObjToFollow.position - ThisTransform.
position;

         if(DirToObject != Vector3.zero)
               ThisTransform.localRotation = Quaternion.
LookRotation(DirToObject.normalized,Vector3.up);
   }
   //------------------------------
}
//------------------------------
```

Comments on Code Sample 3-7

- The ObjFace Script will always rotate an object so that its forward vector points towards a destination point in the scene.
- Inside the Awake event, the function `FindGameObjectWithTag` is called to retrieve a reference to the one and only object in the scene tagged as a player, which should be the player space ship. The player represents the default look-at destination for an enemy object.

- The Update function is called automatically once per frame, and will generate a displacement vector from the object location to the destination location, and this represents the direction in which the object should be looking. The `Quaternion.LookRotation` function accepts a direction vector and will rotate an object to align the forward vector with the supplied direction. This keeps the object looking towards the destination. More information on `LookRotation` can be found at the online Unity documentation here: `http://docs.unity3d.com/ScriptReference/Quaternion.LookRotation.html`

This is looking excellent! But before testing this code, make sure the Player object in the scene is tagged as *Player*, and is enabled, and that the enemy is offset away from the player. Be sure to enable the check box *Follow Player* from the *ObjFace* component in the Object Inspector. When you do this, the enemy will always turn to face the player. See Figure 3.40.



Enemy ☑ · Static ☐
Tag: Untagged · Layer: Default

Transform
Position: X 6.347269, Y 0, Z -5.317551
Rotation: X 0, Y 121.3531, Z 0
Scale: X 1, Y 1, Z 1

Capsule Collider ☑
Is Trigger: ☐
Material: None (Physic Material)
Center: X 0, Y 0, Z 0
Radius: 0.47
Height: 1.34
Direction: Z-Axis

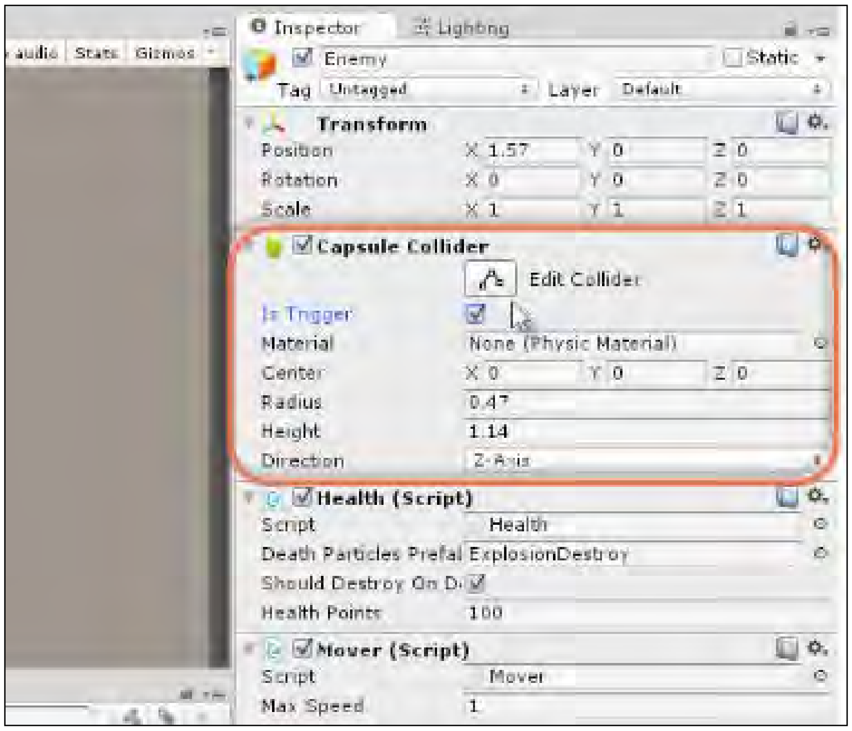
Health (Script) ☑
Script: Health
Death Particles Prefal: ExplosionDestroy
Should Destroy On D: ☑
Health Points: 100

Mover (Script) ☑
Script: Mover
Max Speed: 1

Obj Face (Script) ☑
Script: ObjFace
Obj To Follow: Player (Transform)
Follow Player: ☑

Enemy space ship moving towards player...

Now, if and when the enemy finally collides with the player, it should deal damage and potentially kill the player. To achieve this, a collision between the enemy and player must be detected. Let's start by configuring the enemy. Select the enemy object, and from the Object Inspector enable the check box *Is Trigger* in the *Capsule Collider* component. This changes the Capsule Collider component to allow for a true intersection between the player and enemy, prevent Unity from blocking the collision. See Figure 3.41.



Enemy ✓ · Static
Tag: Untagged · Layer: Default

Transform
Position: X 1.57 Y 0 Z 0
Rotation: X 0 Y 0 Z 0
Scale: X 1 Y 1 Z 1

Capsule Collider ✓
Edit Collider
Is Trigger: ✓
Material: None (Physic Material)
Center: X 0 Y 0 Z 0
Radius: 0.47
Height: 1.14
Direction: Z-Axis

Health (Script) ✓
Script: Health
Death Particles Prefal: ExplosionDestroy
Should Destroy On D: ✓
Health Points: 100

Mover (Script) ✓
Script: Mover
Max Speed: 1

Changing the Enemy Collider to a Trigger

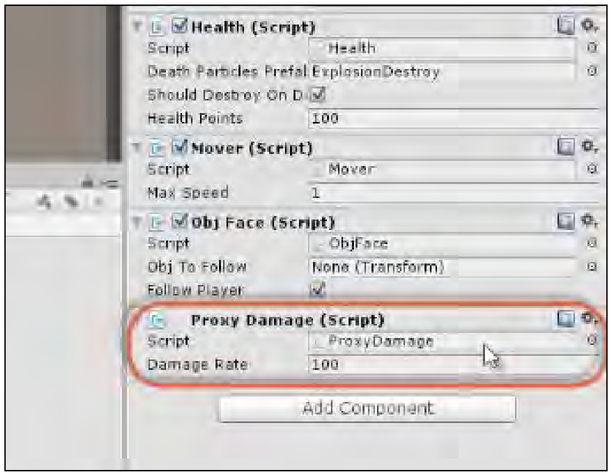
Next, we'll create a script that detects collisions and will continually deal damage to the player as and when they occur, and for as long as the collision state remains. See the following Code Sample 3.8 (*ProxyDamage.cs*), which should be attached to the enemy character.

```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class ProxyDamage : MonoBehaviour
{
    //------------------------------
    //Damage per second
    public float DamageRate = 10f;
    //------------------------------
    void OnTriggerStay(Collider Col)
    {
           Health H = Col.gameObject.GetComponent<Health>();

           if(H == null)return;

           H.HealthPoints -= DamageRate * Time.deltaTime;
    }
    //------------------------------
}
//------------------------------
```

Comments on Code Sample 3-8

- The script *ProxyDamage* should be attached to an enemy character, and it will deal damage to any colliding object with a health component.
- The event `OnTriggerStay` is called once every frame for as long as an intersection state persists.

After attaching the *ProxyDamage* script to an enemy, use the Object Inspector to set the *Damage Rate* of the *ProxyDamage* component. This represents how much health should be reduced on the player, per second, during a collision. For a challenge, I've set the value to 100 health points. See Figure 3.42.



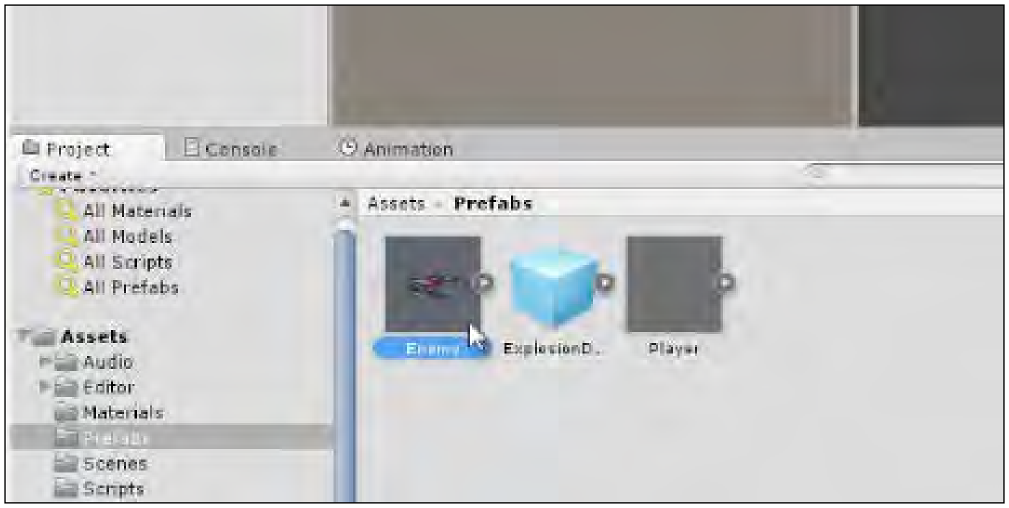
```
Health (Script)
Script                   Health
Death Particles Prefal   ExplosionDestroy
Should Destroy On D      ✓
Health Points            100
Mover (Script)
Script                   Mover
Max Speed                1
Obj Face (Script)
Script                   ObjFace
Obj To Follow            None (Transform)
Follow Player            ✓
Proxy Damage (Script)
Script                   ProxyDamage
Damage Rate              100
Add Component
```

Setting the Damage Rage for a ProxyDamage Component

Now let's give things a test run. Press Play on the toolbar and attempt a collision between the player and enemy. After 1 second, the player should be destroyed. Things are coming along well. But, we'll need more than one enemy to make things challenging…

Enemy Spawning

To make the level fun and challenging we'll need more than simply one enemy. In fact, for a game that's essentially endless we'll need to continually add enemies. These should be added gradually over time. Essentially, we'll need either regular or intermittent spawning of enemies, and this section will add that functionality. Before we can do that, however, we'll need to make a prefab from the enemy object. That can be achieved easily: select the enemy in the Hierarchy Panel, and then drag and drop it into the Project Panel, in the Prefabs folder. This creates an Enemy Prefab. See Figure 3.43.



Creating an Enemy Prefab

Now we'll make a new script (*Spawner.cs*) that spawns new enemies in the scene over time, within a specified radius from the Player Space ship. This script should be attached to a new, empty game object inside the scene. See Code Sample 3.9.

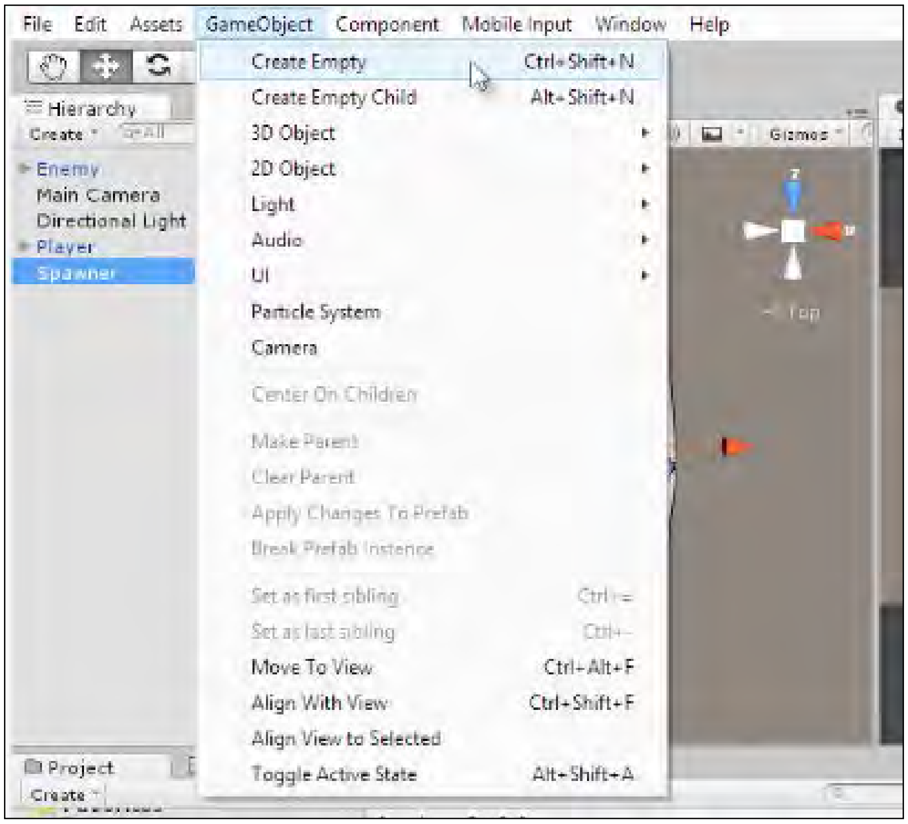
```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class Spawner : MonoBehaviour
```

```
{
   public float MaxRadius = 1f;
   public float Interval = 5f;
   public GameObject ObjToSpawn = null;
   private Transform Origin = null;
   //-----------------------------
   void Awake()
   {
         Origin = GameObject.FindGameObjectWithTag("Player").
GetComponent<Transform>();
   }
   //-----------------------------
   // Use this for initialization
   void Start ()
   {
         InvokeRepeating("Spawn", 0f, Interval);
   }
   //-----------------------------
   void Spawn ()
   {
         if(Origin == null)return;

         Vector3 SpawnPos = Origin.position + Random.onUnitSphere *
MaxRadius;
         SpawnPos = new Vector3(SpawnPos.x, 0f, SpawnPos.z);
         Instantiate(ObjToSpawn, SpawnPos, Quaternion.identity);
   }
   //-----------------------------
}
//-----------------------------
```

Comments on Code Sample 3-9

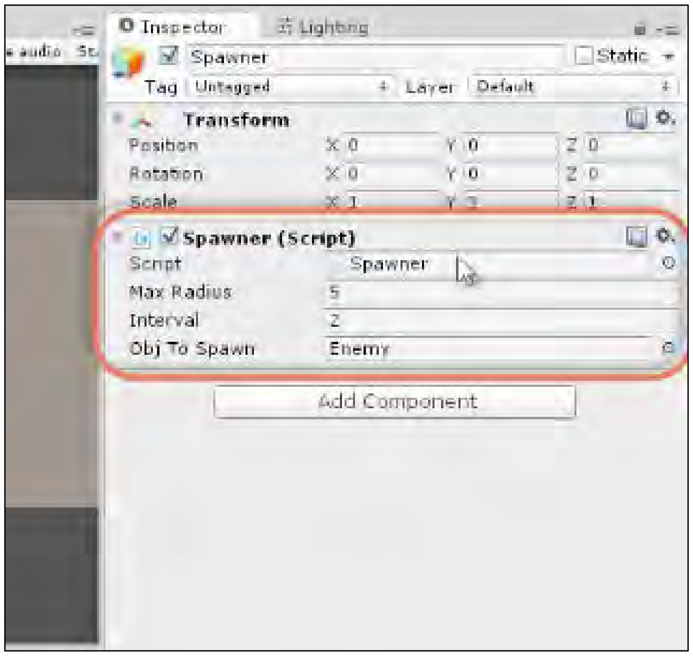
- The `Spawner` class will spawn instances of `ObjToSpawn` on each interval of `Interval`. The interval is measured in seconds. The spawned objects will be created within a random radius from a center point `Origin`.
- During the `Start` event, the function `InvokeRepeating` is called to continually execute the Spawn function on every interval.
- The `Spawn` function will create instances of the enemy in the scene, at a random radius from an origin point. Once spawned, the enemy will behave as normal, heading towards the player.

The Spawner class is a global behavior that applies scene wide. It does not depend on the player specifically, and nor on any specific enemy. For this reason it should be attached to an empty game object. Create one of these by selecting *GameObject* > *Create Empty* from the application menu. Name this *Spawner*, and attach the *Spawner* script to it. See Figure 3.44.



Creating an Empty Game Object

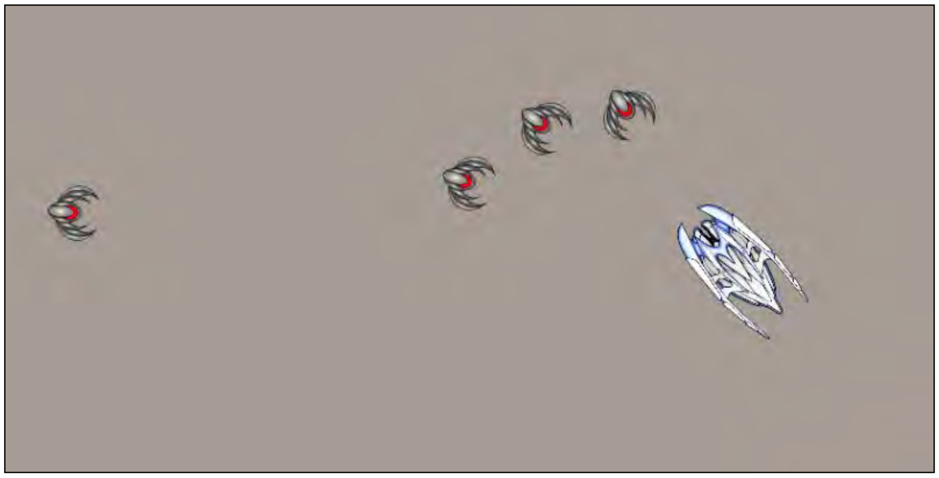
Once added to the scene, from the Object Inspector drag and drop the Enemy prefab into the *Obj To Spawn* field in the *Spawner* component. Set the *interval* to 2 seconds and increase the *Radius* to 5. See Figure 3.45.



Spawner
Tag Untagged Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Spawner (Script)
Script Spawner
Max Radius 5
Interval 2
Obj To Spawn Enemy

Configuring the Spawner for Enemy Objects

And now (drum roll) let's try the level. Press Play on the tool bar and take the game for a test run. You should now have a level with a fully controllable player character surrounding by a growing army of tracking enemy ships! Excellent work. See Figure 3.46.



Spawned enemy objects moving towards the player

Summary

Good job on reaching this far The space shooter is really taking shape now, featuring a controllable player character that relies on native physics, twin-stick mechanics, enemy ships, and a scene wide spawner for enemies. All these ingredients together still don't make a game: we can't shoot, we can't increase the score, and we can't destroy enemies. These issues will need to be addressed, alongside other technical issues that we'll certainly encounter. Nevertheless, we now have a solid foundation for moving further, and in the next chapter we'll complete the shooter.

4 Continuing the Space Shooter

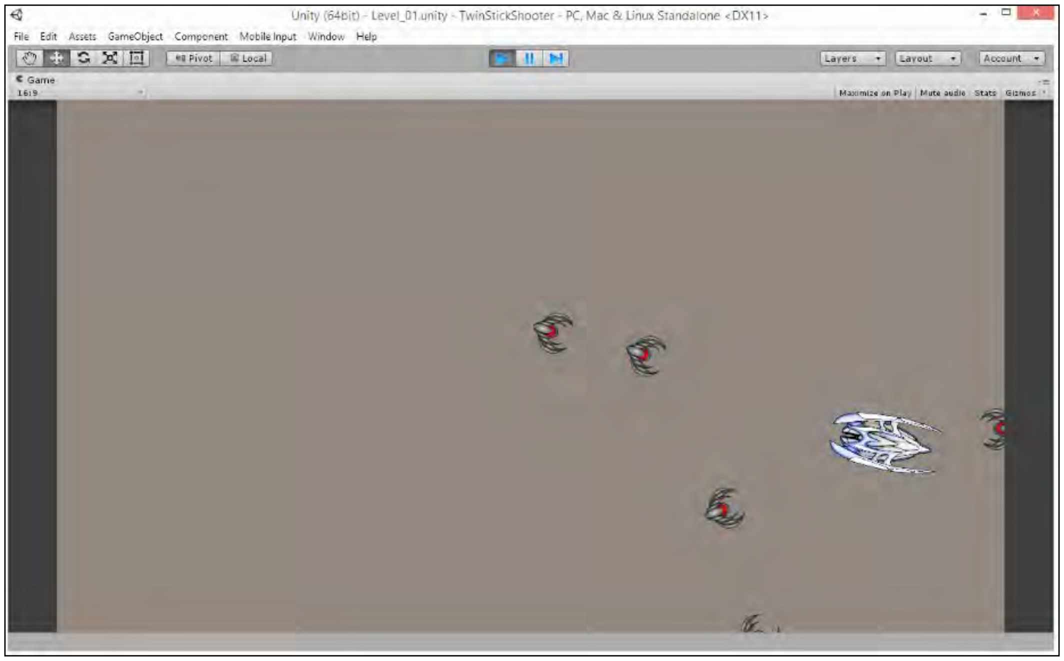
This chapter continues from the previous in creating a twin stick space shooter game. At this stage, we have a working game. At least, the gamer can control a space ship using two axes: movement and rotation. WASD keys on the keyboard control movement (up, down, left and right), and the mouse cursor controls rotation; the space ship always rotates to face the cursor. In addition to player controls, the level features enemy characters that spawn at regular intervals, fly around the level, and move towards the player with hostile intent. And finally, both the player and enemies support a Health component, which means both are susceptible to damage and can be destroyed. Right now, however, the player lacks two important features: they cannot fire a weapon and they cannot increase their score. This chapter tackles these issues and more. Firing weapons, as we'll see, represents a particularly interesting problem. Overall, this chapter considers:

- Weapons and Spawning Ammo
- Memory Management and Pooling
- Sound and Audio
- Scoring
- Debugging and Testing
- Building and Distribution

[The completed project so far can be found in the book companion files, in the Chapter04/Start folder. You can start here and follow along with this chapter, if you don't have your own project already]

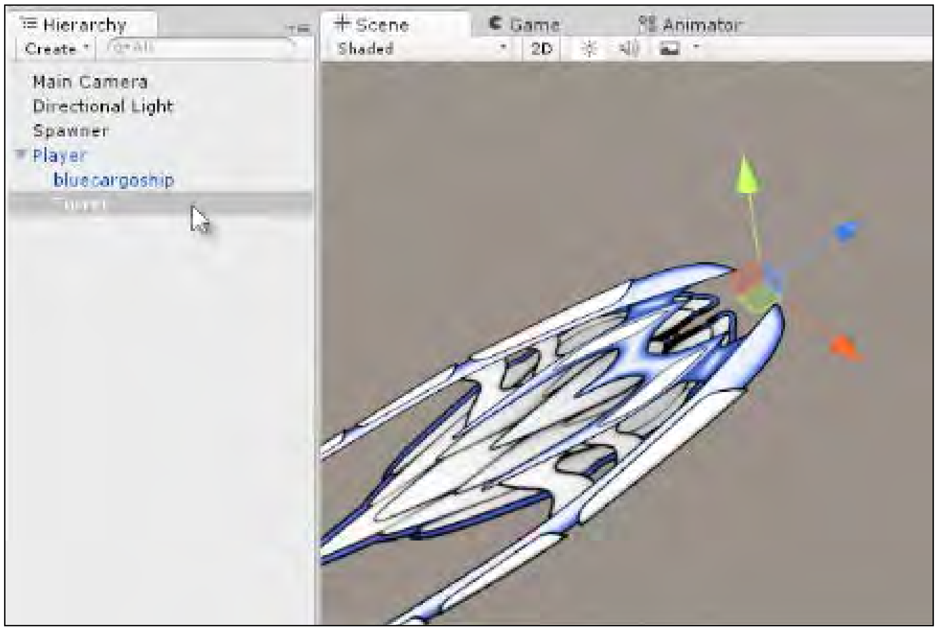
Guns and Gun Turrets

Let's start tackling weapons in detail. Specifically, the level contains a player and enemy ships. The player must shoot enemies but, right now, cannot do so. See Figure 4.1. On thinking carefully about weapons we identify three main concepts or 'things' which needs development: First, there's the spawner or generator- the object that actually fires ammo into the scene when the fire button is pressed. Second, there's the ammo itself which, once generated, travels through the level on its own. And third, there's the ability for ammo to collide with other objects and to damage them.


Unity (64bit) - Level_01.unity - TwinStickShooter - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Mobile Input Window Help
Pivot Local
Layers Layout Account
Game
16:9
Maximize on Play Mute audio Stats Gizmos

The game so far...

Tackling each area in order, we begin with turrets- the points where bullets are spawned and fired. For this game, the player will have only one turret; but ideally the game should support the addition of more, if desired, allowing the player to dual-fire, or more! To create the first turret, add a new Empty game object to the scene by selecting *GameObject* > *Create Empty* from the application menu. Name this *Turret*. Then position the turret object to the front of the space ship, making sure the blue forward-vector arrow is pointing ahead, in the direction that ammo will be fired. Then finally make the turret a child of the space ship by dragging and dropping inside the hierarchy panel. See Figure 4.2.

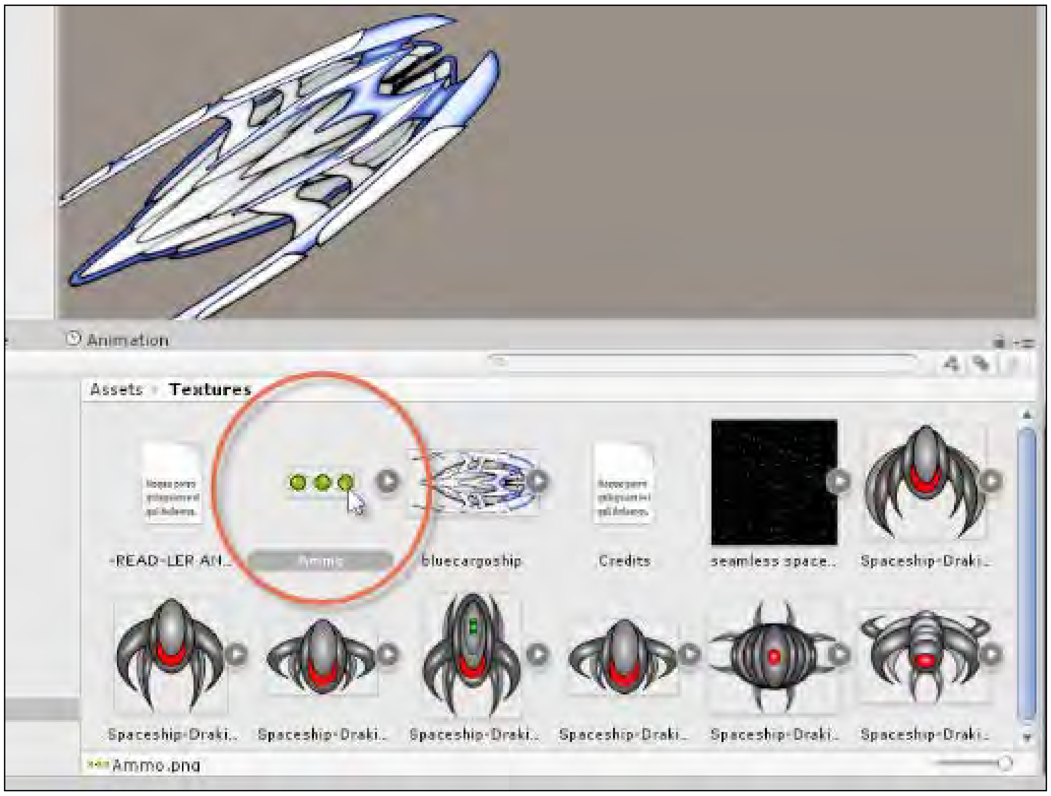


Positioning a Turret Object as a child of the Space Ship

Creating a turret object for the ammo as a spawn location is a splendid beginning, but for ammo to actually be fired, we'll need an ammo object. Specifically, we'll create an ammo prefab that can be instantiated as ammo, when needed. We'll do that next.

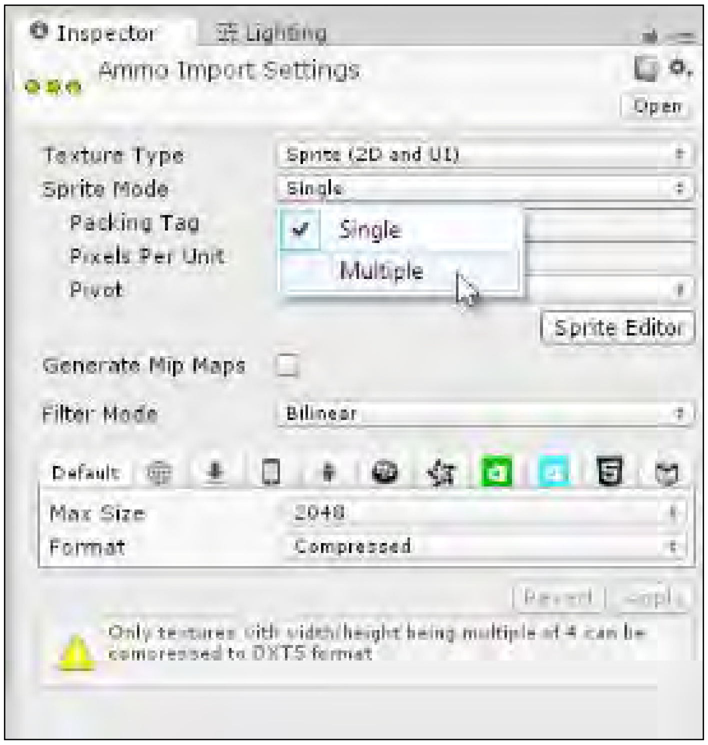
Ammo Prefabs

When the player presses the fire button, the spaceship should shoot ammo objects into the scene. These objects will be based on an ammo prefab. Let's create that prefab now. To start, we'll configure the texture to be used as an ammo graphic. Open the *Textures* folder inside the Project Panel, and select the *Ammo* Texture. This texture features several different versions of an ammo sprite, aligned in a row side by side. See Figure 4.3. When ammo is fired we don't want to show the complete texture; instead, we want to show either just one of the images or the images played as animation sequence, frame by frame.



Preparing to create an Ammo Prefab

Presently, Unity recognizes the texture (and each ammo element) as a complete unit. We can use the Sprite Editor, however, to separate each part. To do that, select the Texture in the Project (if it's not already selected), and then (from the Object Inspector) change the *Sprite Mode* drop-down from *Single* to *Multiple*. This signifies that more than one sprite is contained within the texture space. See Figure 4.4.



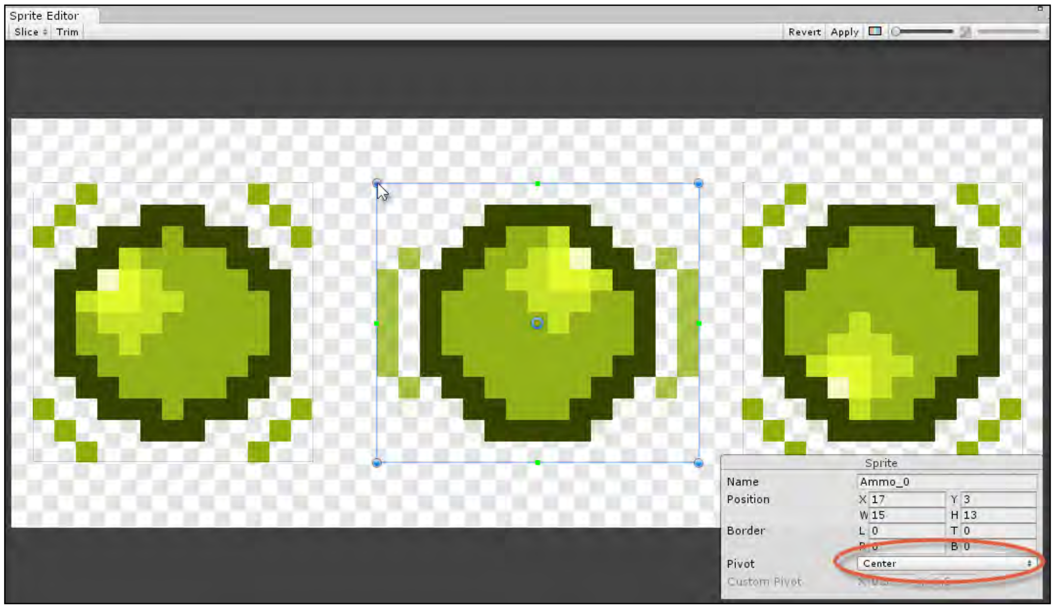
Ammo Import Settings

Texture Type: Sprite (2D and UI)
Sprite Mode: Single
Packing Tag
Pixels Per Unit
Pivot
Generate Mip Maps
Filter Mode: Bilinear
Max Size: 2048
Format: Compressed

Only textures with width/height being multiple of 4 can be compressed to DXT5 format

Select Multiple Sprites for textures featuring more than one sprite

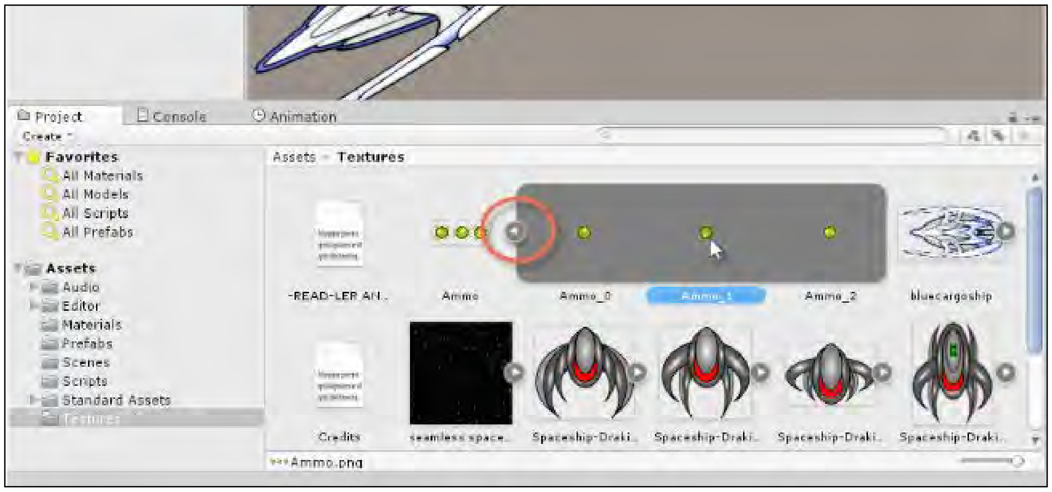
Click the *Apply* button, and then click the *Sprite Editor* button from the Object Inspector. This opens the sprite editor, allowing you to separate each sprite. To do this, click and drag your mouse to box-select each individual sprite, making sure the pivot is aligned to the object center. See Figure 4.5. Then Click *Apply* to accept the changes.



Name	Ammo_0
Position	X 17 Y 3
	W 15 H 13
Border	L 0 T 0
	R 0 B 0
Pivot	Center

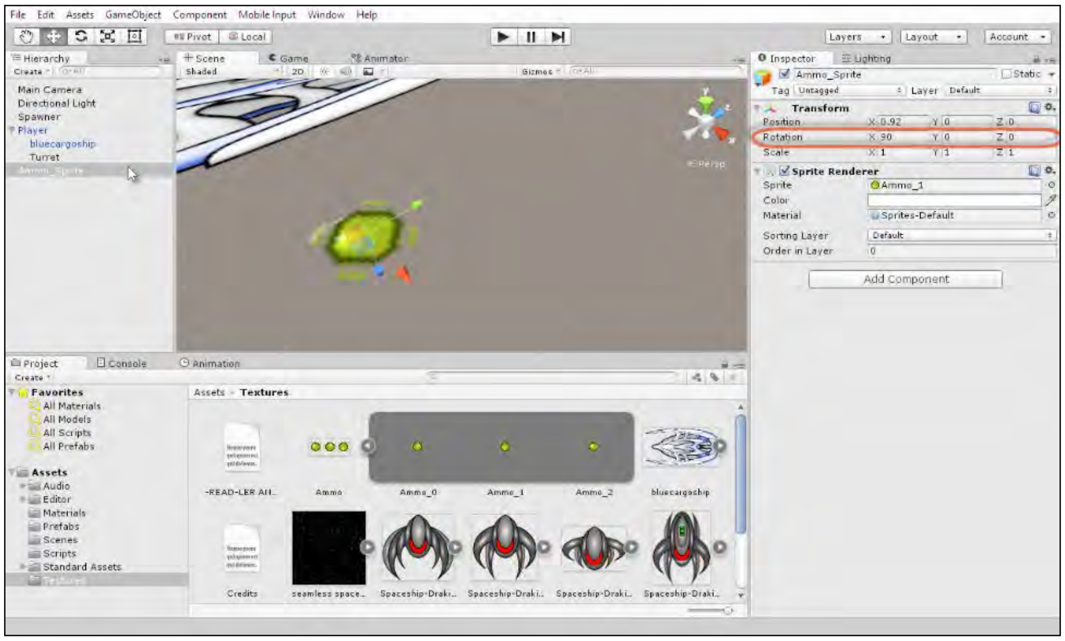
Separating multiple sprites within the Sprite Editor

After accepting the changes inside the Sprite Editor, Unity automatically cuts the relevant sprites into separate units, each of which can now be selected as a separate object inside the Project Panel. Click the right-arrow at the side of the texture, and all sprites within will expand outwards. See Figure 4.6.



Expand all sprites within a texture

Now drag and drop one of the sprites from the Project Panel into the Scene, via the Hierarchy Panel. On doing this, it will be added as a sprite object. This represents the beginning of our ammo prefab. The sprite itself may not initially be oriented to face upwards at the game camera. If so, rotate the sprite by 90 degrees until it looks correct. See Figure 4.7.



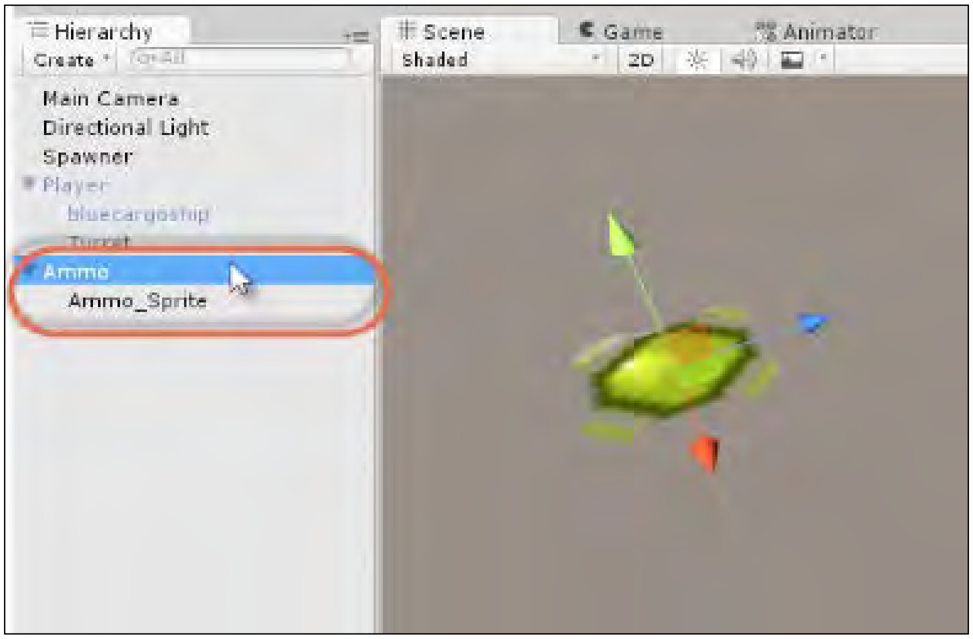
Ammo_Sprite Static
Tag Untagged Layer Default

Transform
Position X 0.92 Y 0 Z 0
Rotation X 90 Y 0 Z 0
Scale X 1 Y 1 Z 1

Sprite Renderer
Sprite Ammo_1
Color
Material Sprites-Default
Sorting Layer Default
Order in Layer 0

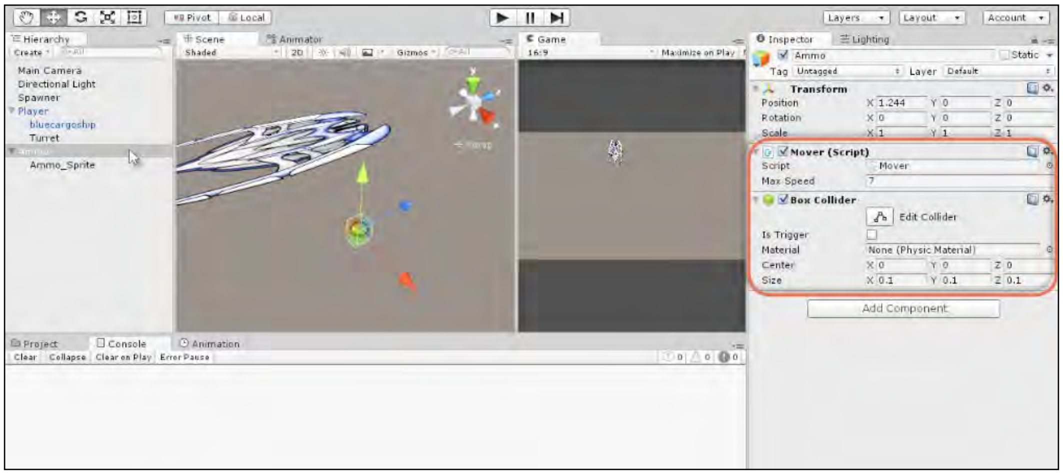
Aligning the ammo sprite

Now create a new, empty game object in the scene (*GameObject* > *Create Empty* from the application menu), and rename it to *Ammo*. Make this new object a parent of the ammo sprite, and ensure its local forward vector is pointing in the direction the ammo should travel. We'll soon reuse the *Mover* script (Created in the previous chapter) on the ammo to make it move.



Building an Ammo Object

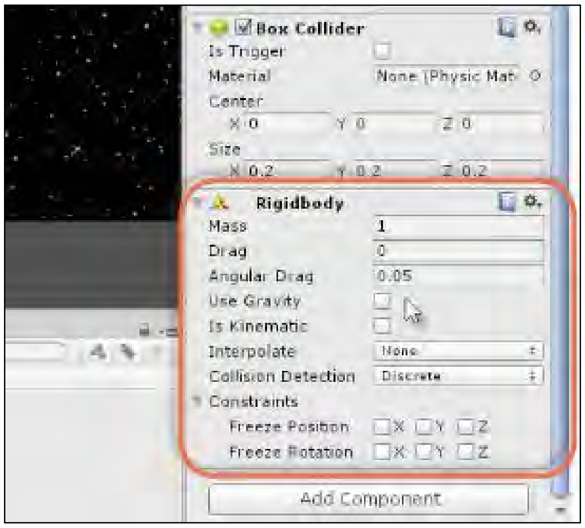
Drag and drop the *Mover.cs* script from the Project Panel onto the Ammo parent object, via the Hierarchy Panel, to add it as a component. Then select the Ammo object and, from the Object Inspector, change the ammo *Max Speed* in the *Mover* component to 7. And finally, add a box collider to the object for approximating its volume (*Component* > *Physics* > *Box Collider* from the application menu), and then test this all in the viewport by pressing *Play* on the toolbar. The Ammo object should shoot forwards, as though fired from a weapon. If it incorrectly moves up or down, then make sure the parent object is rotated so that its blue, forward vector really is pointing forwards. See Figure 4.9.



```
Ammo                                   Static
Tag  Untagged              Layer  Default
Transform
Position     X 1.244   Y 0     Z 0
Rotation     X 0       Y 0     Z 0
Scale        X 1       Y 1     Z 1
Mover (Script)
Script       Mover
Max Speed    7
Box Collider
             Edit Collider
Is Trigger   ☐
Material     None (Physic Material)
Center       X 0       Y 0     Z 0
Size         X 0.1     Y 0.1   Z 0.1
```

Moving forwards with an Ammo Prefab (Mover and Collider)

Next, add a *RigidBody* component to the ammo, to make it part of the Unity physics system. To do this, select the Ammo object and choose *Component* > *Physics* > *Rigidbody* from the application menu. Then, from the Rigidbody component in the Object Inspector, disable the check box *Use Gravity*, to prevent the ammo from falling to the ground during gameplay. For our purposes, gravity need not apply to the ammo, since it should simply travel along and eventually be destroyed. This highlights an important point in game development generally: real world physics need not apply to every object accurately. We only need enough physics to make objects *appear* correct to the player when they're looking. See Figure 4.10.



```
Box Collider
Is Trigger           ☐
Material             None (Physic Mat
Center
  X 0        Y 0        Z 0
Size
  X 0.2      Y 0.2      Z 0.2
Rigidbody
Mass                 1
Drag                 0
Angular Drag         0.05
Use Gravity          ☐
Is Kinematic         ☐
Interpolate          None
Collision Detection  Discrete
Constraints
  Freeze Position    ☐X ☐Y ☐Z
  Freeze Rotation    ☐X ☐Y ☐Z
```

Removing Gravity from the Ammo Object

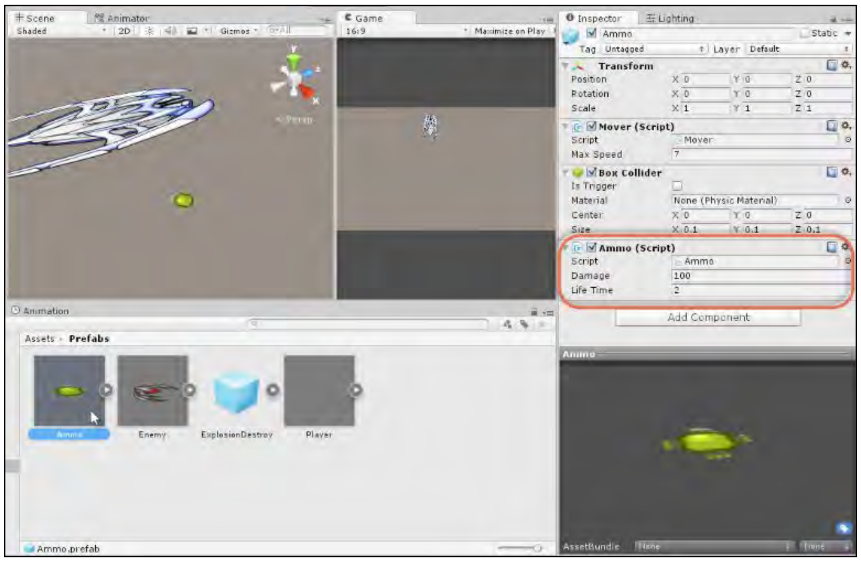
In addition to adding a mover script and physics components, we also need the ammo to behave distinctly. Specifically, it should damage the objects with which it collides, and it should also destroy or disable itself on collision. To achieve this, a new script file must be created, *Ammo.cs*. The full code for this is included in Sample 4-1 below.

```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class Ammo : MonoBehaviour
{
   public float Damage = 100f;
   public float LifeTime = 2f;
   //------------------------------
   void OnEnable()
   {
          CancelInvoke();
          Invoke("Die", LifeTime);
   }
   //------------------------------
   // Update is called once per frame
   void OnTriggerEnter(Collider Col)
   {
          //Get health component
Health H = Col.gameObject.GetComponent<Health>();

          if(H == null)return;

          H.HealthPoints -= Damage;
   }
   //------------------------------
   void Die()
   {
          gameObject.SetActive(false);
   }

   //------------------------------
}
//------------------------------
```

Comments on Code Sample 4-1

- The Ammo class should be attached to the Ammo prefab object, and will be instantiated for all ammo objects created. Its main purpose is to damage any objects with which it collides
- The *OnTriggerEnter* function is invoked for the ammo when it enters a trigger attached to a moveable unit, like the player or enemies. Specifically, it retrieves the *Health* component attached to the object, if it has one, and reduces its health by the *Damage* amount. The Health component was created in the previous chapter
- Notice also that each ammo object will have a *Lifetime*. This represents the amount of time in seconds for which the ammo should remain 'alive' and 'active' after it is fired and generated in the scene. After the lifetime expires, the ammo should either be destroyed entirely, or deactivated (more on this shortly).
- The *Invoke* function is used to deactivate the ammo object after the *Lifetime* interval. This happens during the *OnEnable* event. This is called automatically by Unity *each time* an Object is activated (that is, changed from being disabled to enabled).

Now drag and drop the Ammo script file from the *Scripts* folder in the Project Panel onto the Ammo object, and then finally drag and drop the whole Ammo object in the scene back into the Project Panel, inside the *Prefabs* folder, to create a new Ammo Prefab, See Figure 4.11.



Ammo — Static
Tag: Untagged — Layer: Default

Transform
Position: X 0, Y 0, Z 0
Rotation: X 0, Y 0, Z 0
Scale: X 1, Y 1, Z 1

Mover (Script)
Script: Mover
Max Speed: 7

Box Collider
Is Trigger: ☐
Material: None (Physic Material)
Center: X 0, Y 0, Z 0
Size: X 0.1, Y 0.1, Z 0.1

Ammo (Script)
Script: Ammo
Damage: 100
Life Time: 2

Creating an Ammo Prefab

Congratulations! You've now created an Ammo Prefab, which can be spawned from weapon points to attack enemies directly. This is good, but we've still not handled the spawning process itself, and we'll address that next.

Ammo Spawning

The Ammo prefab created so far presents us with a technical problem which, if not taken seriously, has the potential to cause some serious performance penalties for our game. Specifically, when the space ship weapon is fired, we'll need to generate ammo that launches into the scene and destroys the enemies on collision. This is fine in general, but the problem is that the player could potentially press the fire button many times in quick succession, and could even hold down the fire button for long periods of time, and thereby spawn potentially hundreds of ammo prefabs. We could, of course, use the *Instantiate* function seen already to generate those prefabs dynamically, but this is problematic because instantiate is computationally expensive. When used to generate many items in succession, it will typically cause a nightmarish slow-down that'll reduce the FPS to unacceptable levels. We need to avoid this!

The solution is known as 'Pooling', or 'Object Pooling' or 'Object Caching'. In essence, it means we must spawn a large and recyclable batch of ammo objects at the level start-up (a pool of objects) which, initially, begin hidden or deactivated, and we simply activate the objects as and when needed (when the player fires a weapon). When the ammo collides with an enemy, or when its lifetime expires, we don't destroy the object entirely; we simply deactivate it again, returning it to the pool for re-use later if needed. In this way, we avoid all calls to *Instantiate* and simply re-cycle all ammo objects that we have. To get started at coding this functionality, we'll make an *AmmoManager* class. This class will be responsible for two features: first, generating a pool of ammo objects at scene start-up; and second, for giving us a valid and available ammo object from the pool on demand, such as on weapon-fire. Consider the following *AmmoManager* code in Sample 4.2 to achieve this, as below.

```
//------------------------------
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
//------------------------------
public class AmmoManager : MonoBehaviour
{
   //------------------------------
   //Reference to ammo prefab

   public GameObject AmmoPrefab = null;
```

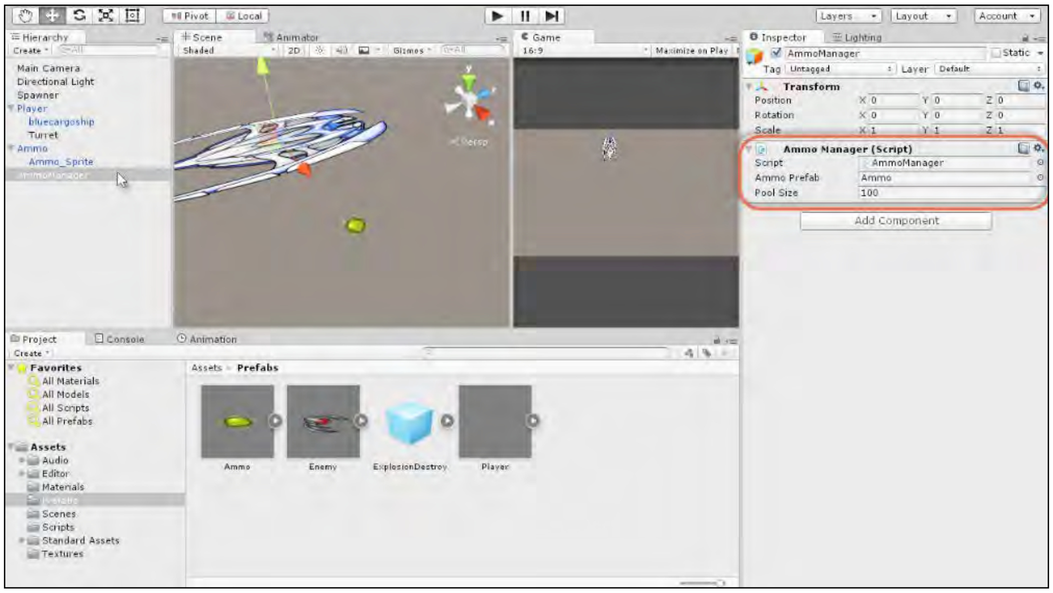
```
    //Ammo pool count
    public int PoolSize = 100;

    public Queue<Transform> AmmoQueue = new Queue<Transform>();

    //Array of ammo objects to generate
    private GameObject[] AmmoArray;

    public static AmmoManager AmmoManagerSingleton = null;
    //-----------------------------
    // Use this for initialization
    void Awake ()
    {
          if(AmmoManagerSingleton != null)
          {
                Destroy(GetComponent<AmmoManager>());
                return;
          }

          AmmoManagerSingleton = this;
          AmmoArray = new GameObject[PoolSize];

          for(int i=0; i<PoolSize; i++)
          {
                AmmoArray[i] = Instantiate(AmmoPrefab, Vector3.zero, 
Quaternion.identity) as GameObject;
                Transform ObjTransform = AmmoArray[i].
GetComponent<Transform>();
                ObjTransform.parent = GetComponent<Transform>();
                AmmoQueue.Enqueue(ObjTransform);
                AmmoArray[i].SetActive(false);
          }
    }
    //-----------------------------
    public static Transform SpawnAmmo(Vector3 Position, Quaternion 
Rotation)
    {
          //Get ammo
          Transform SpawnedAmmo = AmmoManagerSingleton.AmmoQueue.
Dequeue();

          SpawnedAmmo.gameObject.SetActive(true);
          SpawnedAmmo.position = Position;
          SpawnedAmmo.localRotation = Rotation;
```

```
            //Add to queue end
            AmmoManagerSingleton.AmmoQueue.Enqueue(SpawnedAmmo);

            //Return ammo
            return SpawnedAmmo;
        }
        //------------------------------
    }
    //------------------------------
```

Comments on Code Sample 4-2

- The `AmmoManager` features an `AmmoArray` member variable, which holds a complete list of all ammo objects generated at start-up (during the `Awake` event).
- The `AmmoArray` will be sized to `PoolSize`. This refers to the total number of ammo objects to be generated.
- Once generated, each ammo object is deactivated with `SetActive(false)` and is held in the pool until needed.
- `AmmoManager` uses the class `Queue`, from the Mono Library, to manage how specific ammo objects are selected from the pool to be activated when fire is pressed. The queue works as a FIFO object (First in, First out). That is, ammo objects are added to the queue, one at a time, and can be removed when selected to be activated. The object removed from the queue is always the object at the front. More information on the Queue class can be found online here: `https://msdn.microsoft.com/en-us/library/7977ey2c%28v=vs.110%29.aspx`
- The `Enqueue` function of the `Queue` object is called during `Awake` to add objects initially into the queue, one by one, as they are generated.
- The `SpawnAmmo` function should be called to generate a new item of ammo in the scene. This function does not rely on the `Instantiate` function, but uses the `Queue` object instead. It removes the first ammo object from the queue, activates it, and then adds it to the end of the queue again, behind all the other ammo objects. In this way, a cycle of generation and re-generation happens, allowing all ammo objects to be recycled.
- The `AmmoManager` is coded as a Singleton Object, meaning that one and only one instance of the object should exist in the scene at any one time. This functionality is achieved through the static member `AmmoManagerSingleton`. For more information on Singleton Objects, see my Packt book 'Mastering Unity Scripting;: `https://www.packtpub.com/game-development/mastering-unity-5x-scripting`

To use this class, create a new GameObject in the scene, called *AmmoManager*, by selecting *GameObject* > *Create Empty* from the application menu. Then drag and drop the *AmmoManager* script from the Project Panel onto the Object in the scene. Once created, drag and drop the Ammo Prefab, from the prefabs folder, into the *AmmoPrefab* slot for the *AmmoManager* component in the Object Inspector See Figure 4.12.



Ammo Manager (Script)
Script: AmmoManager
Ammo Prefab: Ammo
Pool Size: 100

Adding the Ammo Manager to an Object

Now the scene features an *AmmoManager* object for holding an Ammo Pool, off screen and hidden. But still nothing about our existing functionality actually connects a fire button press, from the gamer, with the generation of ammo in the scene. That is, we have no code to actually make the ammo visible and working! This connection should now be made, via the *PlayerController* script that we started in the previous chapter. This class should now be amended to handle ammo generation. The recoded *PlayerController* class is included in code sample 4-3 below. Amendments are highlighted.

```
//------------------------------
using UnityEngine;
using System.Collections;
//------------------------------
public class PlayerController : MonoBehaviour
{
   //------------------------------
   private Rigidbody ThisBody = null;
```

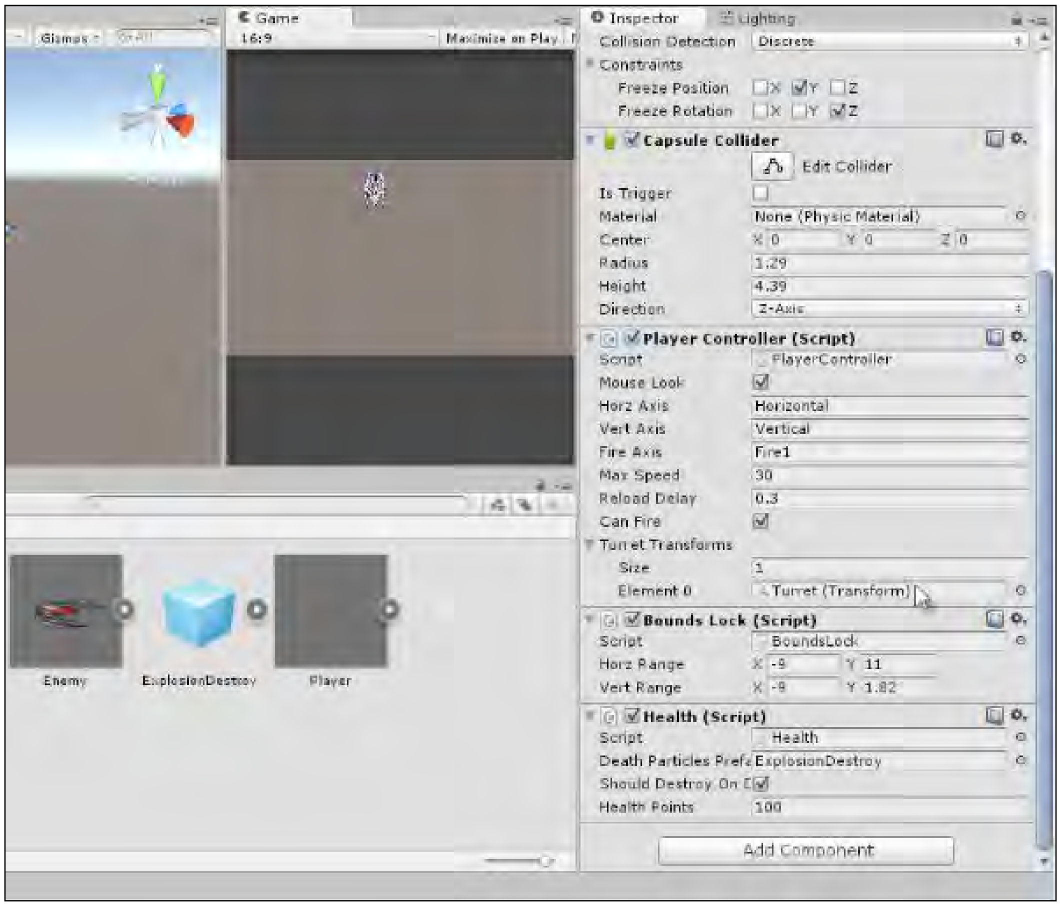
```
    private Transform ThisTransform = null;

    public bool MouseLook = true;
    public string HorzAxis = "Horizontal";
    public string VertAxis = "Vertical";
    public string FireAxis = "Fire1";

    public float MaxSpeed = 5f;
    public float ReloadDelay = 0.3f;
    public bool CanFire = true;

    public Transform[] TurretTransforms;
    //------------------------------
    // Use this for initialization
    void Awake ()
    {
          ThisBody = GetComponent<Rigidbody>();
          ThisTransform = GetComponent<Transform>();
    }
    //------------------------------
    // Update is called once per frame
    void FixedUpdate ()
    {
          //Update movement
          float Horz = Input.GetAxis(HorzAxis);
          float Vert = Input.GetAxis(VertAxis);
          Vector3 MoveDirection = new Vector3(Horz, 0.0f, Vert);
          ThisBody.AddForce(MoveDirection.normalized * MaxSpeed);

          //Clamp speed
          ThisBody.velocity = new Vector3(Mathf.Clamp(ThisBody.
velocity.x, -MaxSpeed, MaxSpeed),
                                          Mathf.Clamp(ThisBody.
velocity.y, -MaxSpeed, MaxSpeed),
                                          Mathf.Clamp(ThisBody.
velocity.z, -MaxSpeed, MaxSpeed));

          //Should look with mouse?
          if(MouseLook)
          {
                //Update rotation - turn to face mouse pointer
                Vector3 MousePosWorld = Camera.main.
ScreenToWorldPoint(new Vector3(Input.mousePosition.x, Input.
mousePosition.y, 0.0f));
```

```
                MousePosWorld = new Vector3(MousePosWorld.x, 0.0f,
MousePosWorld.z);

                //Get direction to cursor
                Vector3 LookDirection = MousePosWorld -
ThisTransform.position;

                //FixedUpdate rotation
                ThisTransform.localRotation = Quaternion.
LookRotation(LookDirection.normalized,Vector3.up);
          }

          //Check fire control
          if(Input.GetButtonDown(FireAxis) && CanFire)
          {
                foreach(Transform T in TurretTransforms)
                      AmmoManager.SpawnAmmo(T.position, T.rotation);

                CanFire = false;
                Invoke ("EnableFire", ReloadDelay);
          }
    }
    //------------------------------
    void EnableFire()
    {
          CanFire = true;
    }
    //------------------------------
    public void Die()
    {
          Destroy(gameObject);
    }
}
//------------------------------
```

Comments on Code Sample 4-3

- `PlayerController` now features an array variable `TurretTransform`, listing all child empties being used as turret spawn locations.
- During the `Update` function, the `PlayerController` checks for Fire button presses. If detected, the code cycles through all turrets and spawns one ammo object at each turret location.
- Once ammo is fired, a `ReloadDelay` is engaged. This means the delay must first expire before new ammo can be fired again.

After adding this code to the *PlayerController*, select the Player object in the scene and then drag and drop the Turret empty into the *TurretTransform* slot. This example uses only one turret, but you could add more if desired. See Figure 4.13.

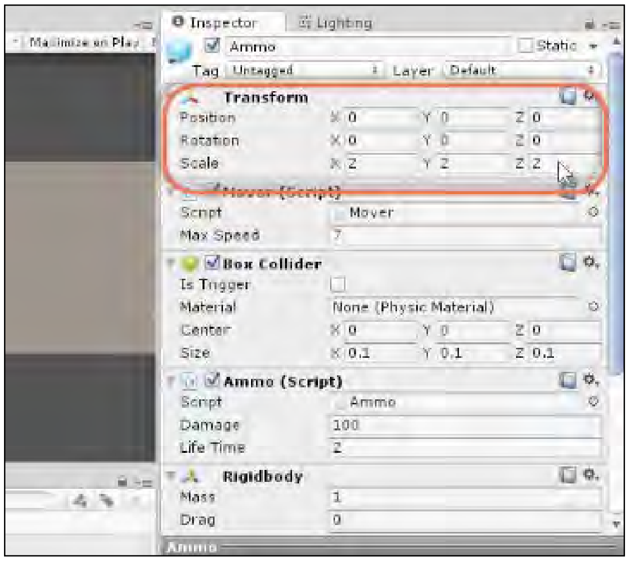


```
Collision Detection      Discrete
Constraints
  Freeze Position        X [✓ Y] Z
  Freeze Rotation        X Y [✓ Z]
Capsule Collider
  Is Trigger
  Material               None (Physic Material)
  Center                 X 0   Y 0   Z 0
  Radius                 1.29
  Height                 4.39
  Direction              Z-Axis
Player Controller (Script)
  Script                 PlayerController
  Mouse Look             ✓
  Horz Axis              Horizontal
  Vert Axis              Vertical
  Fire Axis              Fire1
  Max Speed              30
  Reload Delay           0.3
  Can Fire               ✓
  Turret Transforms
    Size                 1
    Element 0            Turret (Transform)
Bounds Lock (Script)
  Script                 BoundsLock
  Horz Range             X -9   Y 11
  Vert Range             X -9   Y 1.82
Health (Script)
  Script                 Health
  Death Particles Prefa  ExplosionDestroy
  Should Destroy On D    ✓
  Health Points          100
```

Configuring the Turret Transforms for spawning ammo

And now you're ready to play-test and fire ammo. By playing the scene and pressing Fire on the keyboard or mouse (left-click), ammo will be generated. Excellent! But on testing this, you may notice two main problems. First, the ammo appears too big or too small. And second, the ammo sometimes bounces, or flips, or reacts to the player spaceship. Let's fix these in turn.

If the ammo appears wrongly sized, you can simply change the scale of the prefab. Select the Ammo prefab in the Project Panel, and from the Object Inspector enter a new scale into the Transform component. See Figure 4.14.


Ammo
Tag: Untagged
Layer: Default

Transform
Position: X 0, Y 0, Z 0
Rotation: X 0, Y 0, Z 0
Scale: X 2, Y 2, Z 2

Mover (Script)
Script: Mover
Max Speed: 7

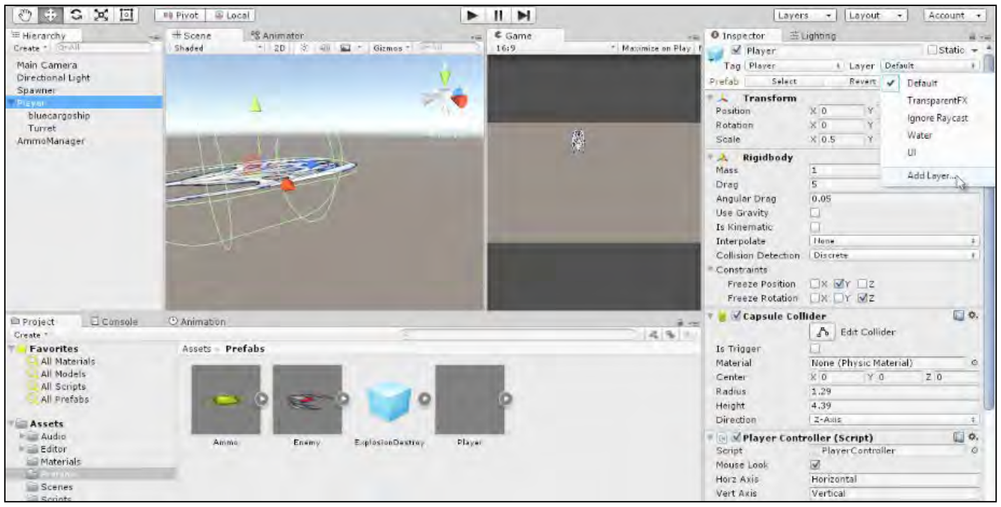
Box Collider
Is Trigger: ☐
Material: None (Physic Material)
Center: X 0, Y 0, Z 0
Size: X 0.1, Y 0.1, Z 0.1

Ammo (Script)
Script: Ammo
Damage: 100
Life Time: 2

Rigidbody
Mass: 1
Drag: 0

Changing the Ammo Prefab Scale

If the ammo appears to bounce or react to the player spaceship, then we'll need to make the ammo 'immune' or 'unresponsive' to the player. To achieve this, we can use Physics Layers. In short, both the Player Space Ship and Ammo should be added to a single Layer, and all objects on this layer should be defined as immune from each other in terms of physical reactions. First, select the Player Object in the scene. Then from the Object Inspector, click the *Layer* Drop Down, and choose *Add Layer* from the context menu. See Figure 4.15.


Player
Tag: Player
Layer: Default

Transform
Position: X 0
Rotation: X 0
Scale: X 0.5

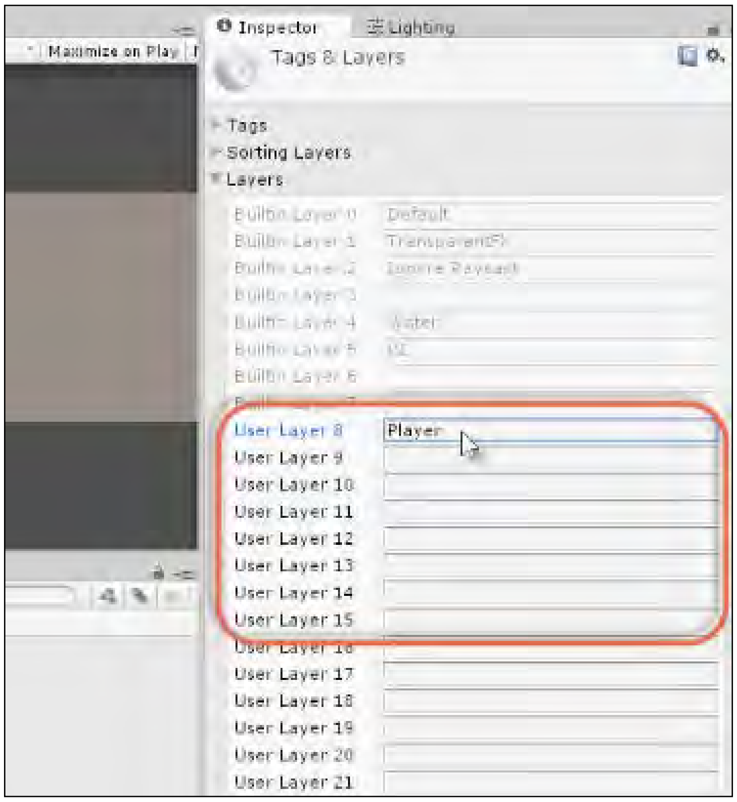
Rigidbody
Mass: 1
Drag: 5
Angular Drag: 0.05
Use Gravity: ☐
Is Kinematic: ☐
Interpolate: None
Collision Detection: Discrete
Constraints
Freeze Position: ☐ X ✓ Y ☐ Z
Freeze Rotation: ☐ X ☐ Y ✓ Z

Capsule Collider
Is Trigger: ☐
Material: None (Physic Material)
Center: X 0, Y 0, Z 0
Radius: 1.29
Height: 4.39
Direction: Z-Axis

Player Controller (Script)
Script: PlayerController
Mouse Look: ✓
Horz Axis: Horizontal
Vert Axis: Vertical

Creating a New Layer for Physics Exclusions

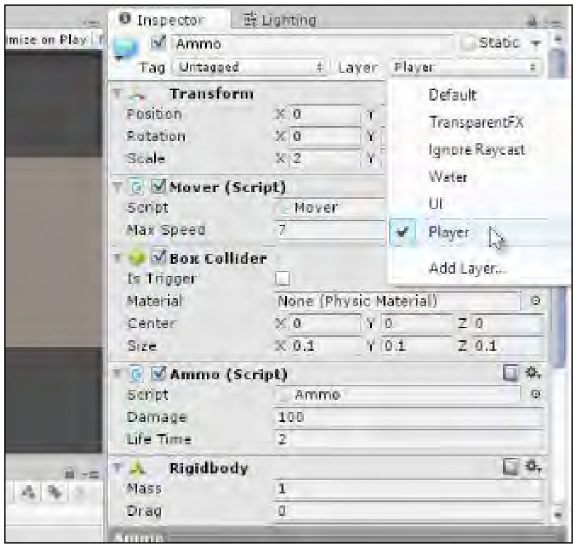
Name the Layer *Player*. This is to indicate that all objects attached to the layer are associated with the Player. See Figure 4.16.



Builtin Layer 0	Default
Builtin Layer 1	TransparentFX
Builtin Layer 2	Ignore Raycast
Builtin Layer 3	
Builtin Layer 4	Water
Builtin Layer 5	UI
Builtin Layer 6	
Builtin Layer 7	
User Layer 8	Player
User Layer 9	
User Layer 10	
User Layer 11	
User Layer 12	
User Layer 13	
User Layer 14	
User Layer 15	
User Layer 16	
User Layer 17	
User Layer 18	
User Layer 19	
User Layer 20	
User Layer 21	

Creating a New Layer for Physics Exclusions

Now assign both the Player object in the scene, and the Ammo Prefab in the Project Panel, to the newly created *Player* layer. Select each, and simply click the *Layer* drop-down, selecting the *Player* option. See Figure 4.17. If prompted with a popup dialog, choose to *Change Children* also. This makes sure all child objects are also associated with the same Layer as the parent.


Inspector | Lighting

Ammo | Static

Tag Untagged | Layer Player

Transform
Position X 0
Rotation X 0
Scale X 2

Mover (Script)
Script Mover
Max Speed 7

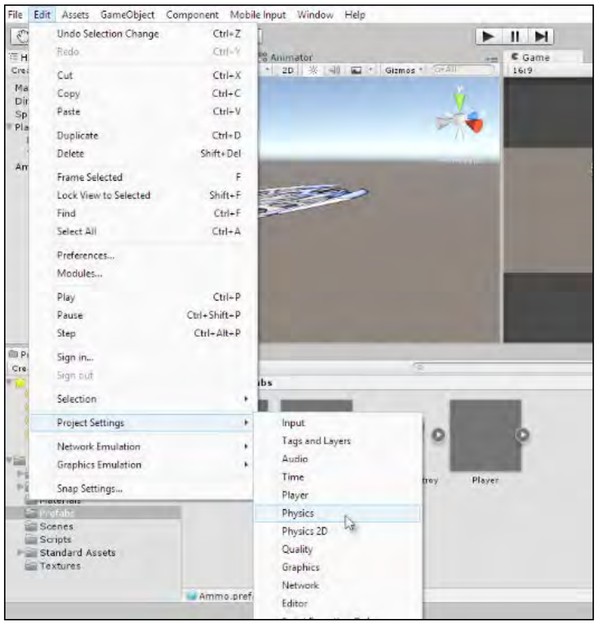
Box Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0 Z 0
Size X 0.1 Y 0.1 Z 0.1

Ammo (Script)
Script Ammo
Damage 100
Life Time 2

Rigidbody
Mass 1
Drag 0

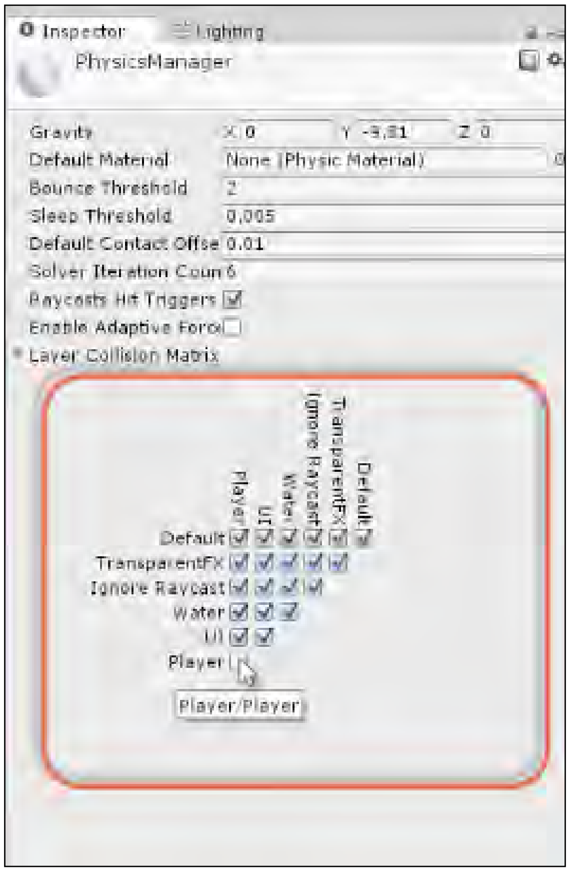
Assigning the Player and Ammo to the Player layer...

Both the Player and Ammo have now been assigned to the same layer. From here we can make all objects in the same layer immune from each other insofar as Physics applies. To do this, select *Edit* > *Project Settings* > *Physics* from the application menu. See Figure 4.18.



Accessing Physics Options

The global Physics Settings appear inside the Object Inspector. At the bottom of the Inspector, the layer collision matrix displays how layers affect each other. Intersecting layers with a check mark can and will affect each other. For this reason, remove the check mark for the Player layer to prevent collisions occurring between objects on this layer. See Figure 4.19.



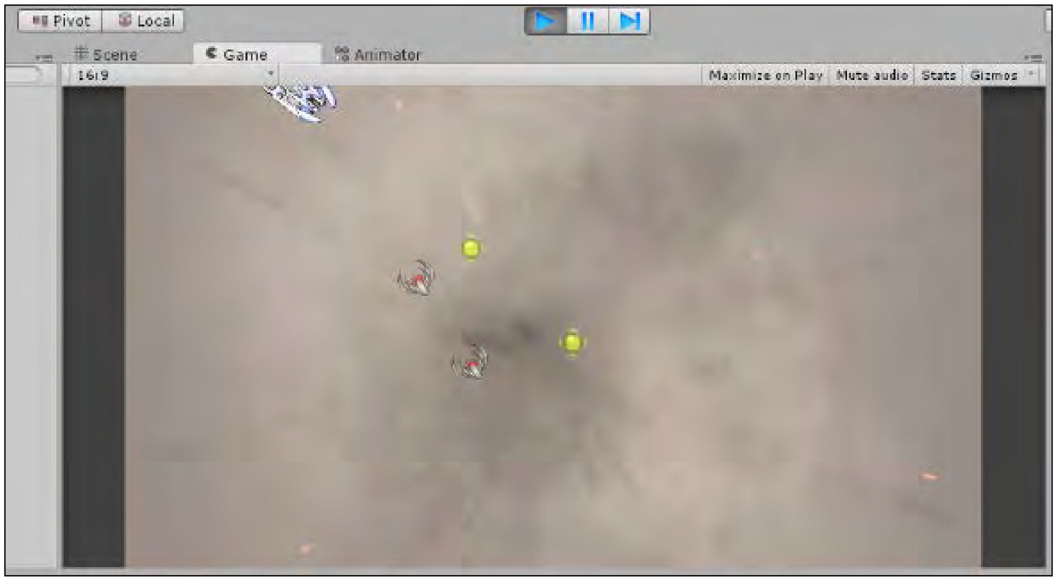
Gravity	X 0 Y -9.81 Z 0
Default Material	None (Physic Material)
Bounce Threshold	2
Sleep Threshold	0.005
Default Contact Offse	0.01
Solver Iteration Coun	6
Raycasts Hit Triggers	✓
Enable Adaptive Forc	☐

Layer Collision Matrix

	Player	UI	Water	Ignore Raycast	TransparentFX	Default
Default	✓	✓	✓	✓	✓	✓
TransparentFX	✓	✓	✓	✓	✓	
Ignore Raycast	✓	✓	✓	✓		
Water	✓	✓	✓			
UI	✓	✓				
Player	☐					

Setting the Layer Collision Matrix for improved collisions

With the Layer Collision Matrix set from the Object Inspector, test run the game so far by pressing *Play* on the toolbar. When you do this, and press fire, ammo will issue from the turrets and no longer react to the player spaceship. The ammo should however collide with, and destroy, the enemies. See Figure 4.20.



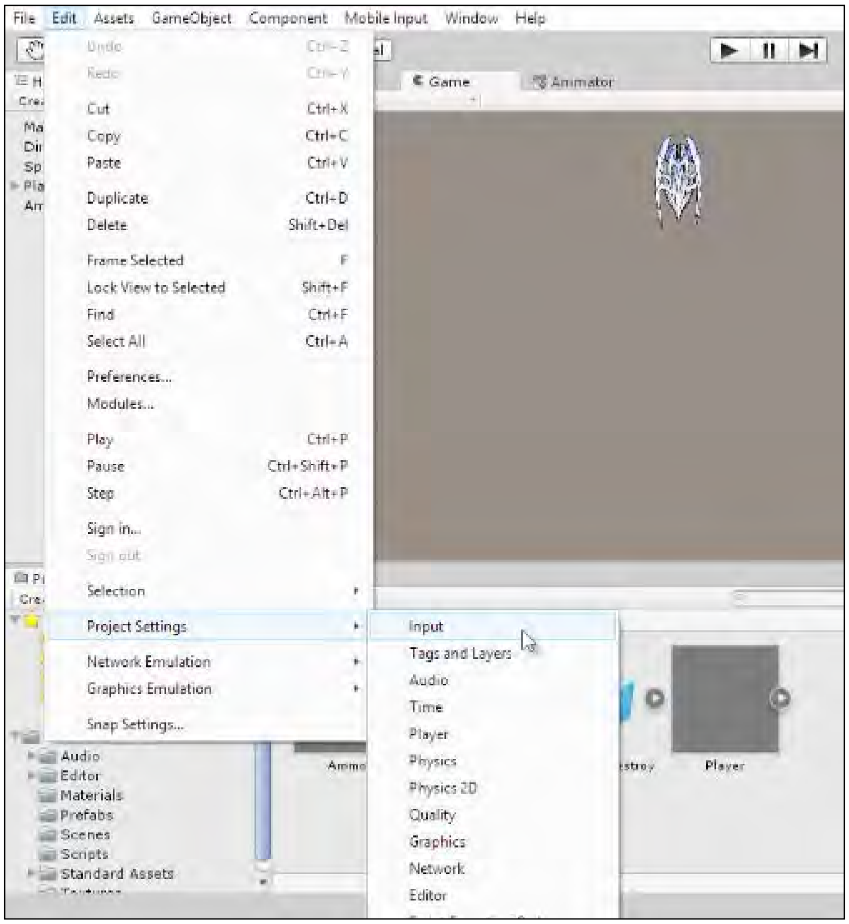
Destroying enemies by shooting guns!

Excellent work! We now have a spaceship that can fire weapons and destroy enemies; and the physics works as expected. But maybe you'd like to customize player controls a little- or perhaps you want to use a gamepad. The next section will explore this issue further.

User Controls

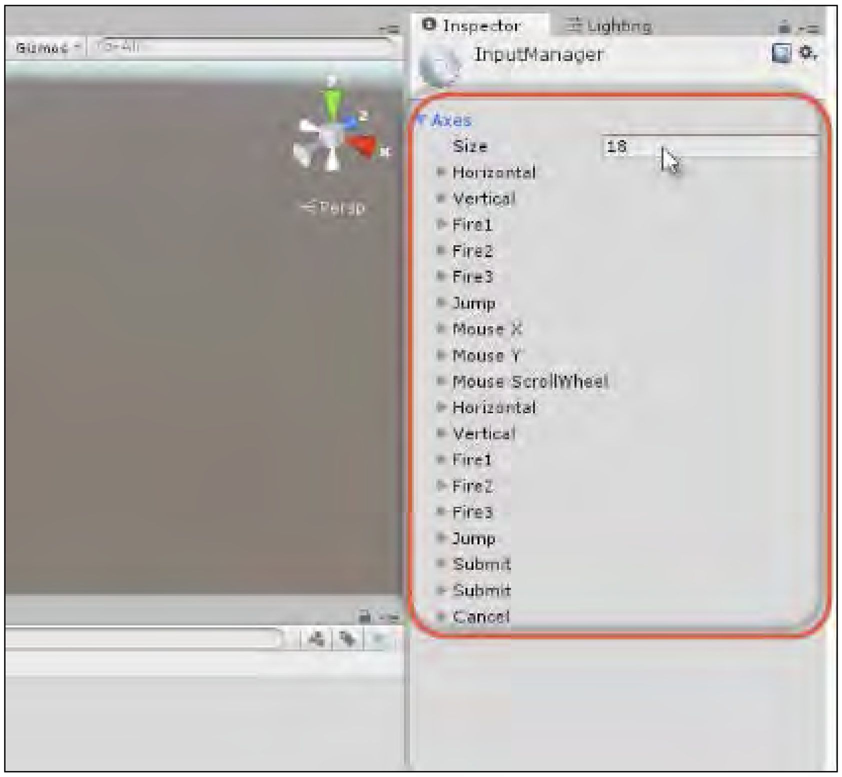
Maybe you don't like the default controls and the key combinations associated to the input axes "Horizontal", "Vertical" and "Fire1". Maybe you want to change them. These input axis are read using the *Input.GetAxis* function (shown earlier) and are specified by human readable names, but it's not immediately clear how unity maps specific input buttons and devices to these virtual axes. Here, we'll see briefly how to customize these.

To get started, let's access the Input settings, by selecting Edit > Project Settings > Input from the application menu. See Figure 4.21.



Accessing the Input Menu...

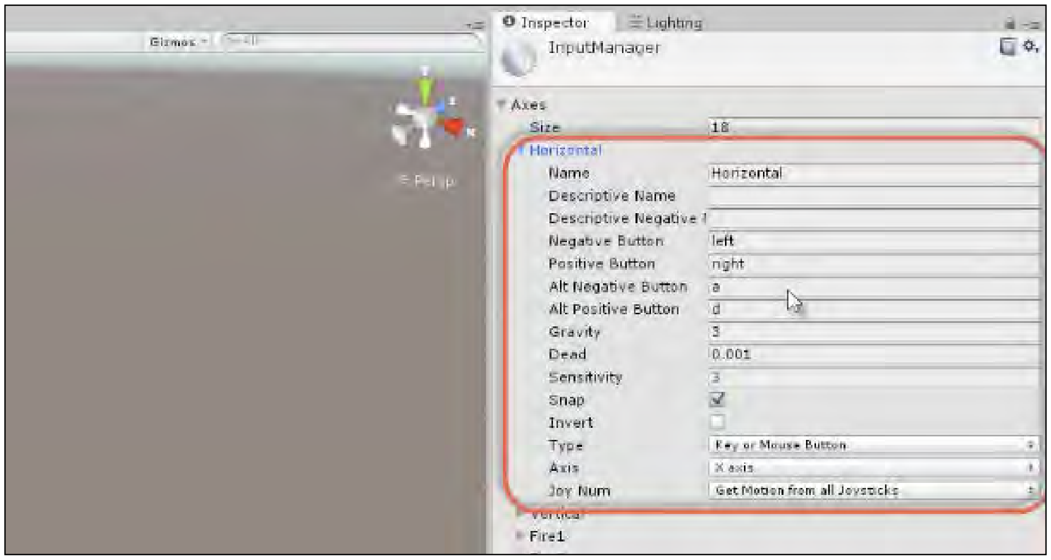
On selecting this option, a collection of custom-defined input axes appear as a list in the Object Inspector. See Figure 4.22. This defines all axes used by the input system. The axes 'Horizontal' and 'Vertical' should be listed here.



Axes
Size 18
Horizontal
Vertical
Fire1
Fire2
Fire3
Jump
Mouse X
Mouse Y
Mouse ScrollWheel
Horizontal
Vertical
Fire1
Fire2
Fire3
Jump
Submit
Submit
Cancel

Exploring the input axes

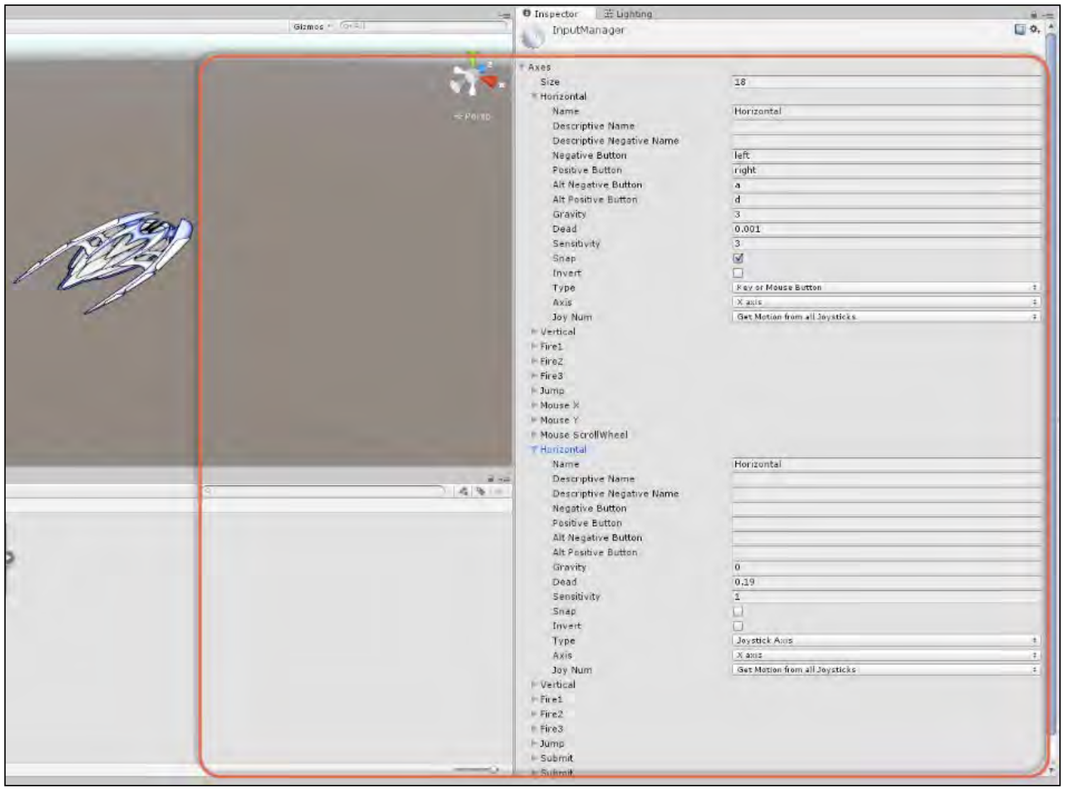
By expanding each axis in the Object Inspector you can easily customize how user input is mapped. That is, how specific keys and controls on hardware devices, like a keyboard and a mouse, will map to an axis. The Horizontal Axis, for example, is defined twice. For the first definition, *Horizontal* is mapped to the *left*, *right*, and *A* and *D* keys on the keyboard. *Right* and *D* are mapped as *Positive* buttons because, when pressed, they produce positive floating-point values from the *Input.GetAxis* function (between 0-1). *Left* and *A* are mapped as *Negative* buttons because, when pressed, they result in negative floating-points values for *Input.GetAxis*. This makes it easy to move objects left and right; by using negative and positive numbers. See Figure 4.23.



Axes	
Size	18
Horizontal	
Name	Horizontal
Descriptive Name	
Descriptive Negative	
Negative Button	left
Positive Button	right
Alt Negative Button	a
Alt Positive Button	d
Gravity	3
Dead	0.001
Sensitivity	3
Snap	✓
Invert	
Type	Key or Mouse Button
Axis	X axis
Joy Num	Get Motion from all Joysticks

Configuring an Input Axis

Notice that *Horizontal* is defined twice in the Object Inspector, once near the top of the list and once near the bottom. These two definitions are accumulative, and not contradictory- they stack atop one another. They allow you to map *multiple* devices to the *same* axis, giving you cross-platform and multi-device control over your games. By default, *Horizontal* is mapped in the first definition to the *left*, *right*, *A* and *D* keys on the keyboard; and in the second definition to joystick motion. Both definitions are valid and work together. You can have as many definitions for the same axis as you need; depending on the controls you need to support. See Figure 4.24.



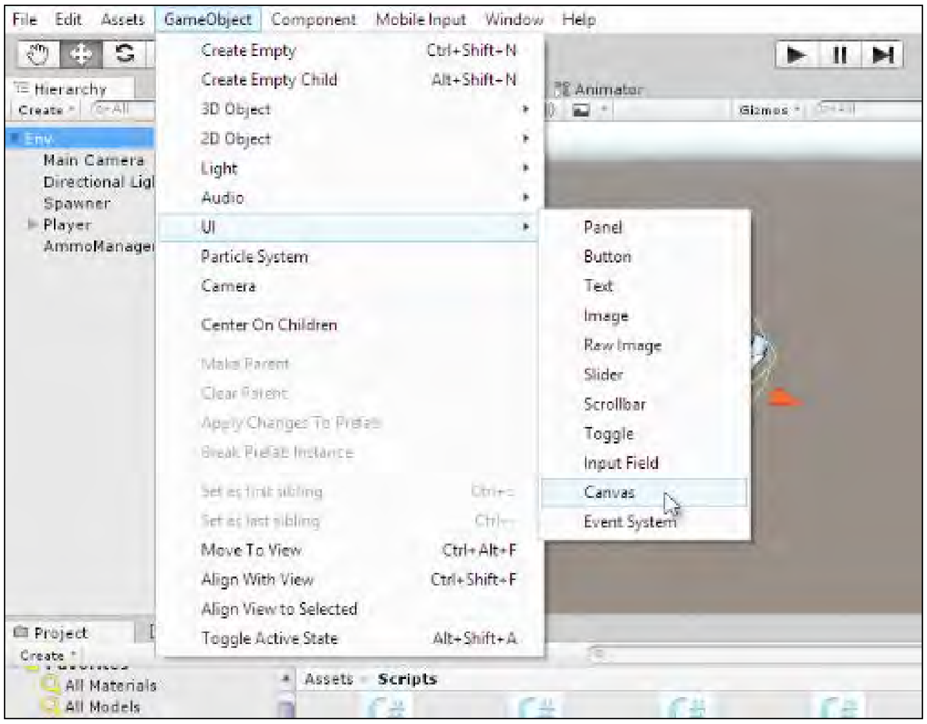
```
Inspector   Lighting
InputManager
▼ Axes
   Size                        18
  ▼ Horizontal
     Name                      Horizontal
     Descriptive Name
     Descriptive Negative Name
     Negative Button           left
     Positive Button           right
     Alt Negative Button       a
     Alt Positive Button       d
     Gravity                   3
     Dead                      0.001
     Sensitivity               3
     Snap                      ✓
     Invert                    ☐
     Type                      Key or Mouse Button
     Axis                      X axis
     Joy Num                   Get Motion from all Joysticks
  ► Vertical
  ► Fire1
  ► Fire2
  ► Fire3
  ► Jump
  ► Mouse X
  ► Mouse Y
  ► Mouse ScrollWheel
  ▼ Horizontal
     Name                      Horizontal
     Descriptive Name
     Descriptive Negative Name
     Negative Button
     Positive Button
     Alt Negative Button
     Alt Positive Button
     Gravity                   0
     Dead                      0.19
     Sensitivity               1
     Snap                      ☐
     Invert                    ☐
     Type                      Joystick Axis
     Axis                      X axis
     Joy Num                   Get Motion from all Joysticks
  ► Vertical
  ► Fire1
  ► Fire2
  ► Fire3
  ► Jump
  ► Submit
  ► Submit
```

Defining two Horizontal Axes

For this project the controls will remain at their defaults, but go ahead and change or add additional controls if you want to support different configurations. More information on Player Input and customizing controls can be found at the online Unity documentation here: `http://docs.unity3d.com/Manual/class-InputManager.html`

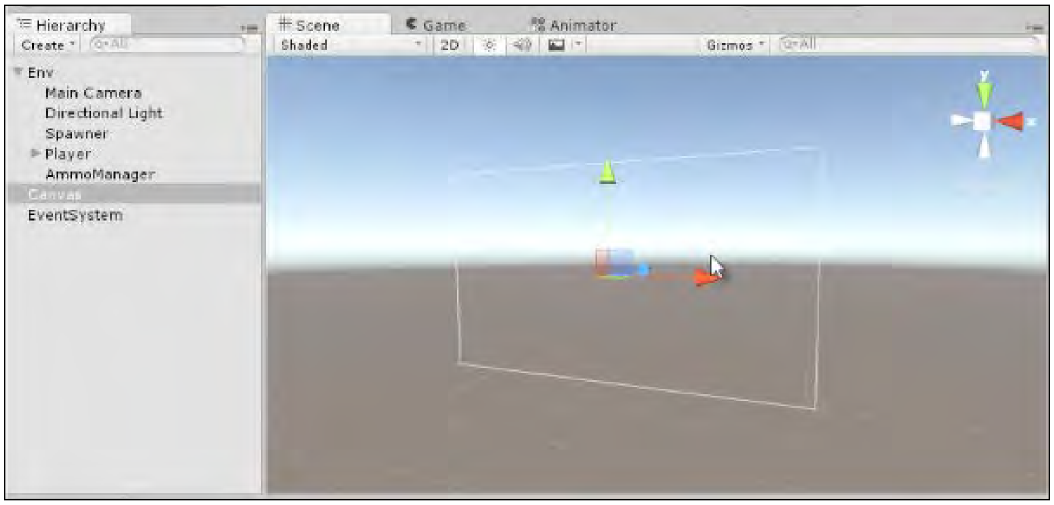
Scores and Scoring – UI and Text Objects

Let's move onto to the scoring system and, in creating this, we'll create a *GameController*. The *GameController* is simply a script or class that manages all game-wide and overarching behavior. This includes the *Score* because, for this game, the score refers to one, single and global number representing the achievements and progress of the player. Before jumping into implementation, start by creating a simple GUI for displaying the game score. GUI is an acronym for Graphic User Interface, and this refers to all the 2D graphical elements that sit atop the game window and provide information to the player. To create this, create a new GUI canvas object by selecting *GameObject* > *UI* > *Canvas* from the application menu. See Figure 4.25.



Adding a Canvas Object to the Scene...

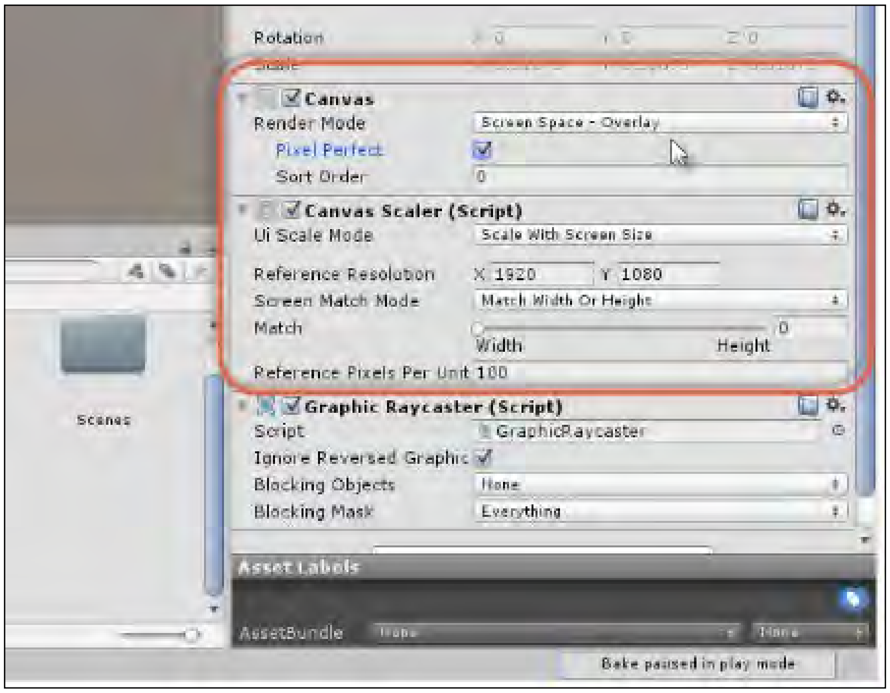
The Canvas object defines the total surface or area inside which the GUI lives, including all buttons, text and other widgets. On being generated in the scene, the Canvas also features in the Hierarchy panel. Initially, the Canvas may be too large or too small to be seen clearly inside the viewport; so select the Canvas object in the hierarchy panel and press the *F* key on the keyboard to focus the object. It should appear as a large vertically aligned rectangle. See Figure 4.26.



Examining the Canvas Object in the Viewport

The Canvas object is not visible itself inside the Game tab. Rather, it acts simply as a container. Even so, it strongly influences how contained objects appear on screen, in terms of size, position and scale. For this reason, before adding objects and refining the design of an interface, it's helpful to configure your canvas object first.

To do this, select the Canvas Object in the scene, and from the Object Inspector, click the *UI Scale Mode* drop-down option from the *Canvas Scaler* Component. From the drop-down list, choose the option *Scale with Screen Size*, and enter a HD resolution into the *Reference Resolution* Field. That is, specify 1920 for the X field, and 1080 for the Y Field. See Figure 4.27.



Rotation X 0 Y 0 Z 0

Canvas
Render Mode: Screen Space - Overlay
Pixel Perfect: ✓
Sort Order: 0

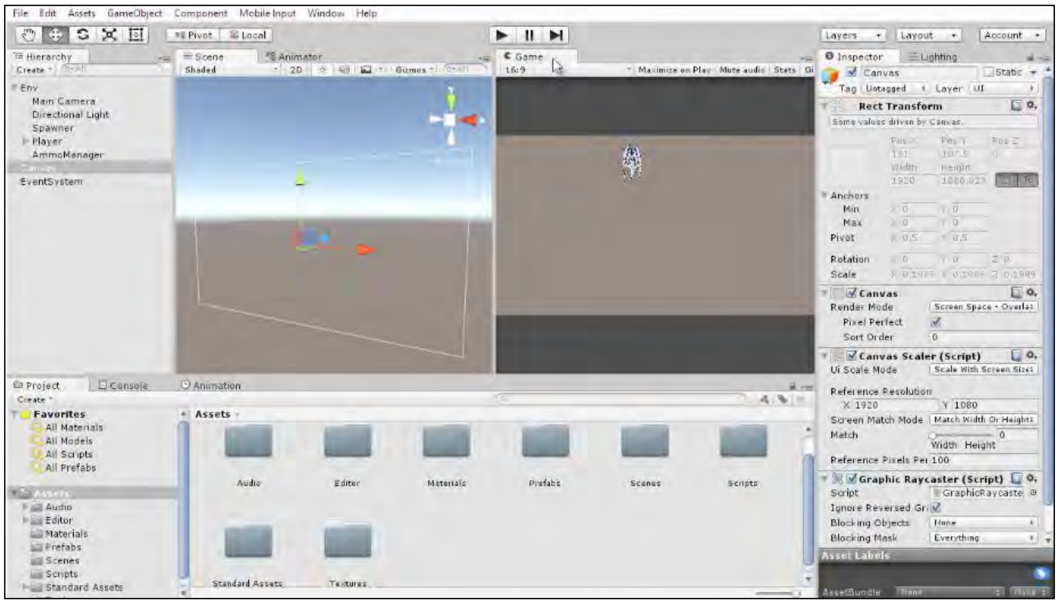
Canvas Scaler (Script)
Ui Scale Mode: Scale With Screen Size
Reference Resolution: X 1920 Y 1080
Screen Match Mode: Match Width Or Height
Match: 0 (Width — Height)
Reference Pixels Per Unit: 100

Graphic Raycaster (Script)
Script: GraphicRaycaster
Ignore Reversed Graphic: ✓
Blocking Objects: None
Blocking Mask: Everything

AssetBundle: None None

Adjusting the Canvas Scaler Component

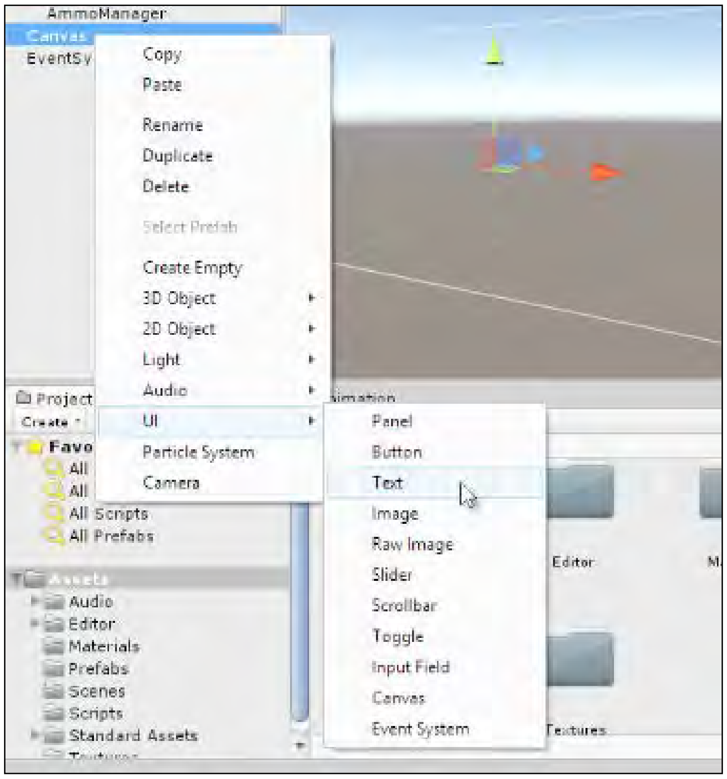
By adjusting the canvas scaler to *Scale with Screen Size*, the User Interface for the game will automatically stretch and shrink (up and down-scale) to fit the target resolution, ensuring each element is scaled to the same proportions, maintaining the overall look and feel. This is a quick and easy method for creating a UI once and to have it adjust size to fit nearly any resolution. It may not always be the best solution to maintaining the highest quality graphical fidelity, but it's functional and suitable in many cases. In any case, before proceeding with UI design, it's helpful to see both the Scene Viewport and Game Tab side by side in the interface (or across two monitors, if you have a multi-monitor configuration). This allows us to build the interface inside the scene viewport, and then to preview its effects in the Game Tab. You can rearrange the Scene and Game tabs simply by dragging and dropping the Game Tab beside the Scene Tab in the Unity Editor. See Figure 4.28.



Canvas
Tag Untagged Layer UI
Rect Transform
Some values driven by Canvas.
Width 1920 Height 1080.023
Canvas
Render Mode: Screen Space - Overlay
Pixel Perfect: ✓
Sort Order: 0
Canvas Scaler (Script)
Ui Scale Mode: Scale With Screen Size
Reference Resolution: X 1920 Y 1080
Screen Match Mode: Match Width Or Height
Match: 0
Reference Pixels Per: 100
Graphic Raycaster (Script)
Script: GraphicRaycaster
Ignore Reversed Gr: ✓
Blocking Objects: None
Blocking Mask: Everything

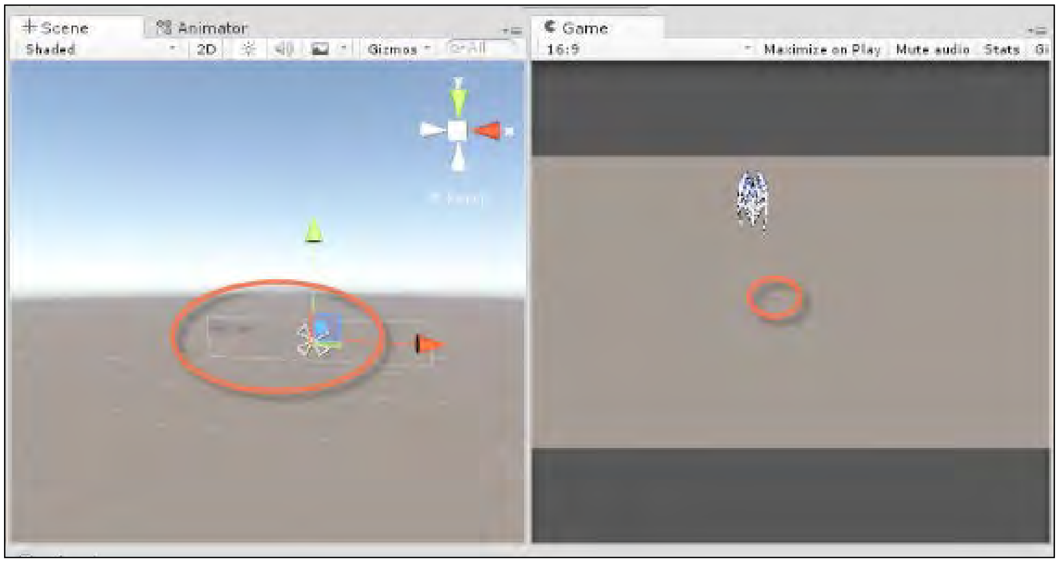
Docking the Scene and Game tabs side by side...

Next, let's add the text widget to the GUI for displaying the game score. To do this, select the Canvas object in the hierarchy panel, and then right-click that object (in the hierarchy panel) to display a context menu. From here, select *UI* > *Text*. This creates a new text object as a child of the *Canvas* object, as opposed to a top-level object with no parent. See Figure 4.29. The text object is useful for drawing text on-screen with a specific color, size and font setting.



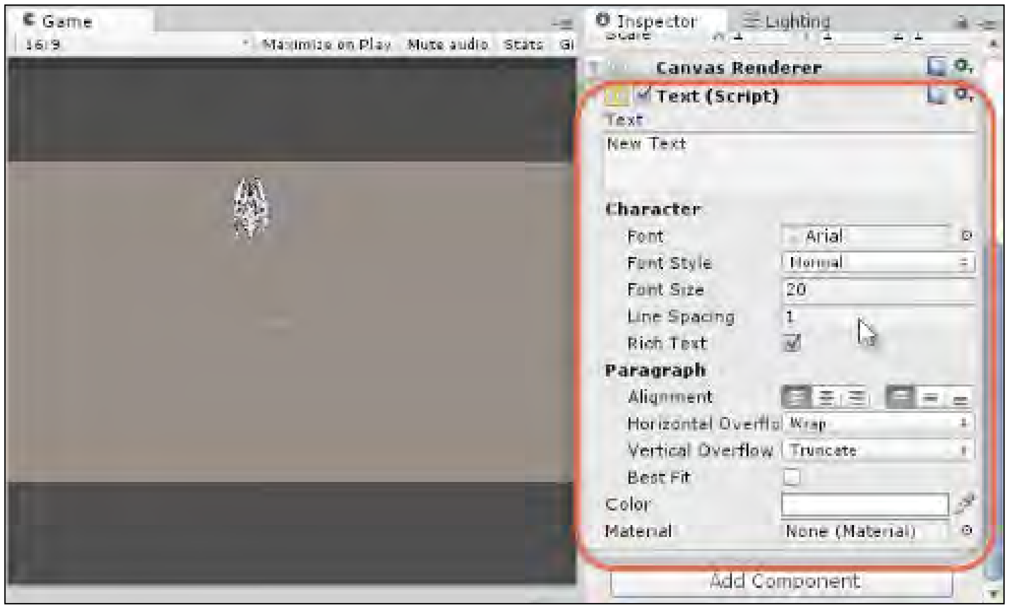
Creating a text object for the UI

By default, the text object may not initially appear visible in either the scene or the viewport, even though it's listed as an object in the Hierarchy Panel. However, look more closely in the scene and you're likely to see very small and dark text, which appears both inside the Canvas and in the Game tab. See Figure 4.30. By default, new text objects feature black text at a small font size. For this project, these settings will need to be changed.



Newly created text objects can sometimes be difficult to see...

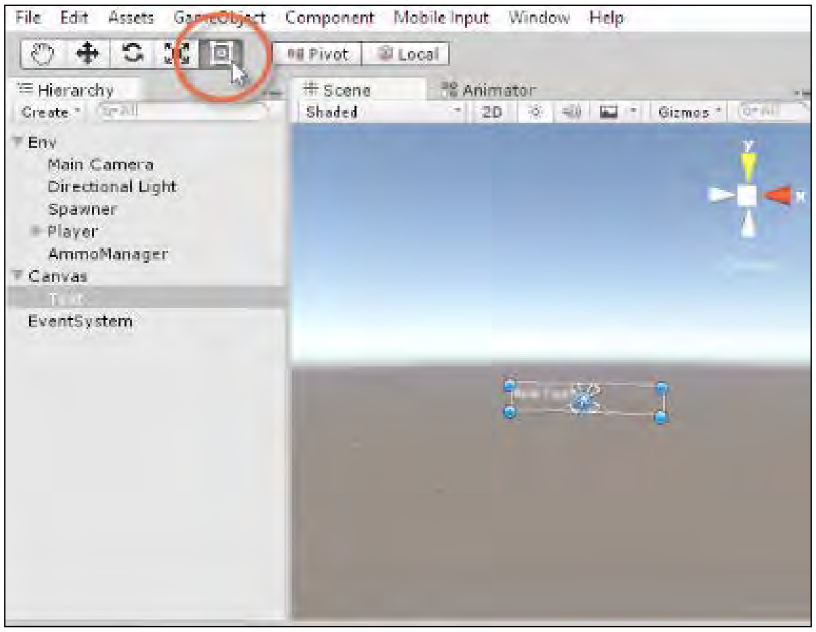
Select the text object in the hierarchy panel, if it's not already selected, and from the Object Inspector (in the Text Component), change the text color to white, and the size to 20. See Figure 4.31.



Text (Script)
Text
New Text
Character
Font Arial
Font Style Normal
Font Size 20
Line Spacing 1
Rich Text ✓
Paragraph
Alignment
Horizontal Overflo Wrap
Vertical Overflow Truncate
Best Fit
Color
Material None (Material)

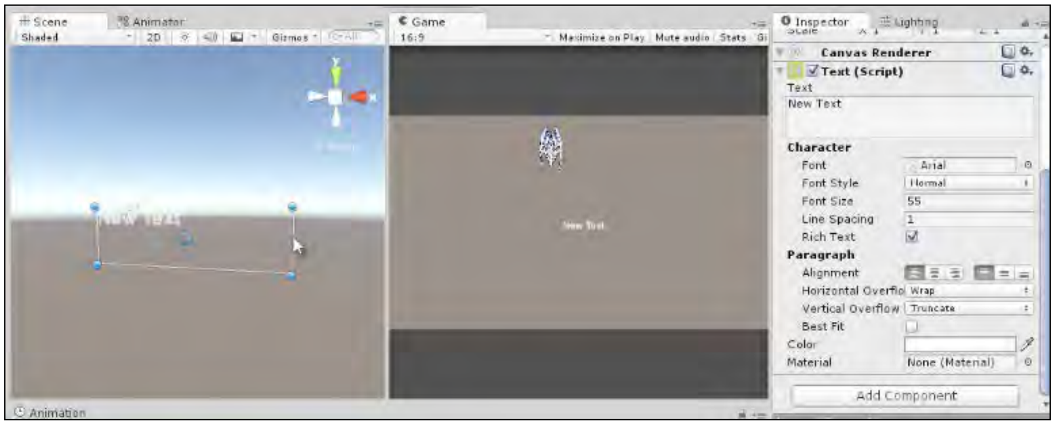
Changing text size and color

The text however still appears too small, even after changing its size. If you increase the size further, however, the text may disappear from view. This happens because each text object has a rectangular boundary, defining its limits, and when the font size increases beyond what can fit inside the boundary, the text automatically hides altogether. To fix this, we'll increase the text boundary. To do that, switch to the *Rect Transform* tool with *T*, or select the tool from the Tool bar. See Figure 4.32.



Selecting the Rect Transform Tool

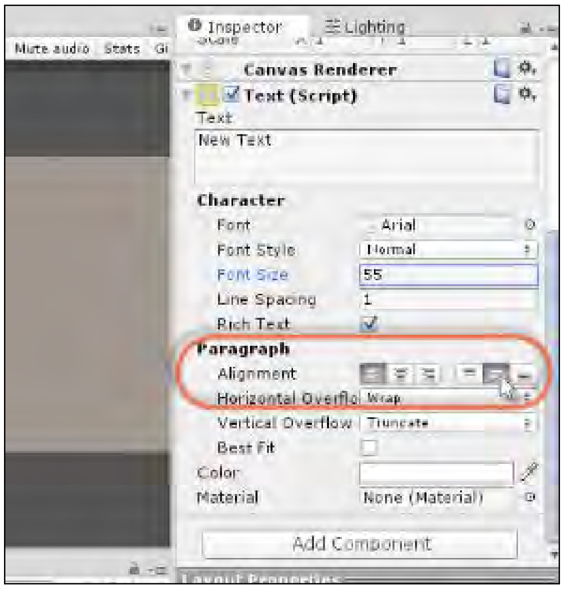
On activating the Rect Transform Tool, a clearly defined boundary will be drawn around the selected text object in the Scene viewport, indicating its rectangular extents. Let's increase the boundary size to accommodate larger text. To do this, simply click and drag on the boundary edges with the mouse to extend them as needed. See Figure 4.33. This will increase the boundary size, and now you can increase the font size to improve text readability.



Canvas Renderer
Text (Script)
Text: New Text
Character
Font: Arial
Font Style: Normal
Font Size: 55
Line Spacing: 1
Rich Text: ✓
Paragraph
Alignment
Horizontal Overflow: Wrap
Vertical Overflow: Truncate
Best Fit: ☐
Color
Material: None (Material)

Adjust the Text Rectangle to support larger font sizes

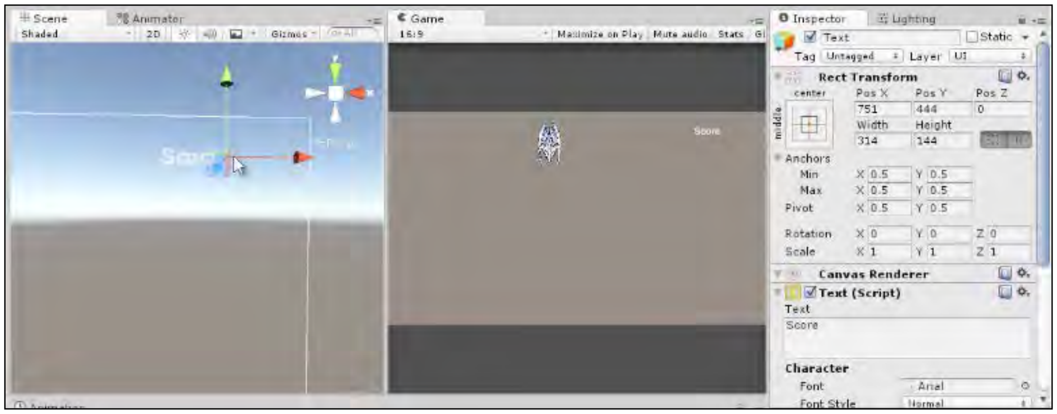
In addition to setting the text boundary size, the text can also be vertically aligned to the boundary center. Simply click the center alignment button for the vertical group. For horizontal alignment, the text should remain left-aligned, to allow for the score display. See Figure 4.34.



Canvas Renderer
Text (Script)
Text: New Text
Character
Font: Arial
Font Style: Normal
Font Size: 55
Line Spacing: 1
Rich Text: ✓
Paragraph
Alignment
Horizontal Overflow: Wrap
Vertical Overflow: Truncate
Best Fit: ☐
Color
Material: None (Material)

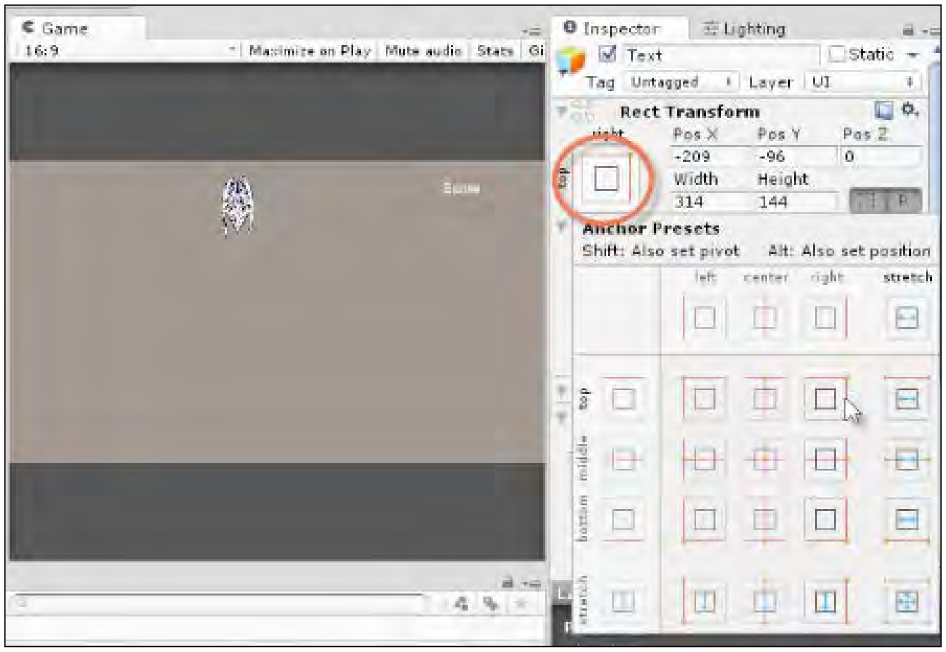
Aligning Text within the Boundary

Although the text is now aligned vertically within its containing boundary, we'll still need to align it as a whole to the Canvas container, to ensure it remains on-screen at the same position and orientation, even if the Game window is resized and realigned. To do this, we'll use *Anchors*. To start, use the Transform tool (*W*) to reposition the text object to the top-right corner of the screen, at the location where the score should appear. The object will automatically move within a 2D plane, as opposed to within 3D space. As you move the text object in the Scene Viewport, check its appearance in the Game tab to ensure it looks correct and appropriate. See Figure 4.35.



Positioning the Score Text within the Game Tab

To secure the position of the text object on-screen (preventing it from sliding or moving), even if the Game tab is resized by the user, we can set the Object's anchor position to the top-right corner of the screen. This ensures the text is always positioned as a constant, proportional offset from its anchor. To do this, click the Anchor preset button, inside the *RectTransform* component, in the Object Inspector. When you do this, a preset menu appears, from which you can choose a range of alignment locations. Each present is graphically presented as a small diagram, including a red dot at the location of anchor alignment. Select the top-right preset. See Figure 4.36.



Aligning the text object to the screen...

Excellent work! The text object is now created and ready to use. Of course, in play mode, the text remains unchanged and doesn't display a real score. That's because we need to add some code. But, overall, the text object is in place and we can move on…

Working with Scores – Scripting with Text

To display a score in the GUI, we'll first need score functionality; that is, code to create a score system. Essentially, the score functionality will be added to a general, overarching *GameController* class, responsible for all game-wide logic and features. The code for the GameController and its score feature set is included in Code Sample 4-4, as below. This file should be added to the Scripts folder of the project.

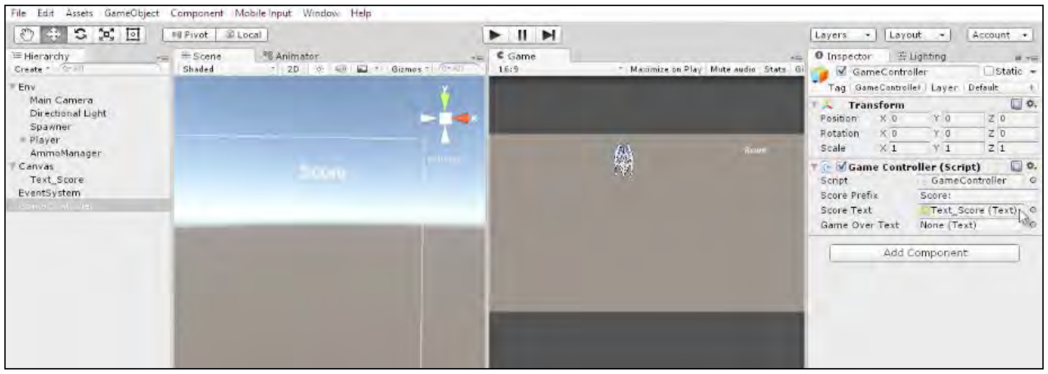
```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
//------------------------------
public class GameController : MonoBehaviour
{
   //Game score
   public static int Score;
```

```
        //Prefix
        public string ScorePrefix = string.Empty;

        //Score text object
        public Text ScoreText = null;

        //Game over text
        public Text GameOverText = null;

        public static GameController ThisInstance = null;
        //------------------------------
        void Awake()
        {
               ThisInstance = this;
        }
        //------------------------------
        void Update()
        {
               //Update score text
               if(ScoreText!=null)
                      ScoreText.text = ScorePrefix + Score.ToString();
        }
        //------------------------------
        public static void GameOver()
        {
               if(ThisInstance.GameOverText!=null)
                      ThisInstance.GameOverText.gameObject.SetActive(true);
        }
        //------------------------------
}
```

Comments on Code Sample 4-4

- The GameController class uses the namespace `UnityEngine.ui`. This is important because it includes access to all the UI classes and objects within Unity. If you don't include this namespace inside your source files, then you cannot use UI objects from that script.
- The GameController class features two `Text` public members, namely `ScoreText` and `GameOverText`. These refer to two text objects, both of which are optional insofar as the GameController code will work just fine, even if the members are null. `ScoreText` is a reference to a text GUI object for displaying score text; and the `GameOverText` is for displaying any message when a game over condition occurs.

To use the *GameController* code, create a new, empty object in the scene, named *GameController*. Then drag and drop the *GameController* Script file onto that object. Once added, drag and drop the 'Score Text' object into the *Score Text* field for the *GameController*, in the Object Inspector. See Figure 4.37. In the *Score Prefix* field, enter the text that should prefix the score itself. The score on its own is simply a number (such as 1000). The prefix allows you to add text to the front of that score, indicating to the player what the numbers mean.



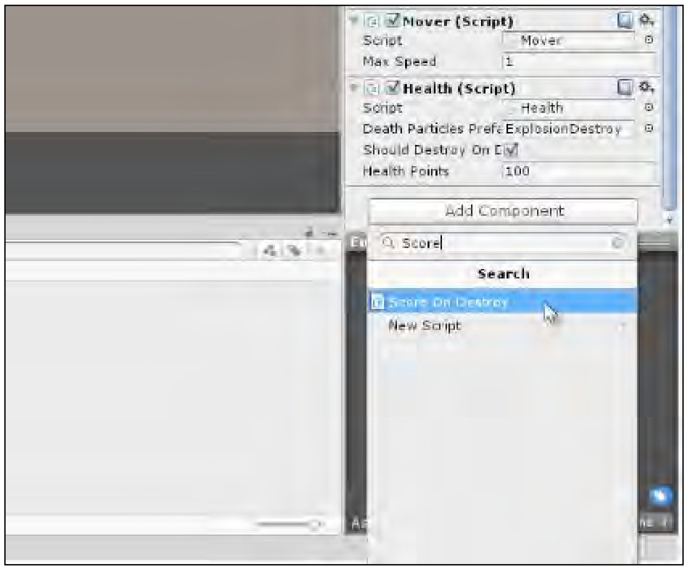
GameController
Static
Tag GameController Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Game Controller (Script)
Script GameController
Score Prefix Score:
Score Text Text_Score (Text)
Game Over Text None (Text)

Creating a GameController for maintaining Game Score...

Now take the game for a test run, and you'll see the score display at the top-right corner of the Game tab, using the GUI text object. This is fine, but the score always remains at 0 right now. This is because we have no code, yet, to increase the score. For our game, the score should increase when an Enemy object is destroyed. To achieve that, we'll create a new Script file *ScoreOnDestroy*. This is included in Code Sample 4-5, as below.

```
using UnityEngine;
using System.Collections;
//-----------------------------
public class ScoreOnDestroy : MonoBehaviour
{
   //-----------------------------
   public int ScoreValue = 50;
   //-----------------------------
   void OnDestroy()
   {
         GameController.Score += ScoreValue;
   }
   //-----------------------------
}
//-----------------------------
```

The script should be attached to any object that assigns you points when it's destroyed, such as the enemies. The total number of points assigned is specified by `ScoreValue`. To attach the script to the enemy prefab, select the Prefab in the Project Panel, and from the Object Inspector click the button *Add Component*. Then type *ScoreOnDestroy* into the search field to add the component to the prefab. Once added, specify the total number of points to be allocated for destroying an enemy. For this game, a value of *50* points is assigned. See Figure 4.38.



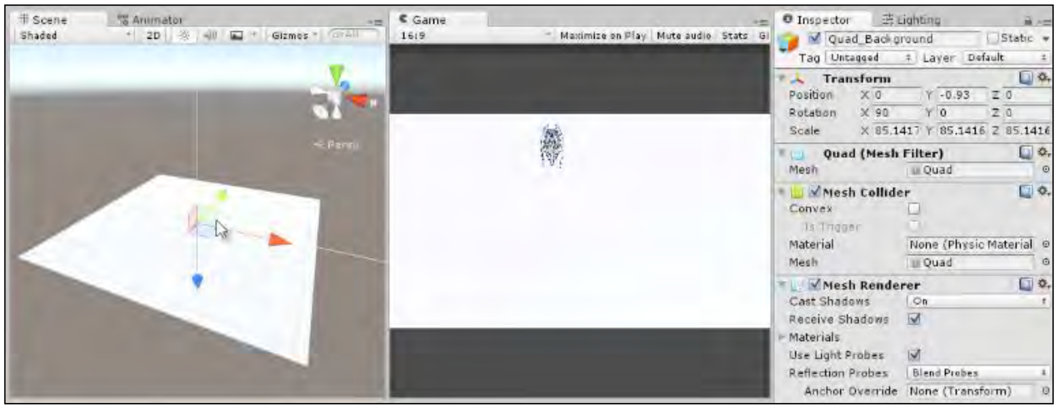
Mover (Script)
Script Mover
Max Speed 1
Health (Script)
Script Health
Death Particles Prefa ExplosionDestroy
Should Destroy On D ✓
Health Points 100

Adding a Score component to the Enemy prefab

Great work! You now have destroyable enemies that assign you points on destruction. This means you can finally have an in-game score, and could even extend gameplay to include high-score features and leaderboards. This also means our game is almost finished and ready to build. Next, we'll add some final touches.

Polishing

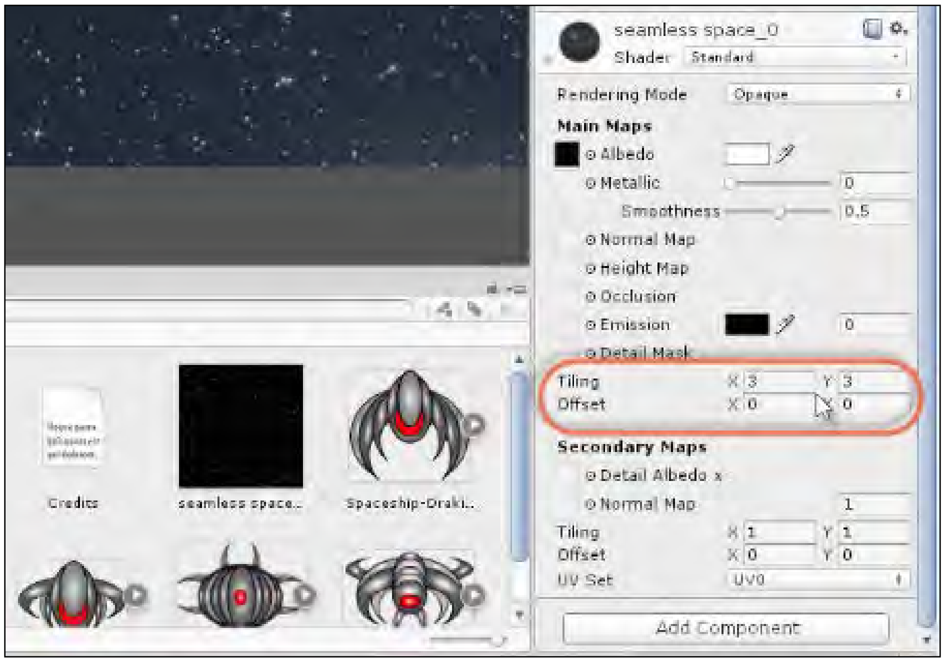
In this section we'll add the final touches to the game. First on the agenda is to fix the game background! Until now, the background has simply displayed the default background color associated with the game camera. But, since the game is set in space, we should display a space background. To do this, create a new Quad object in the scene that'll display a space image. *Choose GameObject* > *3D Object* > *Quad* from the menu. Then rotate the object and move it downwards so it displays a flat, vertically aligned backdrop. You may need to scale the object to look correct. See Figure 4.39.



Quad_Background Static
Tag Untagged Layer Default
Transform
Position X 0 Y -0.93 Z 0
Rotation X 90 Y 0 Z 0
Scale X 85.1417 Y 85.1416 Z 85.1416
Quad (Mesh Filter)
Mesh Quad
Mesh Collider
Convex
Is Trigger
Material None (Physic Material
Mesh Quad
Mesh Renderer
Cast Shadows On
Receive Shadows ✓
Materials
Use Light Probes ✓
Reflection Probes Blend Probes
Anchor Override None (Transform)

Creating a backdrop for the level; building a Quad

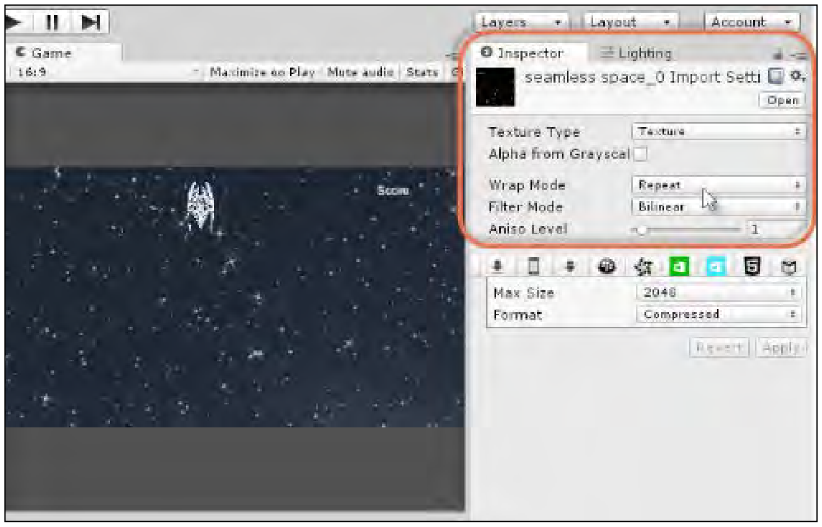
Now drag and drop the space texture from the Project Panel onto the Quad in the scene, to apply it as a material. Once assigned, select the Quad, and change the tiling settings from the material properties in the Object Inspector. Increase the X and Y tiling to 3. See Figure 4.40.



seamless space_0
Shader Standard
Rendering Mode Opaque
Main Maps
Albedo
Metallic 0
Smoothness 0.5
Normal Map
Height Map
Occlusion
Emission 0
Detail Mask
Tiling X 3 Y 3
Offset X 0 Y 0
Secondary Maps
Detail Albedo x
Normal Map 1
Tiling X 1 Y 1
Offset X 0 Y 0
UV Set UV0

Configuring Texture Tiling

If texture tiling seems broken for you, then be sure to check the Texture Importing settings. To do that, select the texture in the Project Panel, and from the Object Inspector, ensure the Texture Type is set to *Texture*, and the Wrap mode is set to *Repeat*. See Figure 4.41.

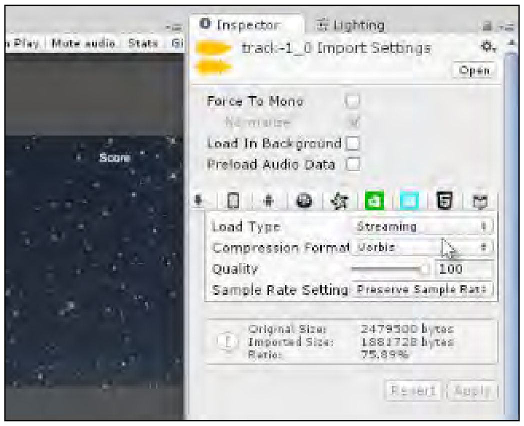


seamless space_0 Import Setti

Texture Type	Texture
Alpha from Grayscal	☐
Wrap Mode	Repeat
Filter Mode	Bilinear
Aniso Level	1
Max Size	2048
Format	Compressed

Configuring a texture for seamless tiling

Now the level has a suitable background. Let's add some background music, which will play on a loop. To do this, first select the music track in the Project Panel, inside the *Audio* Folder. When selected, make sure the music *Load Type*, from the Object Inspector, is set to *Streaming*, and further that *Preload Audio Data* is disabled. See Figure 4.42. This improves loading times, as Unity will not need to load all music data into memory as the scene begins.



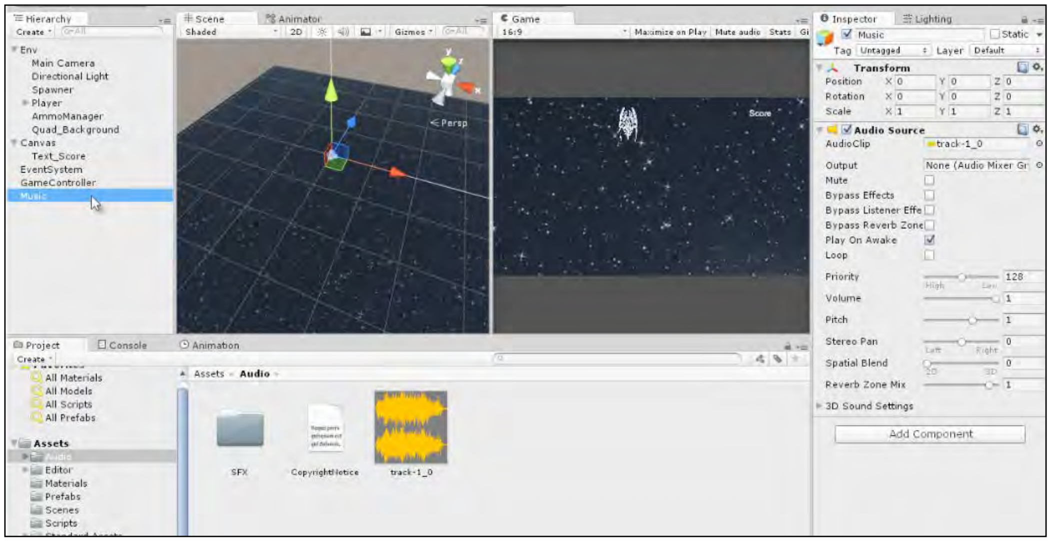
track-1_0 Import Settings

Force To Mono	☐
Normalize	✓
Load In Background	☐
Preload Audio Data	☐
Load Type	Streaming
Compression Format	Vorbis
Quality	100
Sample Rate Setting	Preserve Sample Rate

```
Original Size:    2479500 bytes
Imported Size:    1881728 bytes
Ratio:            75.89%
```

Configuring Audio Data ready for Playback

Next, create a new, empty GameObject in the scene named *Music*, and then drag and drop the music track from the Project Panel onto the Music object, adding it as an *Audio Source* Component. Audio Source components playback sound effects and music. See Figure 4.43.



Music	✓	Static	☐
Tag	Untagged	Layer	Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

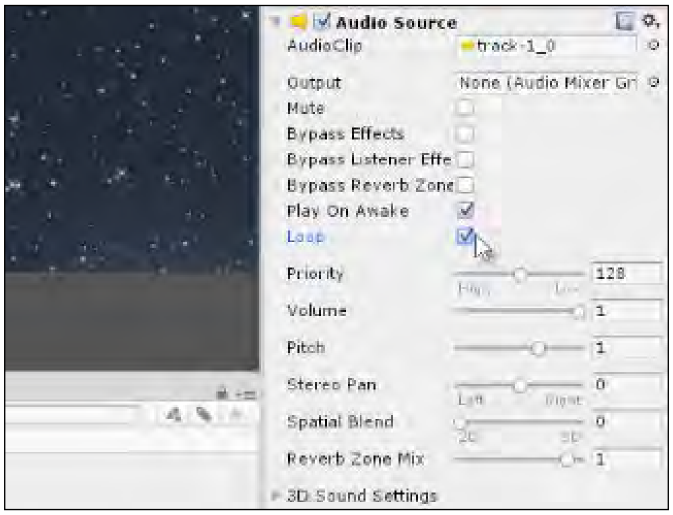
Audio Source

AudioClip	track-1_0
Output	None (Audio Mixer Gr
Mute	☐
Bypass Effects	☐
Bypass Listener Effe	☐
Bypass Reverb Zone	☐
Play On Awake	✓
Loop	☐
Priority (High – Low)	128
Volume	1
Pitch	1
Stereo Pan (Left – Right)	0
Spatial Blend (2D – 3D)	0
Reverb Zone Mix	1
3D Sound Settings	

Creating a GameObject with an AudioSource component

From the Audio Source Component in the Object Inspector, enable the check boxes *Play on Awake*, and *Loop* to ensure the music is played from the level beginning and loops endlessly for as long as the game is running. The Spatial Blend field should be set to 0, meaning 2D. In short, 2D Sounds have a consistent volume throughout the level regardless of the player position. This is because 2D sounds are not spatially located.

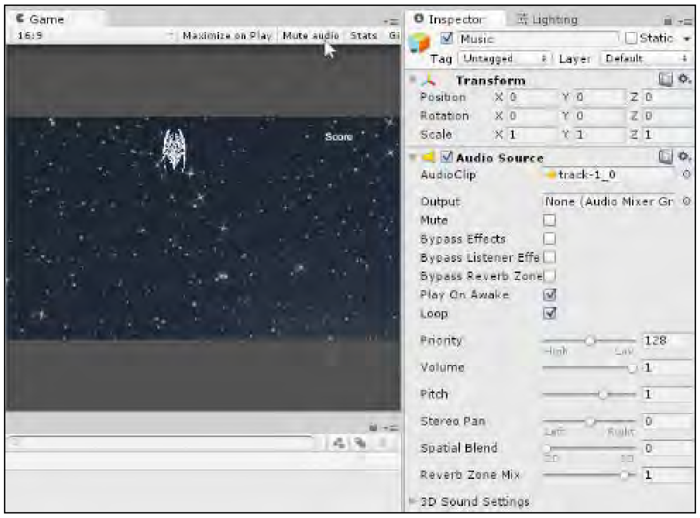
3D Sounds, in contrast, are used for gunshots, footsteps, explosions and other sounds that exist in 3D space and whose volume should change based on how close the player is standing to them when they play. See Figure 4.44.



```
AudioClip              track-1_0
Output                 None (Audio Mixer Gr
Mute                   ☐
Bypass Effects         ☐
Bypass Listener Effe   ☐
Bypass Reverb Zone     ☐
Play On Awake          ✓
Loop                   ✓
Priority               128
Volume                 1
Pitch                  1
Stereo Pan             0
Spatial Blend          0
Reverb Zone Mix        1
```

Looping a Music Track

And now, let's take the game for a test run! Click the *Play* button on the tool bar and test it out. If the music doesn't seem to play, check that the *Mute Audio* button is not enabled from the Game Tab. See Figure 4.45.

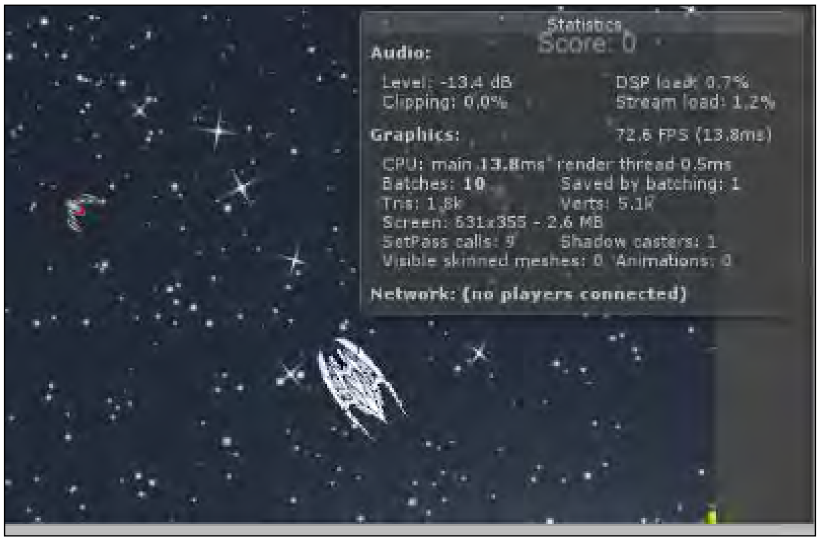


```
☑ Music                          ☐ Static
Tag  Untagged          Layer  Default
Position   X 0   Y 0   Z 0
Rotation   X 0   Y 0   Z 0
Scale      X 1   Y 1   Z 1
AudioClip              track-1_0
Output                 None (Audio Mixer Gr
Mute                   ☐
Bypass Effects         ☐
Bypass Listener Effe   ☐
Bypass Reverb Zone     ☐
Play On Awake          ✓
Loop                   ✓
Priority               128
Volume                 1
Pitch                  1
Stereo Pan             0
Spatial Blend          0
Reverb Zone Mix        1
```

Playing a Game – disabling Mute Audio, if needed

Testing and Diagnosis

With practically all games you'll need to spend considerable time testing and debugging heavily to reduce bugs and errors as far as humanly possible. With this sample program very little debugging and testing has been required for you, but that's not because the game is simple. It's because I've already pre-checked and pre-tested most of the code and functionality before presenting the material to you in this book; ensuring that you get a 'smooth learning experience'. For your own projects, however, you'll need to do lots of testing. One way to get started, is by using the *Stats* panel. To open this, click the *Stats* button on the *Game* Tab. See Figure 4.46.

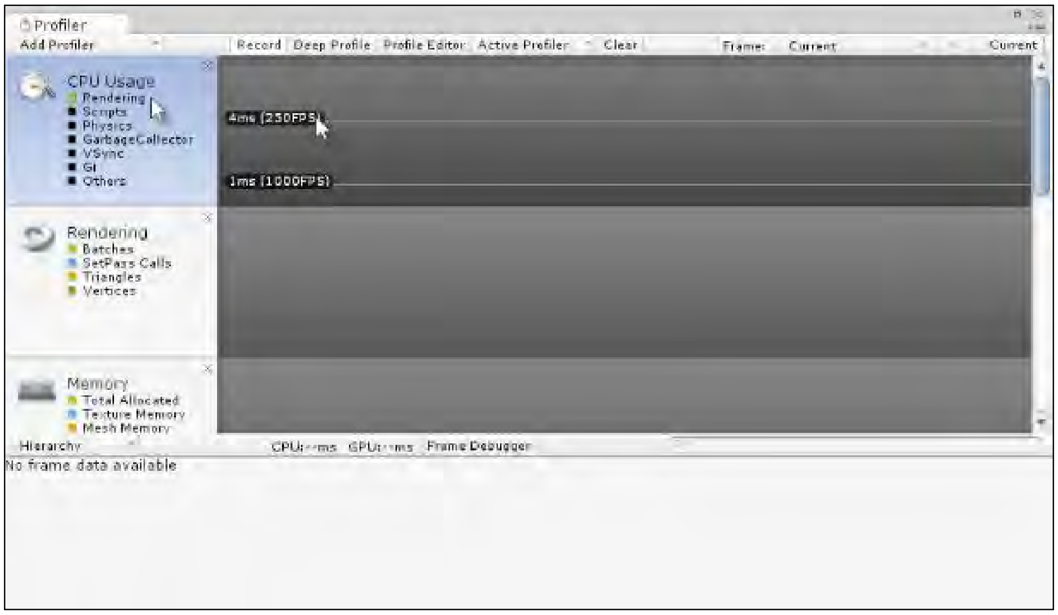


```
Audio:
  Level: -13.4 dB                DSP load: 0.7%
  Clipping: 0.0%                 Stream load: 1.2%
Graphics:                        72.6 FPS (13.8ms)
  CPU: main 13.8ms  render thread 0.5ms
  Batches: 10          Saved by batching: 1
  Tris: 1.8k           Verts: 5.1k
  Screen: 631x355 - 2.6 MB
  SetPass calls: 9     Shadow casters: 1
  Visible skinned meshes: 0  Animations: 0
Network: (no players connected)
```

Viewing Game Performance Information via the Stats Panel

More details on the Stats panel is included in Chapter 2 of this book, and more information can be found online at the Unity Documentation here: `http://docs.unity3d.com/Manual/RenderingStatistics.html`

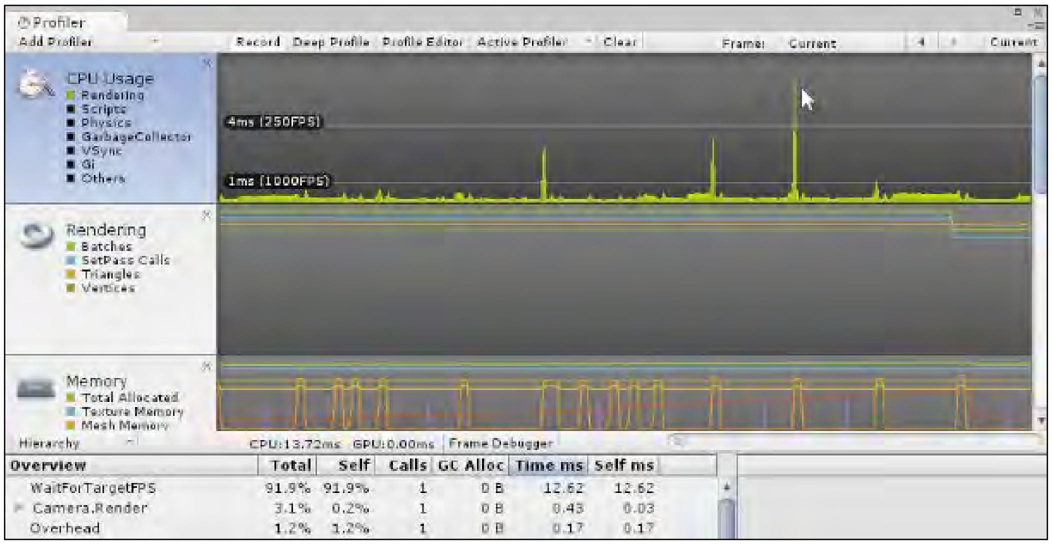
Another Debugging tool is the *Profiler*. This is useful when the Stats panel has already helped you identify a general problem, such as a low FPS, and you want to dig deeper to find where the problem might be located. More details on the Profiler are included later, in Chapter 6, but a short introduction is worth including here. To access the Profiler tool, select *Window* > *Profiler* from the application menu. This displays the Profiler window. See Figure 4.47.



No frame data available

Accessing the Profiler Window

With the Profiler Window open, click Play on the toolbar to play test your game. When you do this, the Profiler Window fills with color-coded performance data, in a graph. See Figure 4.48. Green represents the performance of rendering (graphical) data. Reading and understanding the graph requires some experience, but as a general rule, watch out for 'mountains and peaks'. That is, watch out for sharp fluctuations in the graph (sharp ups and downs), as this *could* indicate a problem, especially when it roughly coincides with frame rate drops.

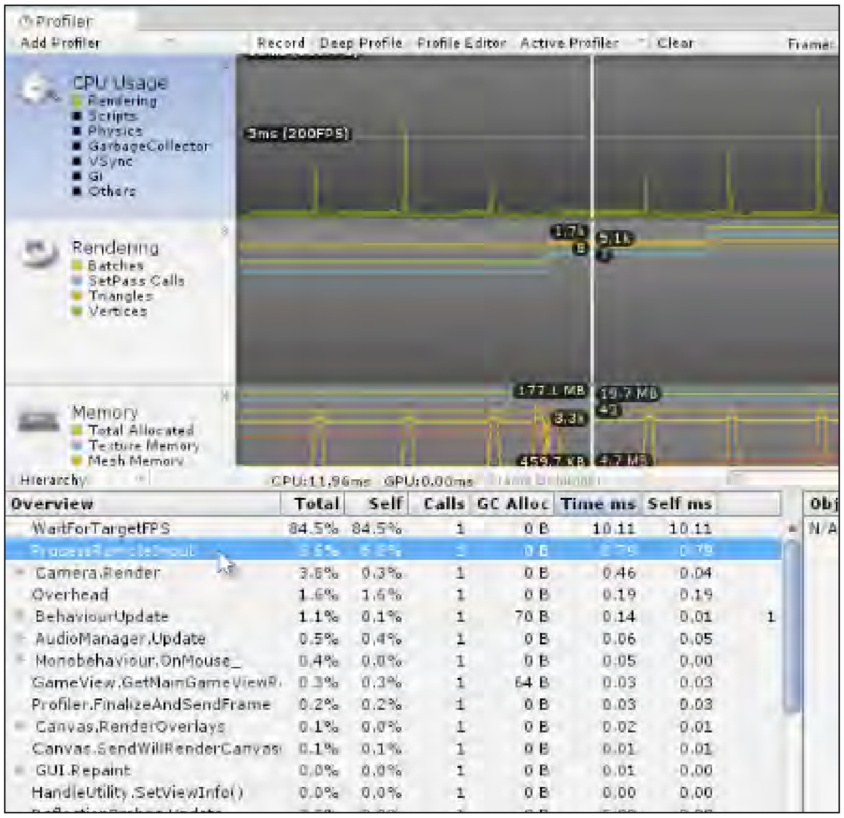


Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	91.9%	91.9%	1	0 B	12.62	12.62
Camera.Render	3.1%	0.2%	1	0 B	0.43	0.03
Overhead	1.2%	1.2%	1	0 B	0.17	0.17

During gameplay the Profiler populates with data

If you want to investigate further, simply *Pause* the game, and then click inside the graph. The horizontal axis (X axis) represents the most recent Frames, and the Vertical axis represents workload. When you click in the graph, a line marker is added to indicate the frame under investigation. Beneath the graph, a list of all main processes for that frame are presented, typically ordered from top to bottom in the heaviness of their workload the proportion of frame time for which the process accounted.

Heavier process are listed at the top. See Figure 4.49.



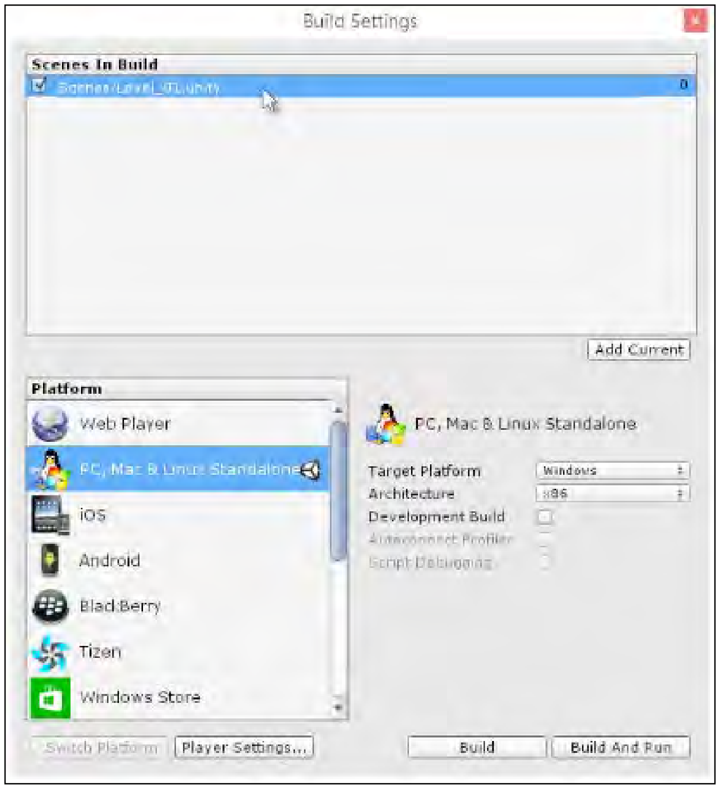
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms	
WaitForTargetFPS	84.5%	84.5%	1	0 B	10.11	10.11	
ProcessRemoteInput	6.6%	6.6%	1	0 B	0.79	0.79	
Camera.Render	3.8%	0.3%	1	0 B	0.46	0.04	
Overhead	1.6%	1.6%	1	0 B	0.19	0.19	
BehaviourUpdate	1.1%	0.1%	1	70 B	0.14	0.01	1
AudioManager.Update	0.5%	0.4%	1	0 B	0.06	0.05	
Monobehaviour.OnMouse_	0.4%	0.0%	1	0 B	0.05	0.00	
GameView.GetMainGameViewR	0.3%	0.3%	1	64 B	0.03	0.03	
Profiler.FinalizeAndSendFrame	0.2%	0.2%	1	0 B	0.03	0.03	
Canvas.RenderOverlays	0.1%	0.0%	1	0 B	0.02	0.01	
Canvas.SendWillRenderCanvas	0.1%	0.1%	1	0 B	0.01	0.01	
GUI.Repaint	0.0%	0.0%	1	0 B	0.01	0.00	
HandleUtility.SetViewInfo()	0.0%	0.0%	1	0 B	0.00	0.00	

Investigating performance data with the Profiler

More information on the Profiler can be found at the Online Unity Documentation here: `http://docs.unity3d.com/Manual/Profiler.html`

Building

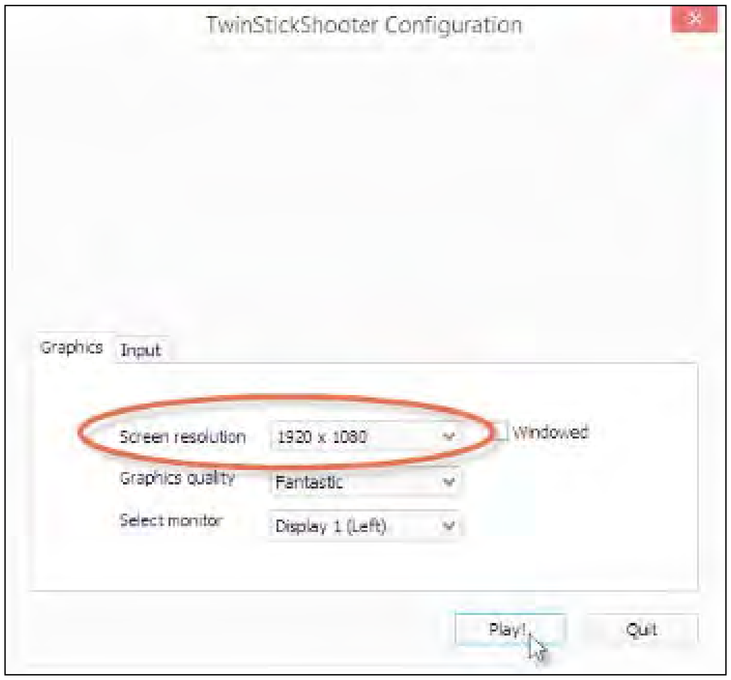
Now, finally, we're ready to *Build* our game into a stand-alone form ready to send off to friends, family and testers! The process for doing this is the same as that detailed in Chapter 2, for building the coin collection game. From the application menu, choose *File* > *Build* Settings. From the Build Dialog, add our level to the level list, by simply clicking the button *Add Current*. Or else, drag and drop the level from the Project Panel into the level list. See Figure 4.50.



Target Platform: Windows
Architecture: x86
Development Build: ☐

Preparing to Build the Space Shooter

For this game, the Target Platform will be Windows. Consequently, select the option *PC, Mac and Linux Standalone* from the Platform list, if it's not selected already. If the Switch Platform button (at the bottom-left) is not disabled, then you will need to press that button, confirming to Unity that it should build for the selected platform, as opposed to a different platform. And then click the *Build and Run* button. On clicking this, Unity prompts you to select a folder on your computer where the built file will be output and saved. Once generated, double click the executable to run it and test. See Figure 4.51.



Screen resolution 1920 x 1080
Graphics quality Fantastic
Select monitor Display 1 (Left)

Test running the game as a standard Windows executable

Summary

Great work! We're really on a roll now, having completed two solid Unity projects. The first project was a coin collection game, and the second a twin stick shooter. Both are, ultimately, simple games in that they don't rely on advanced mechanics or display sophisticated features. But, even very sophisticated games, when boiled down to their fundamental ingredients, can be found to rest on a similar foundation of essential concepts, such as the ones we've covered so far. That's why our projects are so critical to understanding Unity in a deep way. Next, we'll move onto creating a more 2D focused game, considering interfaces, sprites, and physics, and lot's more!