Unity Game Development

Succinctly®

by Jim Perry

Syncfusion | Technology Resource Portal

Unity Game Development Succinctly

By

Jim Perry

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey
Copy Editor: John Elderkin
Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Jim Perry has been a software engineer for more than 20 years, and he spent three of those in the game development industry. He's a former multiyear Microsoft XNA MVP and current ID@Xbox MVP. This is his fourth book on game development. When he's not writing code for his day job or his books, he's usually working to create the next hit indie game.

Preface

Code samples

While this e-book will give the reader mostly step-by-step instructions on how to get a full game created, some gaps might exist. The full project will be available online here at http://machxgames.com/files/UnitySuccinctly.zip.

How to contact me

I'm available to answer game development-related questions online—general game development-related, Unity specific, or sometimes Monogame/XNA for those who haven't made the transition yet to Unity. Here are some ways to contact me:

Twitter: @MachXGames

Email: jim@machxgames.com

Facebook: post a message to the one of the Unity/Xbox groups:
https://www.facebook.com/groups/UnityIndieDevs
https://www.facebook.com/groups/XboxOneIndieDevs

Chapter 1 Getting Started

Introduction

Indie game development is a huge business, unique in that developers are capable of making money on par with that of the big-name professional game companies such as Blizzard (*World of Warcraft*, *Diablo*) or Bioware (*Mass Effect*, *Dragon Age*). Granted, it's the rare indie developer who reaches such heights, but someone reading this could go on to develop the next *Minecraft* or *Angry Birds*. Not so long ago, being an indie game developer meant posting your game on a couple of Internet sites and hoping for the best. But with the mobile explosion, the opening of Steam's Greenlight service, and tools that make it possible to get on consoles, indies are capable of standing toe-to-toe with AAA devs and even outshining them with original and risky games that publishers won't touch.

Among the tools indies can choose to use for their game development, Unity has become one of the top choices. Aside from a free version that offers everything most developers will need to create a game, constant feature additions, two dozen supported platforms, an Asset Store in which developers can buy and sell content, a huge community of fellow developers to learn from and obtain support from, and outstanding docs, tutorials, and videos from the Unity team to help you out, Unity is helping millions of developers (4.5 million registered developers as of 2015) realize their dreams of creating their own games. According to statistics from 2015, more than 600 million gamers have played games developed using Unity. It holds 45% of the game engine market and 47% of game developers are using it.

During the course of this e-book, we'll cover the major features of Unity and use those to create a small 2-D game. We'll go over every step, from downloading and installing Unity to expanding the finished game with features that would make it something resembling games that are selling now on PCs and consoles. I will assume you have some basic knowledge of C#, the most popular language used for coding in Unity games. Some knowledge about the basics of game programming will be useful, but even someone completely new to game development should be able to get up and running quickly. The game we'll be creating is reminiscent of the old game *Robotron 2084* (or *Smash TV*). For those not old enough to remember versions of that great game, here's the setup—the character is stuck in an arena, destroying constantly spawning ghosts. Why ghosts? Who knows—maybe I just finished watching *Ghostbusters*? But Figure 1 offers a screenshot of the finished game.

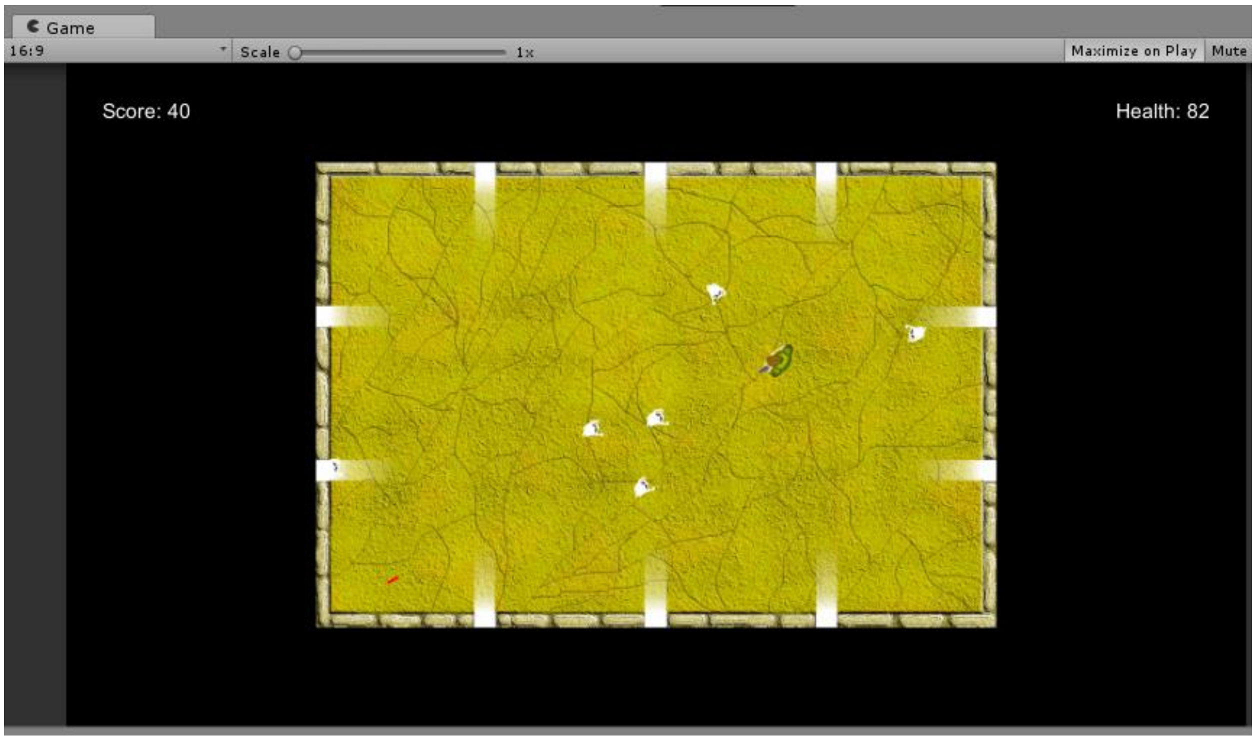


Figure 1: Completed Game Screenshot

What is Unity?

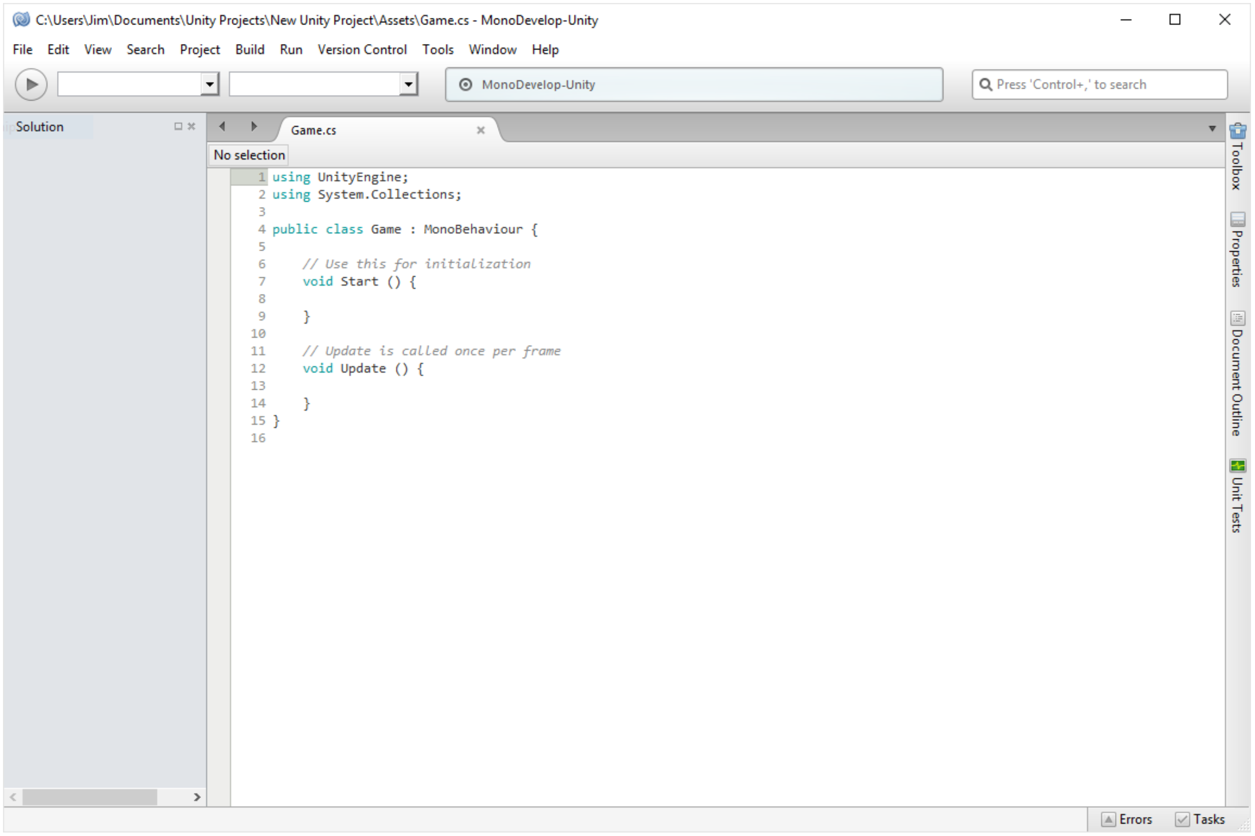
Unity was first released on June 8, 2005. In the 11+ years since then, it has advanced to its current stable version of 5.4.2 (as of the end of October, 2016). At the time of this writing, a beta of version 5.4 is available for those who like to live on the cutting edge.

Unity is a cross-platform game engine. Cross-platform means that it either runs on or allows what is created with it to run on multiple hardware/OS combinations. In the case of Unity, both are true. Currently, Unity can be installed on OS X and Windows operating systems. Ubuntu support (a version of the Linux OS) was released in beta in August, 2015.

A game engine is software that allows for the creation of video games. At a minimum, this means that the software allows the developer to write code to draw 2-D/3-D graphics on the screen, obtain and process input from players that use mouse, keyboard, gamepad, or other hardware, and play sounds and music through the hardware's speakers. A wide variety of other features may be possible through the engine—physics and collision detection, networking, AI, video, etc. Unity supports all of these and more. We'll only use a couple of these features, but we will address implementing others in order to make the game we create attractive to gamers.

Unity, like other game engines, features an Integrated Development Environment (IDE) in which the developer creates the game. There are two parts of Unity's IDE—the main Unity editor and the code editor. We'll look at the Unity IDE and interface shortly.

MonoDevelop was the usual editor until a plug-in was developed that allowed Visual Studio to write and debug Unity game scripts. Originally, a port of another open source IDE for use on Linux, the version packaged with Unity, was specifically customized for developing Unity games.


C:\Users\Jim\Documents\Unity Projects\New Unity Project\Assets\Game.cs - MonoDevelop-Unity
File Edit View Search Project Build Run Version Control Tools Window Help
MonoDevelop-Unity
Press 'Control+,' to search
Solution
Game.cs
No selection
Toolbox
Properties
Document Outline
Unit Tests
Errors
Tasks

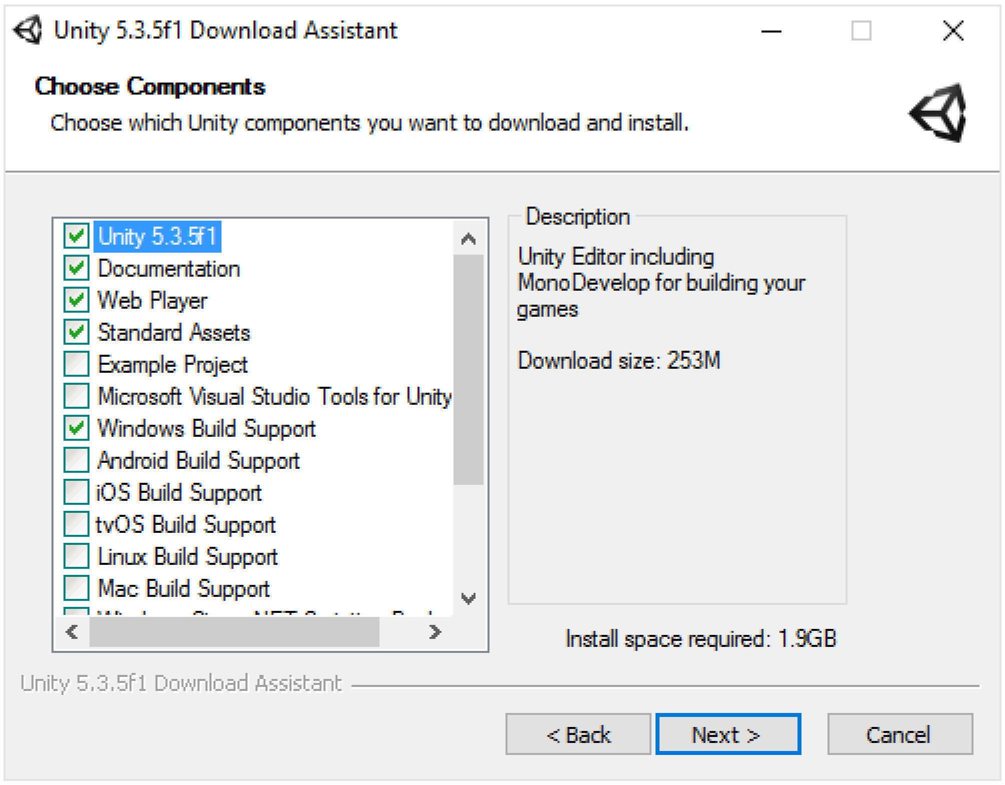
```
using UnityEngine;
using System.Collections;

public class Game : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}

```

Figure 2: Monodevelop IDE

Setting up

First, you should head over to the Unity website and download the latest version. As of the time of this writing, the latest version is 5.4.0f3. The main download site will have all the different versions currently available, but you'll want the free Personal version. Next, you should create a Unity account. You can find the page to create one at https://id.unity.com/account/new.

When you've set up your Unity account, run the installer you downloaded. After a couple of Next buttons, you'll come to the Choose Components dialog, as shown in Figure 3.



Unity 5.3.5f1 Download Assistant

Choose Components

Choose which Unity components you want to download and install.

- [✓] Unity 5.3.5f1
- [✓] Documentation
- [✓] Web Player
- [✓] Standard Assets
- [ ] Example Project
- [ ] Microsoft Visual Studio Tools for Unity
- [✓] Windows Build Support
- [ ] Android Build Support
- [ ] iOS Build Support
- [ ] tvOS Build Support
- [ ] Linux Build Support
- [ ] Mac Build Support

Description

Unity Editor including MonoDevelop for building your games

Download size: 253M

Install space required: 1.9GB

Figure 3: Unity Installer Dialog

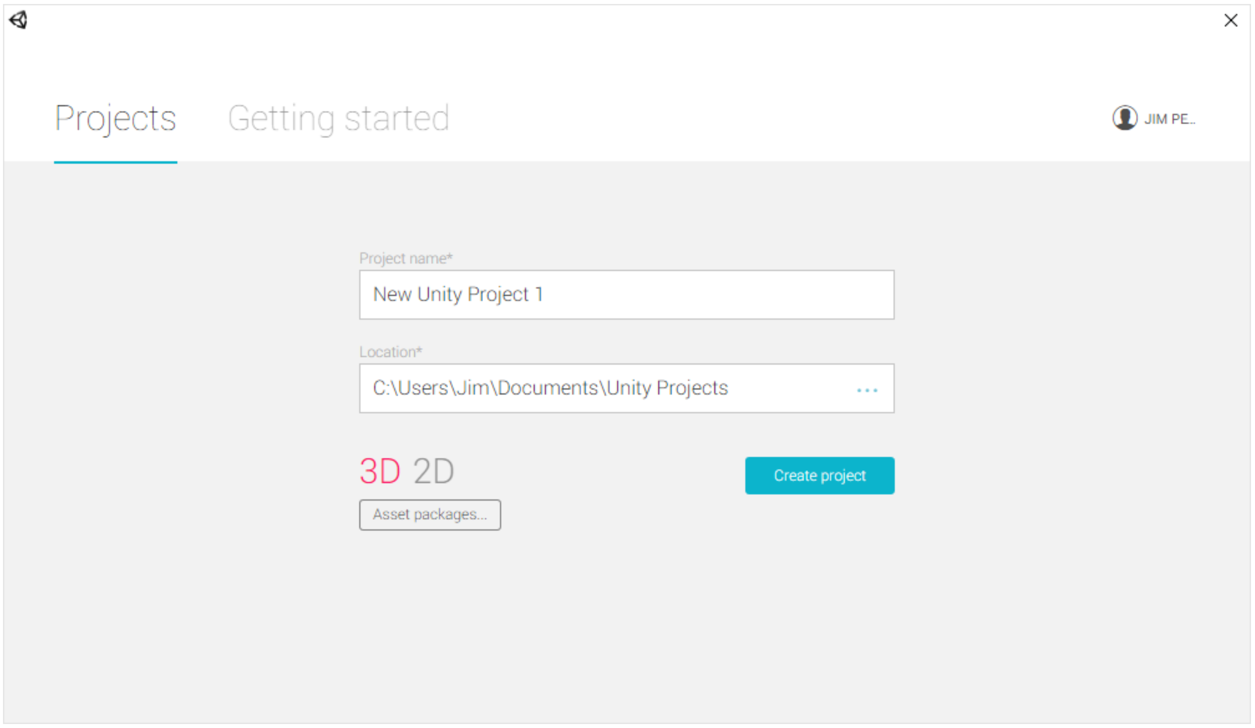
You'll find a couple of important options to select in this dialog.

Unity uses MonoDevelop by default, but if you use Visual Studio as your IDE, you'll want to check the Microsoft Visual Studio Tools for Unity that will allow you to use it as your default editor for Unity. If you don't have Visual Studio, you can download the Community Edition.

The Example Project could be a good starting point as you learn your way around Unity.

By default, only the Windows Build Support option is checked. If you plan on supporting other platforms, you'll want to check those. Be aware that there could be costs or other requirements in order to be able to deploy your games to a specific platform.

After installing Unity, launch it and you'll see the New Project dialog, as shown in Figure 4.



Project name*
New Unity Project 1
Location*
C:\Users\Jim\Documents\Unity Projects

Figure 4: New Project Dialog

The default for new projects is 3-D. Make sure you click **2-D** before you create your project—the game we'll be creating is 2-D, and it is better to start with the correct setting rather than convert later. You can also select the asset packages that get added to the project when it's created. You can easily add packages later if you forget. One package you'll probably want to add is the Visual Studio 2015 Tools.

After selecting the packages you want to include, give the project a suitable name (I called the game we'll be creating *Ghost Arena*), select the location (although I use the default to keep all my projects in one location), and click **Create project**. When Unity finishes, you'll see something like Figure 5.

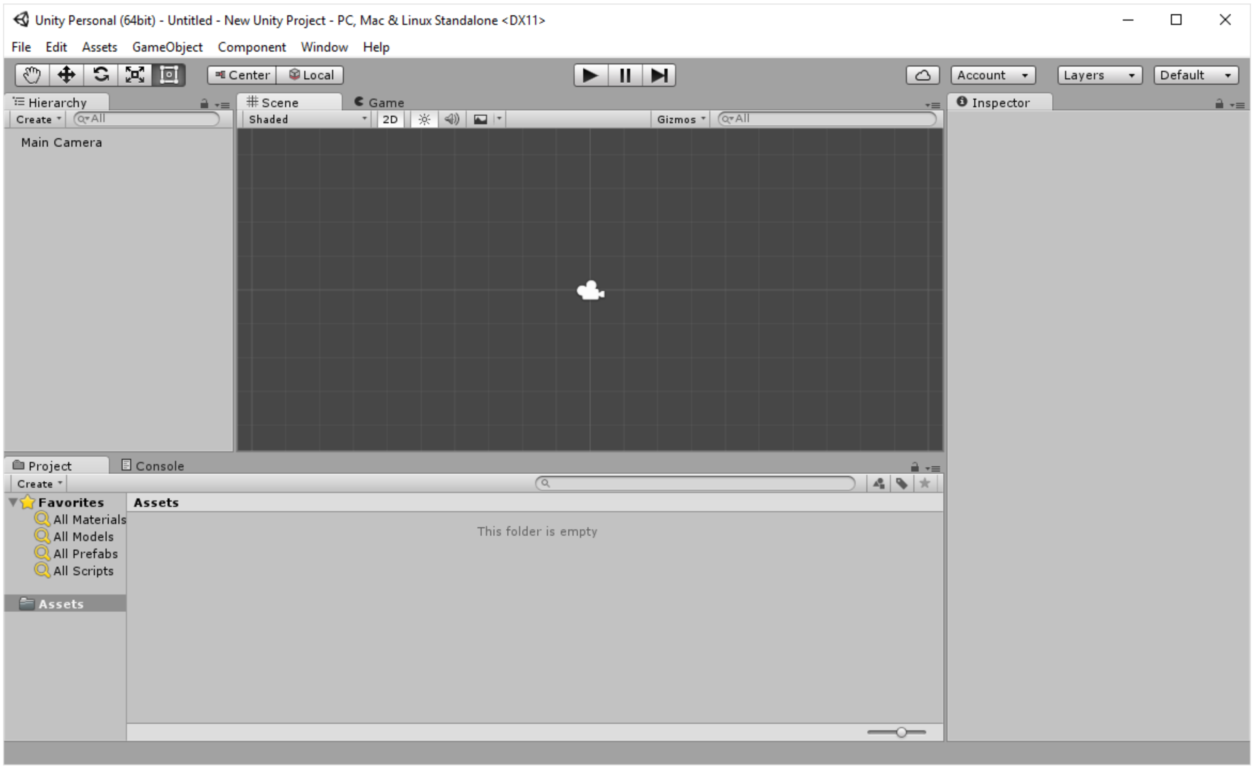

Unity Personal (64bit) - Untitled - New Unity Project - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Window Help
Center Local
Account Layers Default
Hierarchy Scene Game Inspector
Create Shaded 2D Gizmos
Main Camera
Project Console
Favorites
All Materials
All Models
All Prefabs
All Scripts
Assets
This folder is empty

Figure 5: Default Unity IDE

Unity interface

Unity is ready to go. Looking at the figure above, you might find Unity's interface a little intimidating. Let's take some time and go over the main features of the interface.

The menu gives us access to most of Unity's functionality. The File menu allows you to:

- Create, open, and save scenes (see the Scene section for more information)
- Create, open, and save projects
- Display the Build Settings dialog
- Build and run your project
- Exit the application

All very standard, of course.

The first three sections of the Edit menu are similar to most other applications in that they offer Undo and Redo functionality—cut, copy, paste, and delete. The Duplicate menu item creates a copy of whichever object is currently selected in the Hierarchy. Of the remaining items, you'll probably deal with the following a lot:

- Play, Pause, and Step—although you can use the menu items or their shortcuts, you will probably use the toolbar buttons instead.
- Project Settings—the submenus under this menu allow you to set a lot of defined values for the following areas:

- Audio—allows you to specify the defaults for settings such as global volume, speaker mode (stereo, quad, surround, etc.), and audio disabled by default.
- Editor
- Input—allows you to specify the defaults for keys and buttons for in-game actions such as firing, jumping, and movement, along with the sensitivity and deadzone for controls.
- Network—allows you to specify the level of debug message that gets sent to the console and sets the default number of times per second that data is sent across the network.
- Physics (2-D and 3-D)
- Player—allows you to specify default resolution and platform-specific settings, icons, and game and company name.
- Quality—allows you to specify the default settings for each platform along with a number of rendering settings for higher or lower quality.
- Graphics—allows you to specify defaults for a number of shader settings.
- Script Execution Order
- Tags and Layers—allows you to specify your tags for identifying objects by code and rendering layers.
- Time—allows you to specify the default value for fixed timestep, max allowed time for physics calculations and code events, and the time scale (used for effects such as bullet-time).

Note: Definition— Deadzone is a small, set radius around the center of your controller axis where user input is ignored.

The Assets menu is pretty straightforward. It allows you to add assets (something larger than a GameObject) to the current scene. You can create a new asset, which could be anything from code to a new scene to a physics material. In later sections, we'll dig into some of the specific assets we'll use in our game. You can also import packages, which are assets put together to make something larger. These could be levels for your game; weapons consisting of a model, textures, and attached code; or they could be complete base games to which you add your own features. The Assets menu is also where you can open the project in your coding IDE.

The GameObject menu allows you to add objects such as sprites, user-interface elements, lights, and 3-D objects such as cubes and cylinders. You can also adjust objects already in the scene within the Hierarchy or in the view of the scene (useful if you have a lot of objects and need to focus your view on one).

The Component menu allows you to add components to GameObjects or the scene. Components are the functional pieces of GameObjects. You'll find items such as physics colliders, audio filters, obstacles for navigation meshes, and network management pieces.

The Window menu allows you to change the layout of the Unity IDE using several preset layouts. You can also set and delete your own custom layouts. This menu also gives you access to the Asset Store inside the Unity IDE rather than using a separate browser window. You can also display pieces of the IDE—for example, the Hierarchy, Project, Console, or Inspector (if you hide them).

The Help menu has standard menu items but also includes items such as a link to download beta versions of the application and manage your license keys.

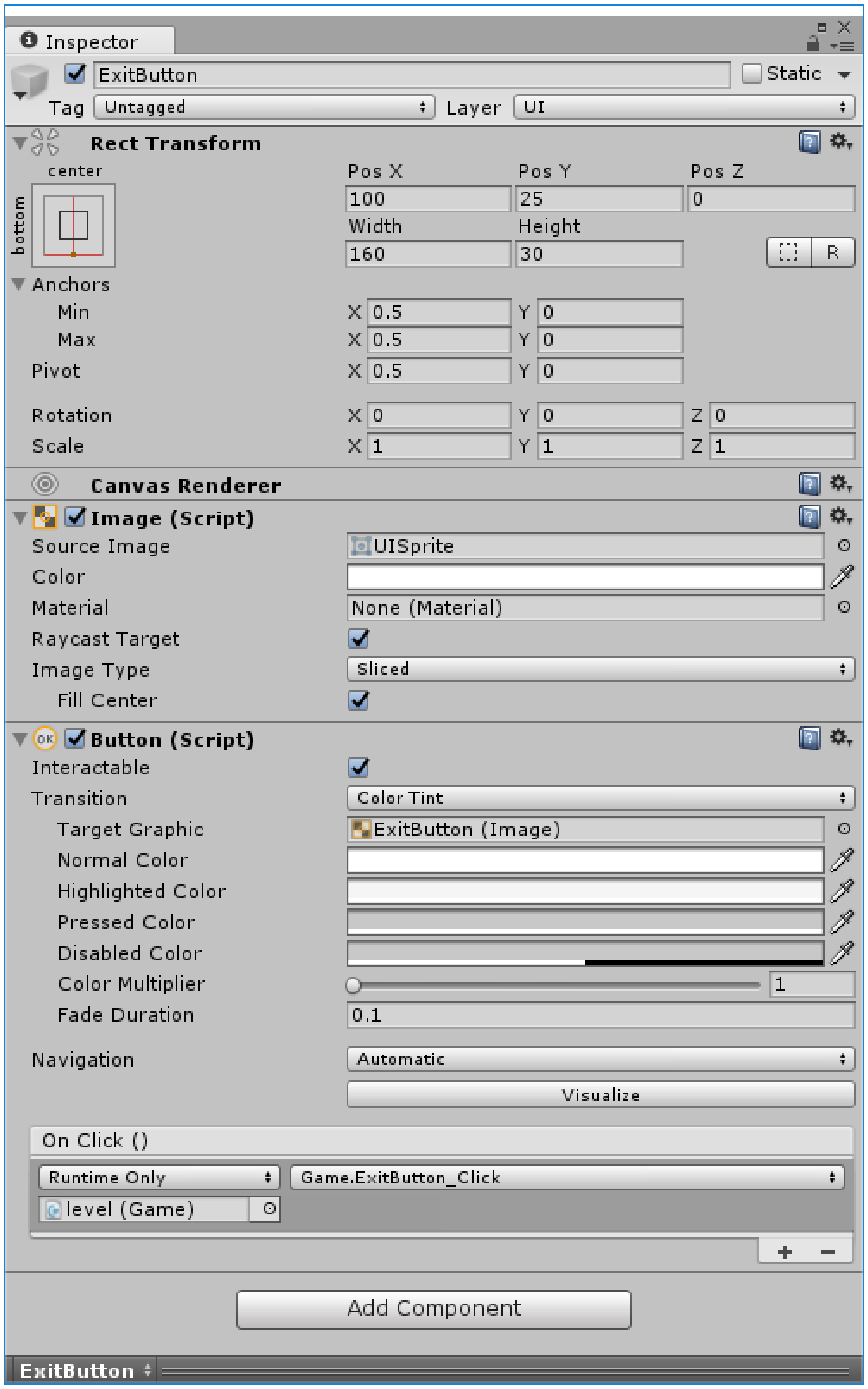
Hierarchy window

The Hierarchy will show you the objects in the current scene. You can make them children of other objects by moving objects around within the Hierarchy. You can also select multiple objects in order to quickly edit common properties, duplicate them, or make prefabs out of them. Right-clicking an object displays a context-sensitive menu for that object.

Note: Definition— A prefab is a template or blueprint that allows you to create new instances of it. Changes to the prefab are reflected in all instances.

Inspector window

Selecting an object in the Hierarchy window will display its properties in the Inspector Window, as shown in Figure 6.


Inspector

ExitButton ✓ — Static ☐

Tag: Untagged — Layer: UI

Rect Transform

center / bottom

	Pos X	Pos Y	Pos Z
	100	25	0
	Width	Height	
	160	30	

Anchors

	X	Y	Z
Min	0.5	0	
Max	0.5	0	
Pivot	0.5	0	
Rotation	0	0	0
Scale	1	1	1

Canvas Renderer

Image (Script) ✓

Source Image	UISprite
Color	
Material	None (Material)
Raycast Target	✓
Image Type	Sliced
Fill Center	✓

Button (Script) ✓

Interactable	✓
Transition	Color Tint
Target Graphic	ExitButton (Image)
Normal Color	
Highlighted Color	
Pressed Color	
Disabled Color	
Color Multiplier	1
Fade Duration	0.1
Navigation	Automatic

Visualize

On Click ()

Runtime Only — Game.ExitButton_Click

level (Game)

Add Component

ExitButton

Figure 6: Inspector Window

The Inspector window lets you quickly change properties and see their effects in either the Scene window or Game window, depending on the property, without having to actually run the game. As you can see in Figure 6, even properties in scripts attached to the object can be set.

Each set of properties can be hidden by clicking the arrow at the top left of each section, allowing you to see exactly what you need to see. Clicking the gear at the top right displays a menu that allows you to reset the properties for that section, remove the component, move it up or down in the list in the window, or copy and paste it. Clicking the book will display the Help section of the documentation for the item in that section.

Project window

The Project window shows all the assets that have been added to the project. The assets within the project can be organized into folders, which means you can have a folder for all of your scripts, scenes, audio files, sprites, etc. For large games, this type of organizing is vital in order for you to quickly find the assets you need.

The left side of the Project window shows the hierarchy of the assets. The root of the Hierarchy will always be an Assets folder. Clicking on a folder will display all the assets in that folder in the right side of the window. Clicking on an asset in the right side will show the properties of that asset in the Inspector window and an asset-specific section of the Inspector window at the bottom will allow actions such as playing an audio file and seeing a sprite with info conveying width/height, type of graphic file, size of the file, etc.

Scene and Game windows

You design the parts of your game in the Scene window. You do this by dragging assets into it from the Hierarchy or adding new assets through menu items. The Game window shows you how the current scene will look when the game is actually being played.

You can control how you navigate around the scene using the toolbar buttons. From left to right, they work this way:

- Hand Tool—when this is selected, clicking and dragging in the scene will move the Camera object. Holding the Alt key while left-clicking or right-clicking will move the camera around the current pivot point or zoom in or out.
- Translate Tool—clicking on an object while this is selected will display controls that allow you to move the object around in the scene on the x, y, and z axes.
- Rotate Tool—clicking on an object while this is selected will display controls that allow you to rotate the object on the x, y, and z axes.
- Scale Tool—clicking on an object while this is selected will display controls that allow you to resize the object on all axes at once.
- Resize Tool—clicking on an object while this is selected will display controls that allow you to resize the object on an axis independent of the others.

There are also shortcut keys for moving around in the scene and moving objects around in the scene. Which shortcuts you use will depend on your normal keyboard and mouse style.

You will also probably find yourself making a lot of mistakes in moving things around in the scene as you get used to Unity’s interface. Undo will become your very close friend.

The Unity editor has more functionality than could possibly be discussed in a short e-book. The online documentation for Unity is very extensive—trying to read through the entire thing would be a Herculean task. As you come across something that you don’t know or need to figure out, reading the relevant portion, as well as hitting the Unity forums, is the best way to go. As you get used to Unity and game development, you'll want to build bigger and more complex games. The documentation will become another friend, as will the rest of the material on Unity's website.

Setting up the game

Unity games are typically structured so that different types of assets are organized into their own folder. For our game, we'll only need six folders:

- Animations
- Audio
- Prefabs
- Scenes
- Scripts
- Sprites

Right-click the **Assets** folder in the Project window or the **Assets** folder window itself and select **Create | Folder** for each folder we'll need. You should end up with the structure in Figure 7.

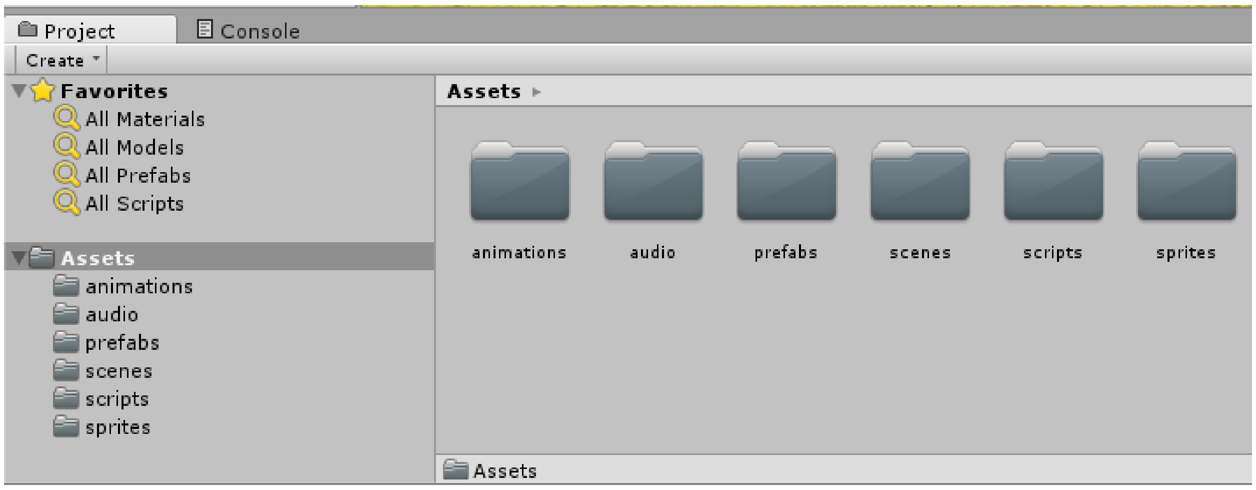


Figure 7: Project Folder Structure

At this point, you now have a blank slate to begin creating your game. We'll first take a look at scenes and scene management.

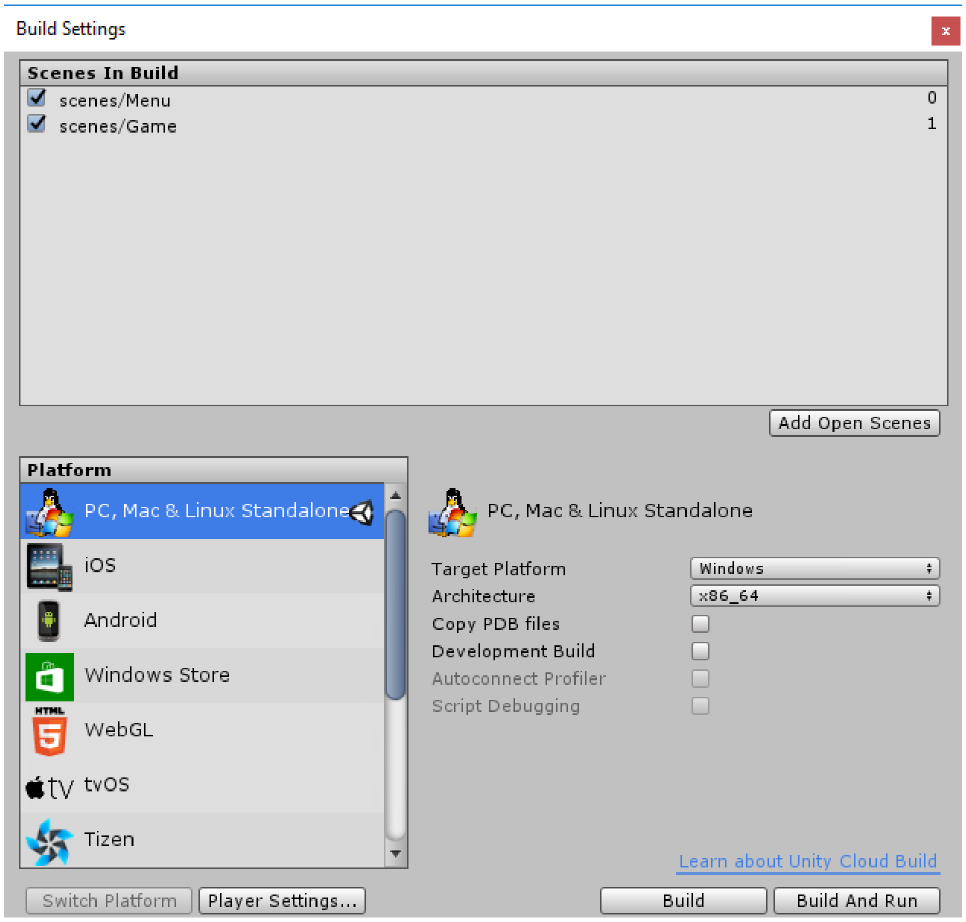
Chapter 2 Scenes and Scene Management

Unity games are organized using scenes. A scene represents a distinct visual portion of the game—a menu, options screen, the screen with the actual gameplay, etc.

Each scene will contain GameObjects and code. GameObjects can be menu buttons, text, input fields, sprites, audio sources, camera, particle systems, and more. Code files will usually be attached to objects or referenced from other code files. This code will control the GameObjects, handle events such as button clicks, play audio files, run AI entities, and engage every other aspect of your game. GameObjects themselves can contain a lot of built-in functionality, but you'll need to write at least some code in order to make a complete game. Our simple game will contain a minimal amount of code—seven code files and just under 400 lines of code, including "using" sections and white space, which means the actual number of lines is even smaller.

The Scene object in the Unity API is actually a struct with very few members and methods. Unless your game is very complex, you'll probably never use any of them in your code. Most simple games will move from scene to scene as needed using the methods in the **`SceneManager`** class.

The main thing to remember is how your scenes are organized in your project. You'll need to make sure that all of your scenes are added to the project's Build Settings, which is shown in Figure 8.


Build Settings

Scenes In Build

✓ scenes/Menu	0
✓ scenes/Game	1

Target Platform	Windows
Architecture	x86_64
Copy PDB files	☐
Development Build	☐
Autoconnect Profiler	☐
Script Debugging	☐

Figure 8: Build Settings Dialog

The first scene in the list will be the one that's loaded when the game first starts. If the scenes aren't in the order you want, you can simply drag and drop them until they are.

Typically, scenes will be added to the Build Settings automatically as you add them to your project. If for some reason you have scenes that don't show, simply open them and use the Add Open Scenes button in the dialog.

Scene Manager

The **`SceneManager`** class is responsible for handling the transitioning from scene to scene. Table 1 shows two methods for doing this.

Table 1: SceneManager Methods

`LoadScene`	Loads the scene by its name or index in Build Settings.
`LoadSceneAsync`	Loads the scene asynchronously in the background.

The overloaded signatures of the two methods are shown in Code Listing 1.

Code Listing 1: LoadScene Signatures

```
public static void LoadScene(int sceneBuildIndex,
SceneManagement.LoadSceneMode mode = LoadSceneMode.Single);
public static void LoadScene(string sceneName,
SceneManagement.LoadSceneMode mode = LoadSceneMode.Single);
public static AsyncOperation LoadSceneAsync(int sceneBuildIndex,
SceneManagement.LoadSceneMode mode = LoadSceneMode.Single);
public static AsyncOperation LoadSceneAsync(string sceneName,
SceneManagement.LoadSceneMode mode = LoadSceneMode.Single);
```

Both methods take either a string (the name of the scene) or an integer (the index in the build settings) as the first parameter. The second parameter is an enum specifying the mode for the scene loading, as in Table 2.

Table 2: LoadSceneMode Values

Single	Closes all currently loaded scenes and loads a scene.
Additive	Adds the scene to the current loaded scenes.

Most of the time, using **`LoadScene`** will be sufficient, as scenes will load with no more than a small delay. When you make larger games with larger levels, you might look into **`LoadSceneAsync`**.

Scene

The Scene struct has a number of members and methods, but for simple games it would be extremely unusual to need them. If you start writing scripts to use in the Unity editor, they might come in handy, however.

We'll need one more scene that we'll use for our actual gameplay. You can either right-click in the **Scenes** folder and select **Create | Scene** or use the Assets menu. I've simply called the scene Game. If you haven't already saved the scene that was added when the project was created, you'll be prompted to save it. Make sure you save it in the Scenes folder in order to keep things organized.

When you add a scene to your project, it comes with a Camera object. The camera controls what gets shown to the player. For a 2-D game, there's not much you'll need to do for the camera. As long as your sprites stay within the white rectangle that surround the camera icon in the scene window, they'll be displayed when the game runs. As Figure 9 shows, the Camera Preview that shows in the scene window when you select the Camera object in the Hierarchy window will let you know what the scene looks like when the game is run—without requiring that you switch over to the game window.

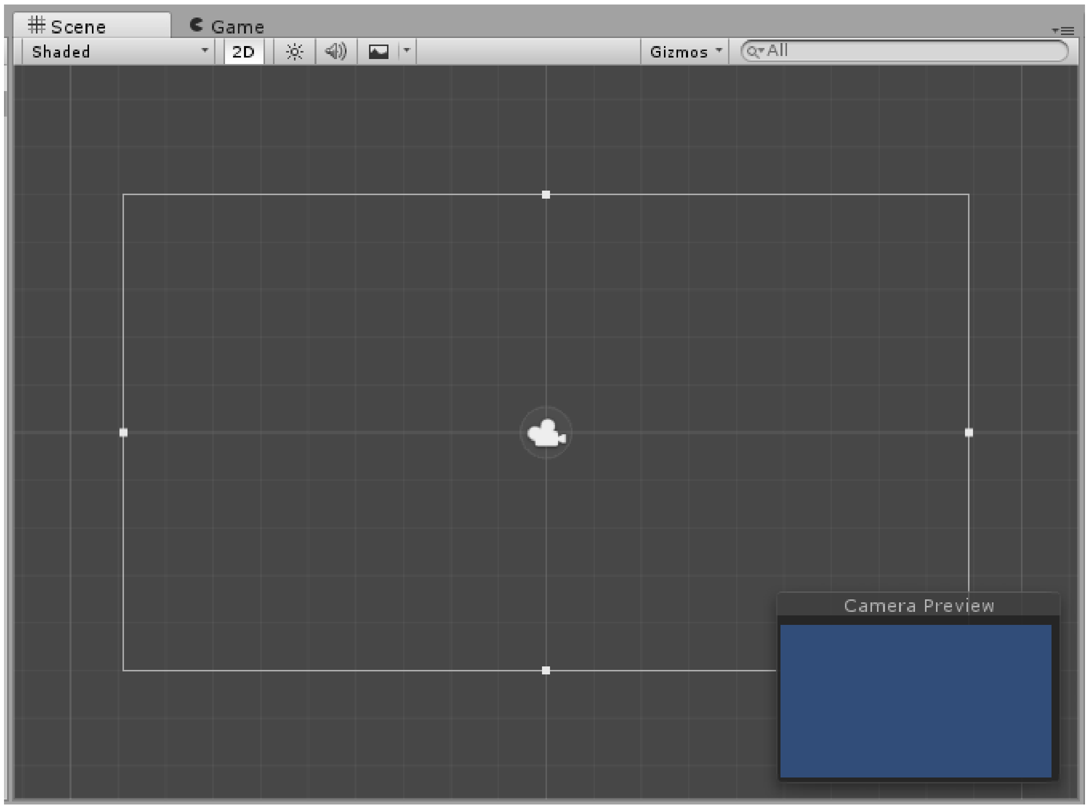
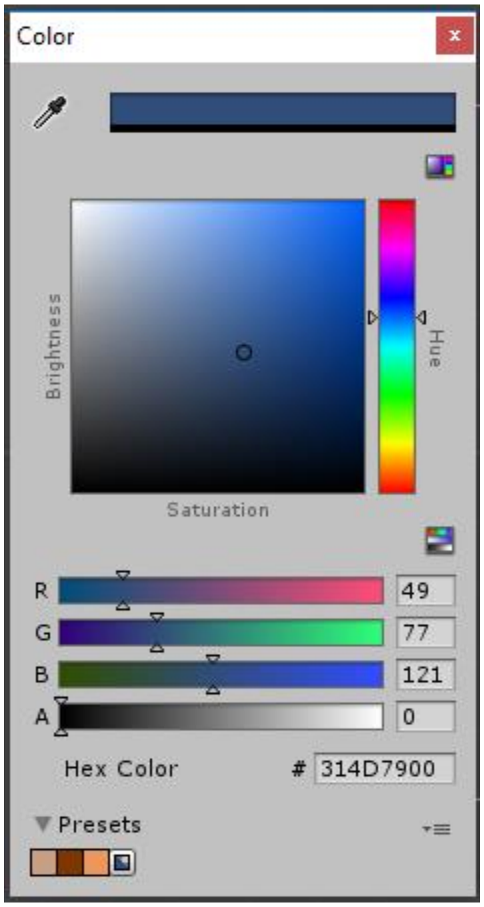


Figure 9: Camera Object Selected and Camera Preview

One thing you'll probably want to do with the camera is change the Background property from the default blue color. Click on the colored bar next to the Background label in the Inspector window to display Color picker dialog shown in Figure 10.



R 49
G 77
B 121
A 0
Hex Color # 314D7900

Figure 10*: Color Picker Dialog*

You can click on the circle in the color box and move it around to change the color or enter the RGB or hex values in the text boxes. In order to set the background to black, you can drag the circle to the bottom left or enter a 0 in each of the RGB text boxes. This will give you black, which I almost always use for the backgrounds of my scenes.

Game code

There is very little code in the game that is scene related, although we will end up with code in the Menu scene in order to load the Game scene and vice versa. If you want to take a look at code ahead of going over it here, look in the Script folder in the sample project. If you load the project in Unity, you can double-click a file to open it in Monogame or Visual Studio (if you have the VS for Unity tools installed). There's a High Score scene in the sample project, but there's no code to support it. If you want to implement functionality in order to keep track of scores, you add code to display it from the Menu scene by adding another button and using the same code logic we'll address shortly.

Obviously, a more complicated game would have more scenes and more scene-related code. A game with multiple levels would have code to load the different level scenes (if the levels were not data-based or able to be handled by one scene).

The code in the Menu scene loads the Game scene when the player selects the difficulty, as shown in Code Listing 2.

Code Listing 2: Difficulty Button Code

```
public void DifficultyButton_Click(int index)
    {
        Globals.DifficultyLevel = index;
        SceneManager.LoadScene("Game");
    }
```

Code Listing 3 shows that the code in the Game scene is virtually the same.

Code Listing 3: Exit Button Code

```
public void ExitButton_Click()
    {
        SceneManager.LoadScene("Menu");
    }
```

Most of your scene code will look this way. The coding is very easy, despite it being such an important part of your game. You'll spend 99% of your development time with scenes working in the Unity editor, laying out objects and tying code into them, not writing code specific to managing the scenes.

You'll notice the methods seem to be handling the click event of buttons, but there are no buttons in the scenes yet. We'll add them and other interface elements in the UI chapter, coming next, and see how to hook up the code to the interface.

Chapter 3 User Interface

The user interface is the part of the game that the user directly interacts with, either by clicking on with a mouse, entering data via the keyboard, or using some other input device. In this section, we'll look at the specific UI elements we use in our sample game and a couple of others that you might want to consider.

Button

We start with the button because it's the easiest thing to use for a main menu, and that's usually the first interaction the player has with a game. Figure 11 shows what our menu will look like.

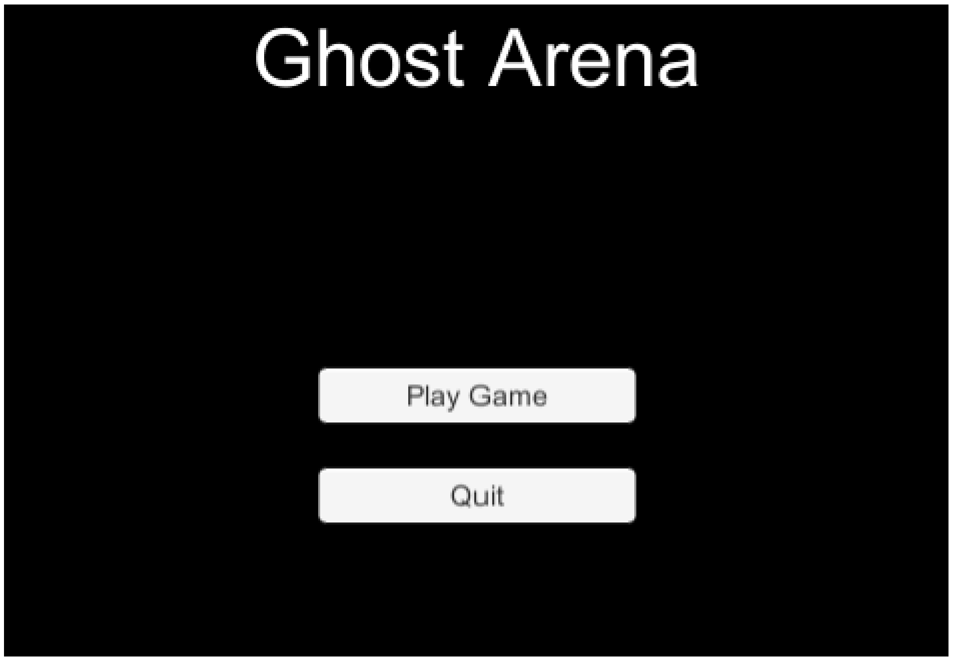
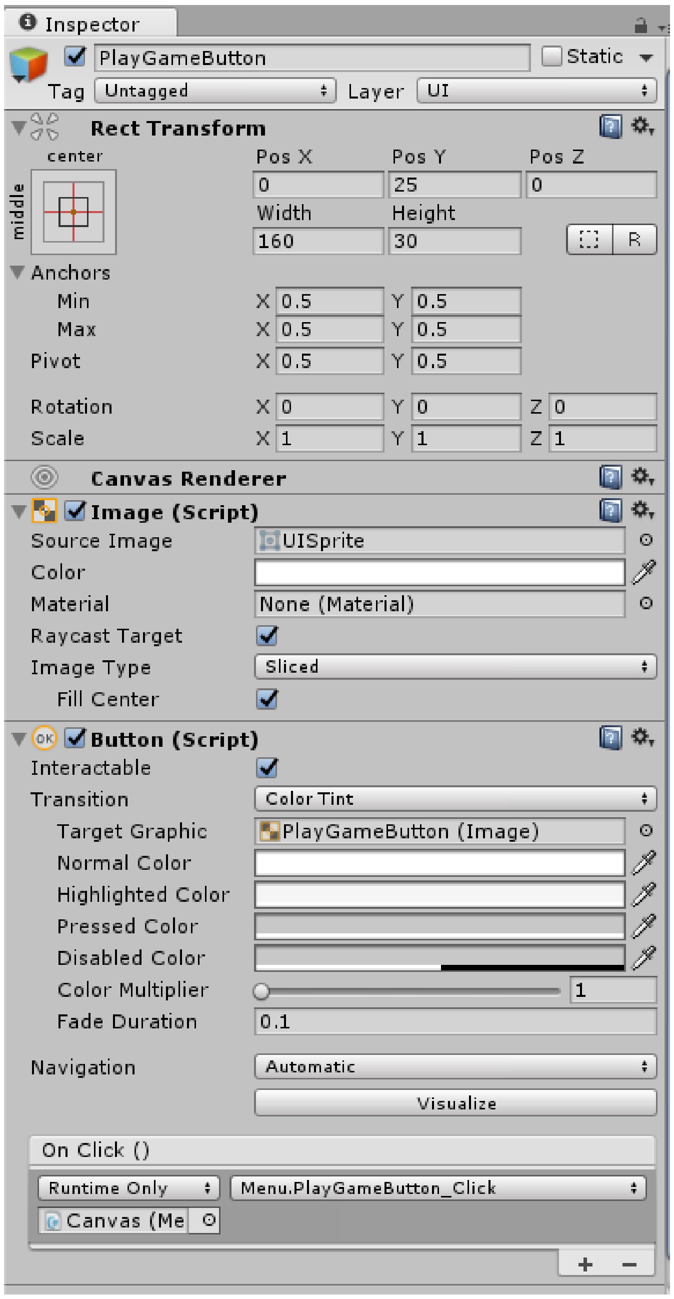


Figure 11: Main Menu

The menu is simply two buttons and a Text object. Not AAA quality, obviously, but we're merely looking for something to get the job done and a button provides all the interaction we need without being overly complicated to implement. There's a bit more behind these two buttons and we'll look at it shortly, but this is what players see when they first start the game.

The Button class has a lot of configurability to it. You can modify the properties of the displayed text, the image that makes up the button, and more, as Figure 12 shows.


Inspector
✓ PlayGameButton ☐ Static
Tag Untagged Layer UI

Rect Transform
center middle
Pos X 0 Pos Y 25 Pos Z 0
Width 160 Height 30
Anchors
Min X 0.5 Y 0.5
Max X 0.5 Y 0.5
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Canvas Renderer

✓ Image (Script)
Source Image UISprite
Color
Material None (Material)
Raycast Target ✓
Image Type Sliced
Fill Center ✓

✓ Button (Script)
Interactable ✓
Transition Color Tint
Target Graphic PlayGameButton (Image)
Normal Color
Highlighted Color
Pressed Color
Disabled Color
Color Multiplier 1
Fade Duration 0.1
Navigation Automatic
Visualize

On Click ()
Runtime Only Menu.PlayGameButton_Click
Canvas (Me
+ −

Figure 12: Button Properties

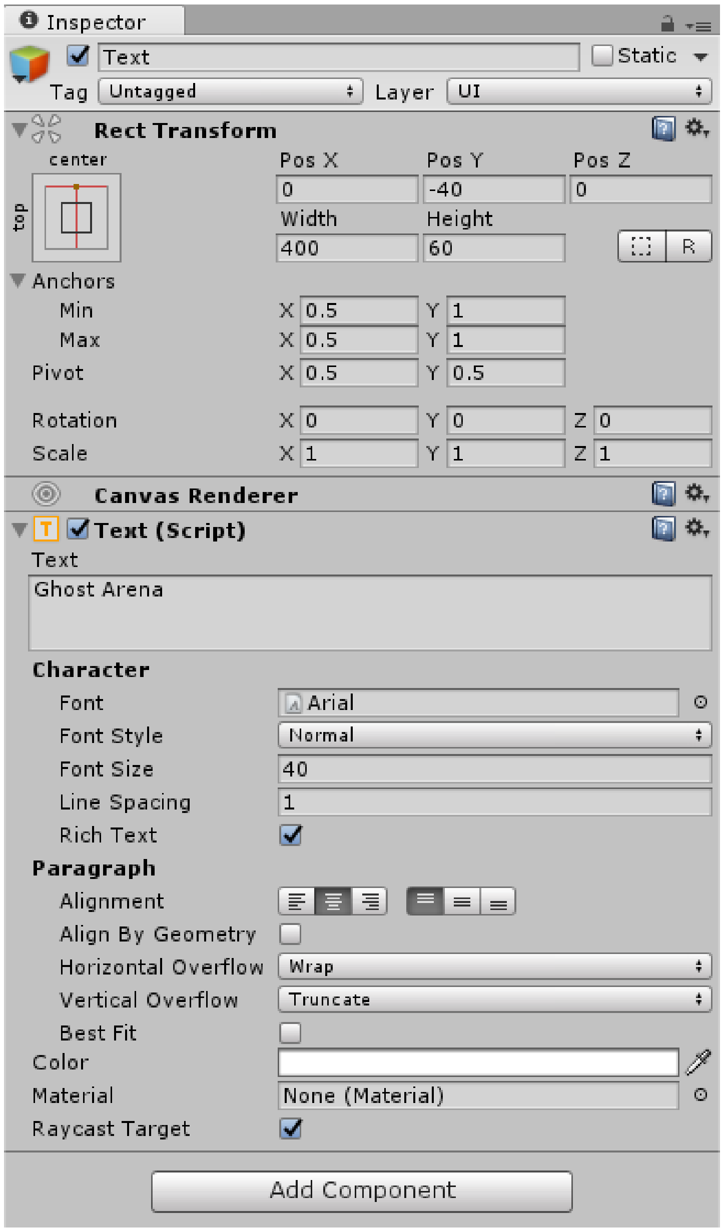
The check box (or toggle, as Unity calls it) at the very top, which will show and hide the button, and the Interactable check box, which will enable and disable allowing the user to click it, are a couple of properties you should get to know.

The other important property is the On Click section, which allows you to attach code to the button in order to allow it to do something. The + and – icons at the bottom of the section allow you to add and remove handlers for your code. Once you add a handler, the section below the Runtime Only drop-down is used to link to the object containing the script that holds the method you want to use for the click event. Normally, you simply drop your scripts in the Canvas holding the button, then you drag the Canvas from the Hierarchy into the control (as shown above). This will populate the drop-down to the right with the methods in the script. Simply select the method to use.

Typically, you'll have only one handler for the Click event, but you can have as many as you'd like. They'll be called in the order they are added to the On Click section.

Text

The text control is a simple label that doesn't have any user interaction. We use this for the title of our main menu scene. You can change the text properties, as with the button in Figure 13.



Inspector

Text — Static
Tag: Untagged — Layer: UI

Rect Transform
center / top
Pos X: 0 — Pos Y: -40 — Pos Z: 0
Width: 400 — Height: 60
Anchors
Min: X 0.5 Y 1
Max: X 0.5 Y 1
Pivot: X 0.5 Y 0.5
Rotation: X 0 Y 0 Z 0
Scale: X 1 Y 1 Z 1

Canvas Renderer

Text (Script)
Text: Ghost Arena
Character
Font: Arial
Font Style: Normal
Font Size: 40
Line Spacing: 1
Rich Text: ✓
Paragraph
Alignment
Align By Geometry: ☐
Horizontal Overflow: Wrap
Vertical Overflow: Truncate
Best Fit: ☐
Color
Material: None (Material)
Raycast Target: ✓

Add Component

Figure 13: Text Properties

Panel

There will be times when you'll want to group UI elements together so that you can easily change them all at once. A panel allows you to do this. We use this to hold the controls that are displayed when the game is over, as shown in Figure 14.

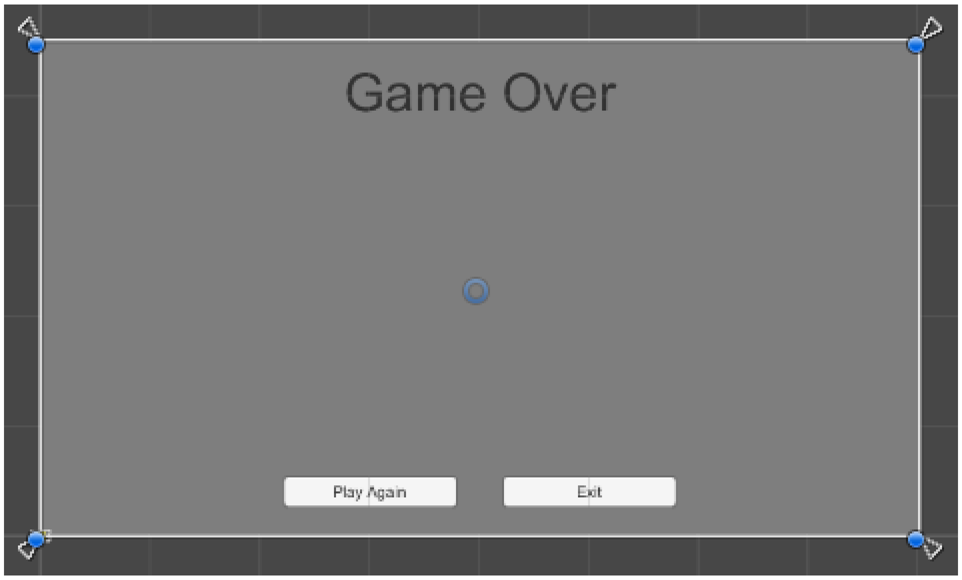


Figure 14: Game Over Panel

The panel is deactivated by default, which hides the buttons and Text object it contains. We'll also use a panel in the menu to allow the player to select the game's difficulty.

Input field

When you want to allow the user to enter text for items such as logging in, entering a name for a high score, etc., you'll want to use an input field, as in Figure 15.

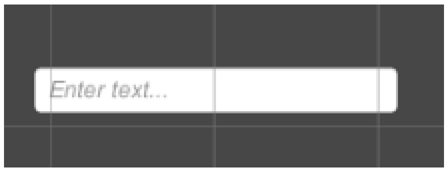
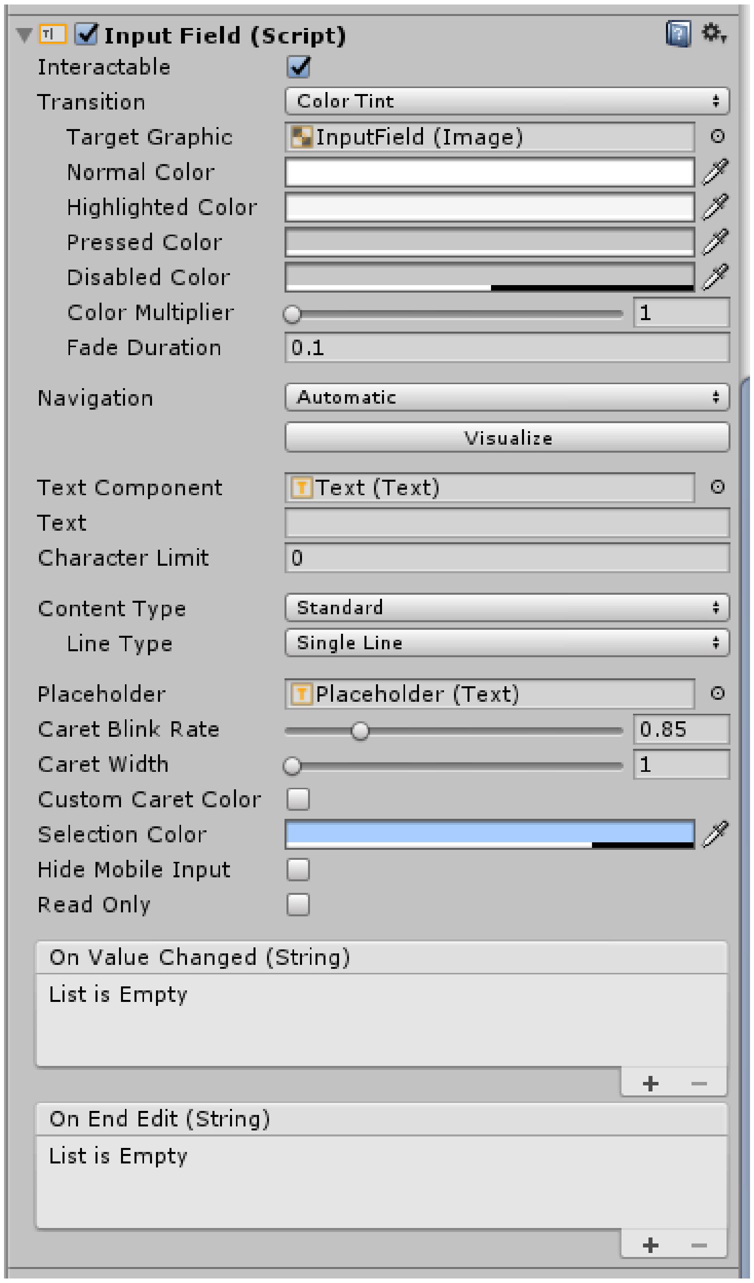


Figure 15: Input Field

This control has many more properties than we've seen before. Figure 16 shows the properties specific to the control.



Input Field (Script)

Interactable	✓
Transition	Color Tint
Target Graphic	InputField (Image)
Normal Color	
Highlighted Color	
Pressed Color	
Disabled Color	
Color Multiplier	1
Fade Duration	0.1
Navigation	Automatic
	Visualize
Text Component	Text (Text)
Text	
Character Limit	0
Content Type	Standard
Line Type	Single Line
Placeholder	Placeholder (Text)
Caret Blink Rate	0.85
Caret Width	1
Custom Caret Color	☐
Selection Color	
Hide Mobile Input	☐
Read Only	☐

On Value Changed (String)
List is Empty

On End Edit (String)
List is Empty

Figure 16: Input Field Properties

There are a number of other controls that you may find useful:

- Toggle (or check box)
- Slider
- Scrollbar
- Drop-down (or combo box)
- Image

If you have any app development experience, getting the hang of these controls shouldn't be too difficult.

Completing the game UI

Along with the buttons and Text object in the main menu, we also have elements for allowing the player to select the game's difficulty. These UI elements are in a panel that is hidden until the player starts the game.

If you've already added the Play and Quit buttons and text component for the Title to the main menu scene, add a panel and drag the buttons into it. If you haven't, now add a panel, then add the buttons to it. You should probably name the panel—as you should do with all your components. I've called it MainPanel.

Add another panel to the scene and name it DifficultyPanel. Next, add four more buttons. These will be for easy, medium, and difficult difficulty levels along with a cancel button. You may find it easier to work with the elements in this panel by hiding the main panel. Lay out the controls to look something like Figure 17, setting the text of the buttons as you go:

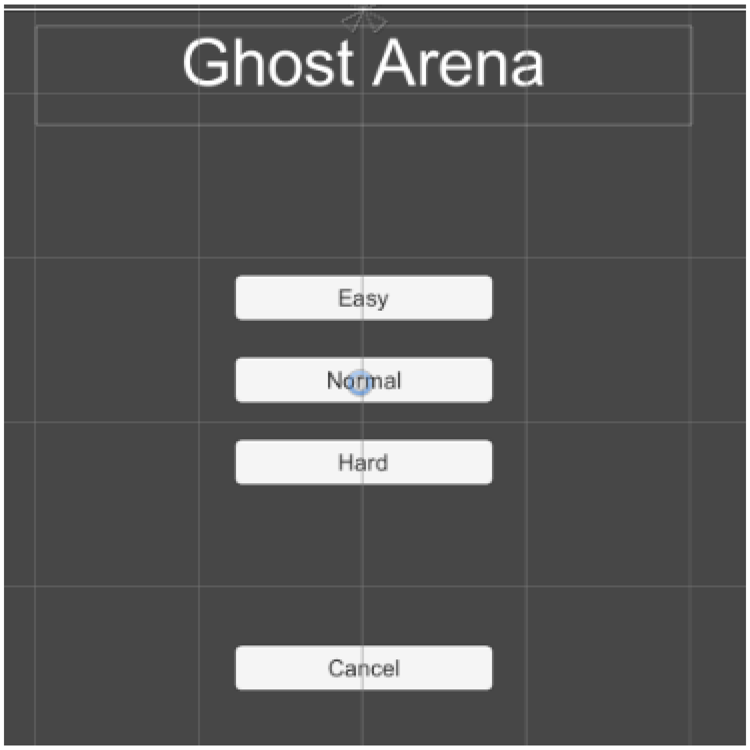


Figure 17: Main Menu Difficulty Controls

When you're done, you should have something that looks like Figure 18 (when both panels are visible).

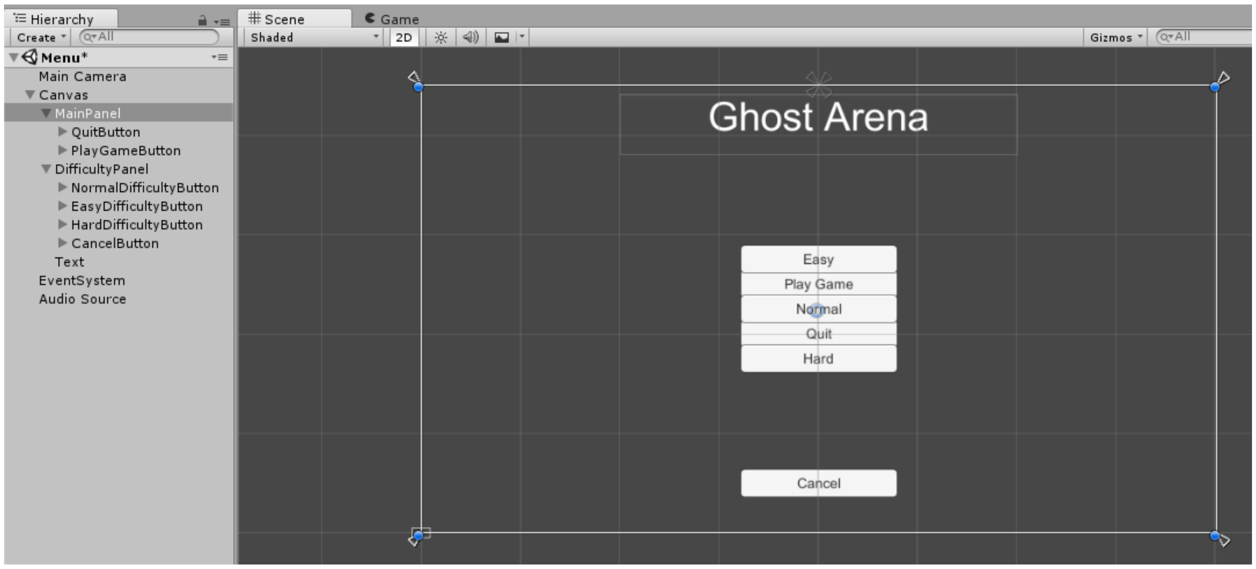


Figure 18*: Main Menu Scene*

The UI in the game scene isn't much more complicated than the menu. There are three text components and the Game Over panel that we saw earlier. Figure 19 shows the three text components.

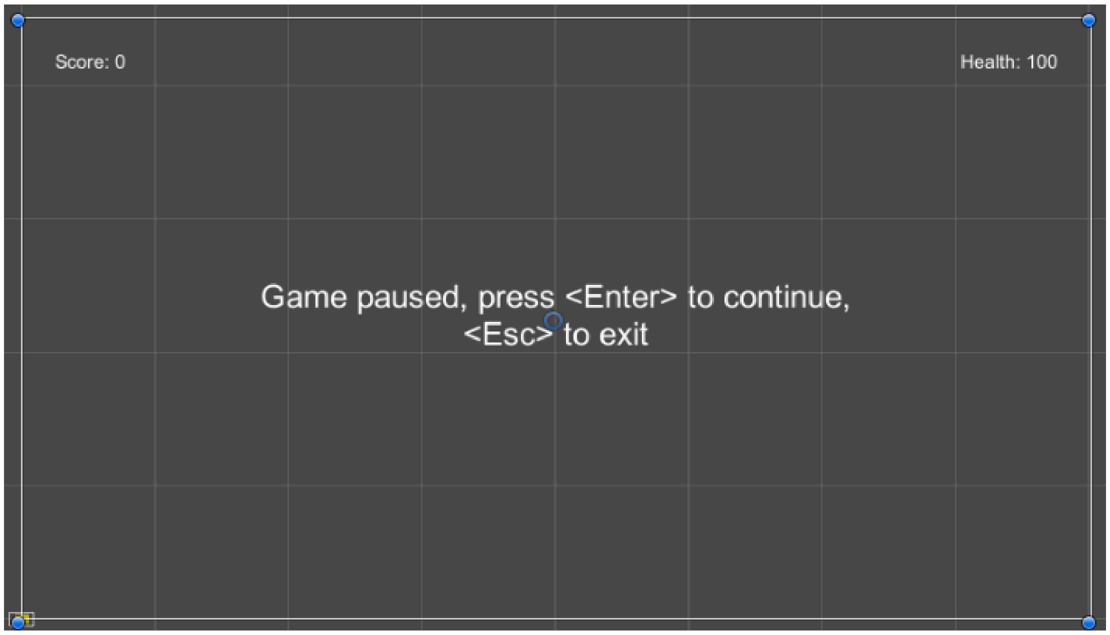


Figure 19: Game Scene Text Components

The pause text is set to a width and height of 500 and 60, respectively. The text font size is 24, horizontally centered, and the horizontal overflow is set to Wrap.

The score text is set to an X/Y position of 25/-25 with the anchor presets set to top left. The box in the upper-left portion of the Rect Transform set of properties is the anchor presets. Clicking on anchor presets will expand the box to show all the possibilities. After setting the X and Y values, hold the Shift and Alt keys while clicking on the top-left preset. The text will move into place.

The health text is set to -25/-25 with the anchor preset set to top right. Simply repeat the steps from the score text, clicking on the top-right preset.

The text for the Game Over is set to a Y value of 166, a width of 400, and a height of 75. The Play Again button has a width of 160 and a height of 30 and is set to an X/Y position of -100/25. The Exit button is set to an X/Y position of 100/25.

Coding the UI

The UI elements won't do anything without adding some code, so let's add it now.

Add a script to the scripts folder called Menu, drag that script from the Project to the Canvas in the Hierarchy, and open it in your code editor. Add the code from Code Listing 4 to the file.

Code Listing 4: Main Menu Code

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Menu : MonoBehaviour
{
      public GameObject MainPanel;
      public GameObject DifficultyPanel;

      public void PlayGameButton_Click()
      {
            DifficultyPanel.SetActive(true);
            MainPanel.SetActive(false);
      }

      public void CancelButton_Click ()
      {
            DifficultyPanel.SetActive(false);
            MainPanel.SetActive(true);
      }

      public void DifficultyButton_Click(int index)
      {
            Globals.DifficultyLevel = index;
            SceneManager.LoadScene("Game");
      }

      public void QuitButton_Click()
```

```
        {
                Application.Quit();
        }
}
```

If you look at the methods, you'll notice there is only one for the difficulty buttons. All three buttons will call this method. The index parameter will tell us which one was clicked. So, how do we get the button to pass something to the method? Let's go over that now.

Select all three of the buttons in the Hierarchy and add a handler to them by clicking **+** at the bottom-right of the On Click part of the buttons' properties in the Inspector. Drag the Canvas from the Hierarchy to the control at the bottom-left of the handler, where it should say "None (Object)." Select the **DifficultyButton_Click** method from the drop-down in the upper-right of the handler for all three buttons. At the bottom-right of the handler section, you will now see a text box in which you specify the parameter value to be passed to the method. Select each button in order—from easy to difficult—and set the values to 0, 1, and 2, respectively.

You can test to ensure that everything is working correctly by putting a breakpoint on the first line of the method and examining the value of the parameter. If you're using Visual Studio, this will require some extra steps to link it to Unity for debugging. If VS is set up correctly, you should see a button on the toolbar of the IDE with the text "Attach to Unity," as shown in Figure 20.

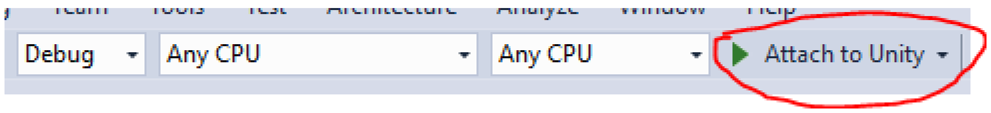


Figure 20: Visual Studio IDE Unity Debugging

If you didn't install Visual Studio Tools for Unity when you installed Unity, you can do so with relative ease. It's available as a separate install here.

The menu code uses a global variable to hold the selection of the difficulty. Create a new script file called Globals and add the following from Code Listing 5.

Code Listing 5: Difficulty Level Variable

```
public class Globals
{
    public static int DifficultyLevel;
}
```

The script for the game scene is in a script file called, unsurprisingly, Game. After creating the script, drag the icon for it in the Scripts folder to an object in the Hierarchy. I used the level sprite rather than the normal Canvas object simply because the level sprite is always displayed. You can use the Canvas if you wish. There's only a small amount of code in it that is UI related.

Code Listing 6: Game Scene UI Declarations

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;
```

```
public class Game : MonoBehaviour
{

    public GameObject GameOverPanel;

    public Text ScoreText;
    public Text HealthText;
    public Text PauseText;
```

The first using statement is added automatically when you create the script in the Unity IDE, but you'll have to add the other two manually in order for the code to change scenes back to the main menu.

The **`GameOverPanel`** member should be self-explanatory. Drag the panel from the Hierarchy window into its spot in the Inspector. The next three members are the Text objects for the score, player health, and pause texts. Drag them from the Hierarchy window into their spots in the Inspector.

All that's left is adding the **`Update`** method code and button handlers, as in Code Listing 7.

Code Listing 7: Game Panel Code

```
void Update()
    {

        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (Globals.CurGameState == GameState.PlayingGame)
            {
                Globals.CurGameState = GameState.PauseGame;
                PauseText.gameObject.SetActive(true);
            }
            else if (Globals.CurGameState == GameState.PauseGame)
                SceneManager.LoadScene("Menu");
        }

        if (Input.GetKeyDown(KeyCode.Return) && Globals.CurGameState ==
GameState.PauseGame)
        {
            PauseText.gameObject.SetActive(false);
            Globals.CurGameState = GameState.PlayingGame;

        }
    }

    public void ExitButton_Click()
    {
        SceneManager.LoadScene("Menu");
    }
    public void PlayAgainButton_Click()
    {
        GameOverPanel.SetActive(false);
        ResetGame();
    }
```

If the player presses the Esc key and the game isn't already paused, we set a switch variable to stop any game-related code from running. The Text object that lets the player know the game is paused is then shown. If the game is already paused, the Esc key exits the game and we load the Menu scene. If the Enter key is pressed while the game is paused, we unpause the game and hide the Pause Text object.

The Exit and Play Again buttons are displayed when the game is over. Clicking the Exit button loads the Menu scene. Clicking the Play Again button hides the Game Over panel and resets the game.

We now have the skeleton for our game. In the next chapter, we'll add the sprites so that the player will have something to look at.

Chapter 4 2-D Graphics and Sprites

No game can exist without a graphical component. Whether it's puzzle pieces, fantastical creatures, vehicles of some kind, or humans running around shooting each other, the graphical component is, without question, the most important aspect of a game. There's only one game that I can think of that had no graphical part to it—a game that utilized audio only. That's the exception to the rule, however. You'll need something for the player to see and manipulate in your games.

The object of our game will be the player running around in a room or arena being attacked by ghosts that will randomly come out of openings in the walls of the arena. If you think about it, you might easily surmise what the graphics pieces will be:

- The arena in which the player will be contained.
- The character the player will control.
- The ghosts.
- The bullets the character will fire.

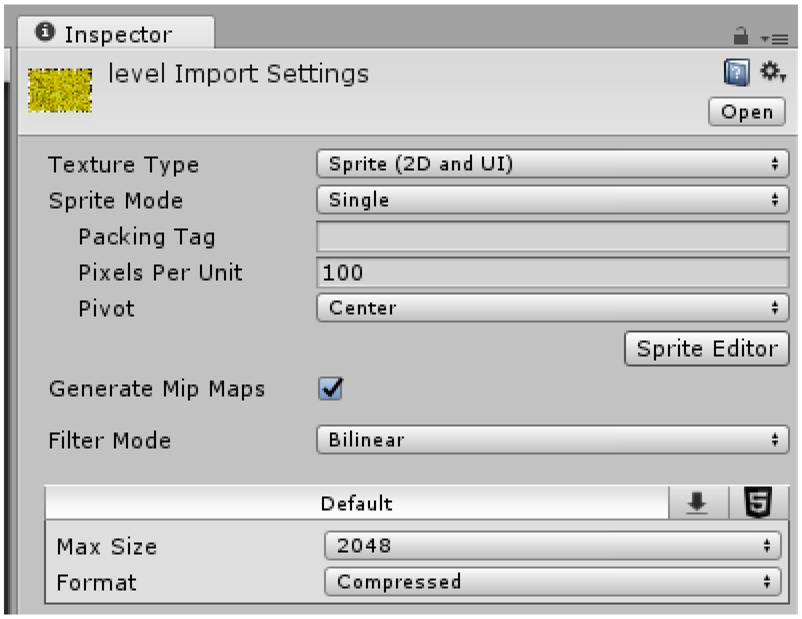
There are several other items we could add, but for a basic game this is all we'll need. Each one of these parts will be represented in the scene by a sprite.

Sprite

A sprite is simply a 2-D image with some added functionality in order to make it easy to manipulate and work with. The image is pulled from a graphics file (usually a .png or similar graphics format) and rendered to the screen by a SpriteRenderer. Note that the graphics file can contain multiple images, not only the one that will be used for the sprite. The images can be related or unrelated to each other in what's usually called an atlas, or they can be frames of an animation for one sprite. We put multiple images in one file in order to reduce memory cost (there could be a lot of lost memory in the area of the file that doesn't actually contain part of the image) or to make organizing your sprites easier. You could have a large sprite composed of multiple parts, some of which will be animated—having all of these parts in their own files might get confusing when you're trying to piece together the sprite.

Two of our sprites will have animations—the character and the ghost—and the other two will have only single images—the bullet and the arena.

We'll begin by implementing the arena sprite. First, add the level graphic file to the Sprites folder. Clicking the item in the Sprites folder should show that the image will be treated as a sprite, as shown below in the screenshot of the Inspector in Figure 21.


Inspector

level Import Settings

Open

Texture Type: Sprite (2D and UI)
Sprite Mode: Single
Packing Tag:
Pixels Per Unit: 100
Pivot: Center

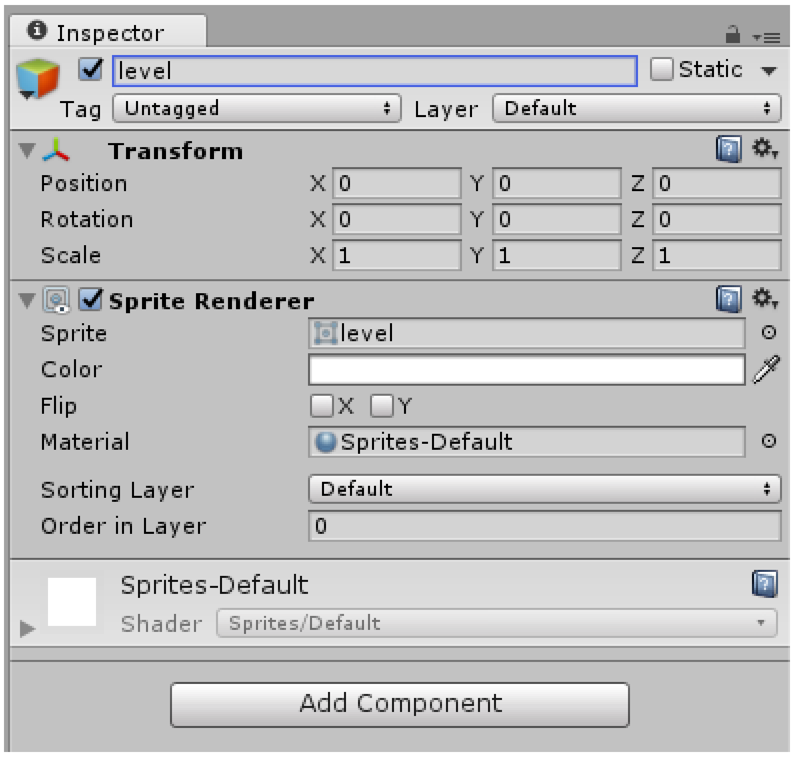
Sprite Editor

Generate Mip Maps: ✓
Filter Mode: Bilinear

Default
Max Size: 2048
Format: Compressed

Figure 21: Sprite Properties

Drag the sprite from the Sprites folder to the Hierarchy window. Figure 22 depicts what the Inspector will show.


Inspector

✓ level   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

✓ **Sprite Renderer**
Sprite: level
Color:
Flip: X Y
Material: Sprites-Default
Sorting Layer: Default
Order in Layer: 0

Sprites-Default
Shader: Sprites/Default

Add Component

Figure 22: Level Sprite Properties

The sprite is automatically positioned in the middle of the scene.

You can click on the Color bar and see what changing the RGB values will do.

The Flip X and Y values does what one would expect—it flips the sprite on the x-axis and y-axis.

The Material property of the SpriteRenderer isn't particularly relevant to us, and you shouldn't change the default value. Materials are more often used along with shaders in 3-D rendering to change how a 3-D object looks. For example, you can take a blank sphere and make it look like a baseball or basketball based on the material you apply to it.

The Sorting Layer and Order in Layer properties allow you to change the order in which the sprites are rendered from back to front. Sprites within the same layer are rendered lowest to highest, meaning a sprite with an Order in Layer of 0 will be covered by a sprite in the same location with an Order in Layer of 1. Sprites with the same value will be rendered in the order they were added to the scene.

Now that we have something for the character to run around in, we need to make sure he can't run out of it. If the character is able to run through the walls of the arenas, that would look strange. That's the behavior of the ghosts. The way we do this is with a Box Collider. This component will keep 2-D objects from moving beyond their bounds.

Select the **Level** object in the Hierarchy. In the Inspector, click **Add Component**. Select the **Physics 2-D** item, then the **Box Collider 2-D** item. Unity will try to fit the collider to the object it's placed on as best it can, but usually it must be edited in order to get the effect you want. Clicking **Edit Collider** in the Box Collider 2-D section of the Inspector will show green lines that define the limits of the box. The lines may be a bit hard to see at first. You can click on the check box of the SpriteRenderer section of the level in the Inspector to see them. Of course, you'll have to make sure the SpriteRenderer is checked so that you can see the image in order to align the collider correctly. You should see the squares in the middle of each line, even with the image being visible. Simply drag them so that the lines are on the inside of the walls of the arena, as shown by the red box in Figure 23.

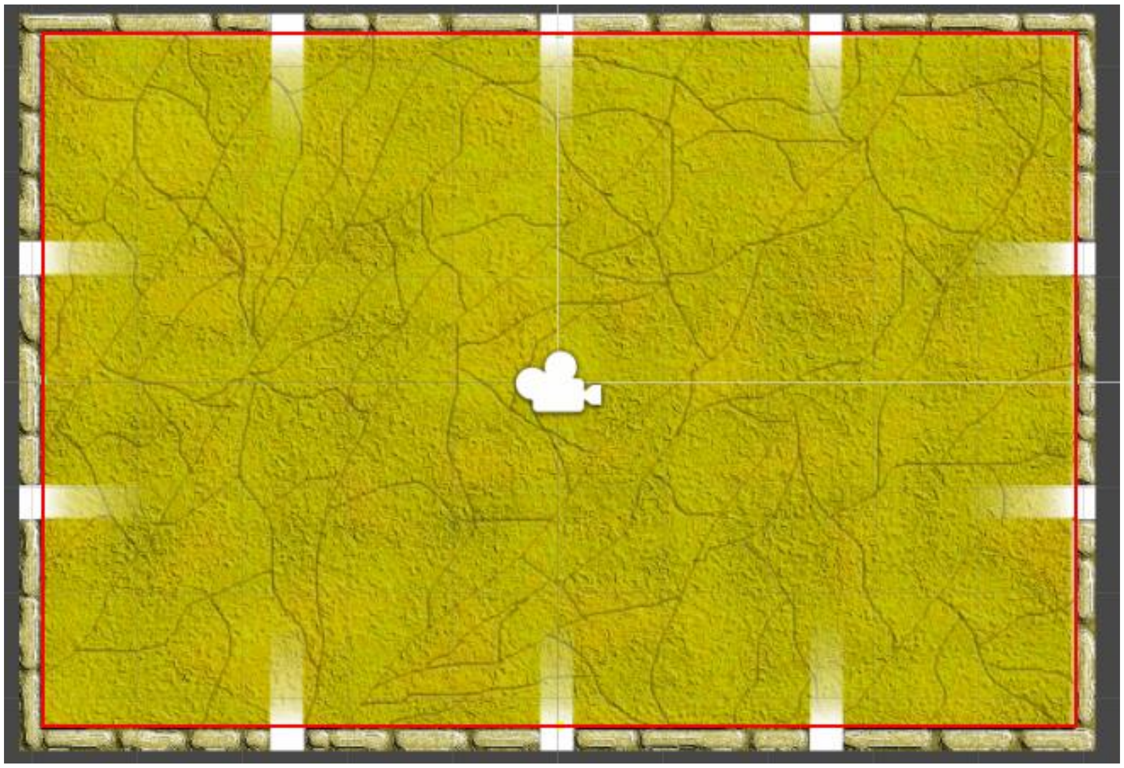


Figure 23*: Box Collider Placement*

When you have the collider in place, make sure you check the **Is Trigger** check box in the Box Collider 2-D section of the Inspector for the Level object. We need to do this because the collider not only keeps the character in the arena, it also keeps the ghosts out when we try to spawn them. The ghosts will spawn just outside of the 10 white sections that represent entrances into the arena. If we don't set the collider as a trigger, the ghosts will hit this when they try to move toward the player and will not be able to go further. With the collider set as a trigger, we can control what happens when an object hits the collider. We can let the ghosts pass through but restrict the character. We will do this a bit later when we add a script to the scene in order to control the player.

Bullet, character, and ghost Sprites

Now that we have an arena for the character to run around in, we need to get the character and ghost sprites into the scene. Add both files to the Graphics folder.

Click the player sprite. You'll also notice at the bottom of the Inspector that the file contains multiple images of the sprite, as shown in Figure 24.

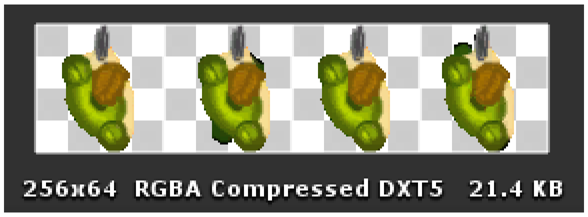


Figure 24: Player Sprite Frames

The slight differences are intended to show that the character is walking when it's moving around the screen, not simply floating or flying. The ghost sprite contains similar frames. Although it does float as it moves around, having only one frame makes things a bit boring. Having it animate as it moves makes the sprite look a bit more realistic. Getting the sprite to move through each of the frames over time is called animation. We'll cover how to do this a bit later, after we have the sprites set up correctly.

In order for the character and ghosts to interact with each other, we need to add a collider as well as one other Physics object. Instead of a Box Collider, however, we'll want to add a Polygon Collider 2-D. We do this in order to make the collision detection more accurate. Using a Polygon Collider is more computationally expensive, but in some cases it's much more accurate. If we were to add a Box Collider, there would be areas inside the box detected as a collision that aren't really collisions. Players will see that the collision detection doesn't work as it should, which will make your game look bad. Look at the ghost sprite in Figure 25.

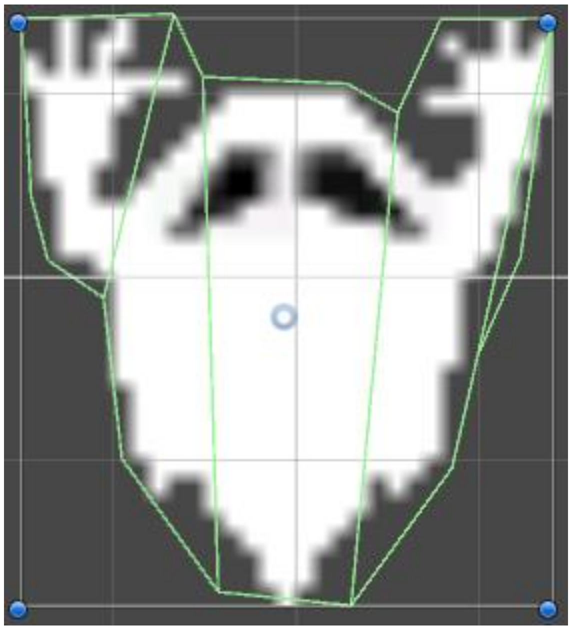


Figure 25: Ghost Sprite Box Collider vs. Polygon Collider

The blue dots are the area used for a Box Collider. The green lines show the area for a Polygon Collider. For the ghost, there will be areas at the top, middle, and bottom sides that will get incorrectly detected as a collision if we used the Box Collider. The player sprite is almost as bad, as we see in Figure 26.

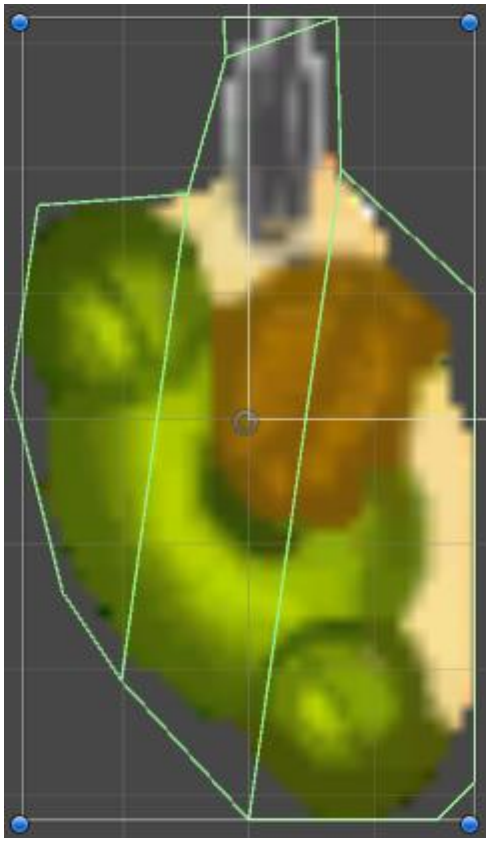


Figure 26*: Player Sprite Box Collider vs. Polygon Collider*

Use the Edit Collider button in the Inspector to tighten the polygon around each sprite. You can click the polygon between two points in order to add a point that allows you to create a tighter polygon. Along with the Collider component, we also need a Rigidbody 2-D component. We'll use this to help with setting up our movement. Add both of these components to the player and ghost sprites.

The character, ghost, and bullet sprites will work a bit differently than the level sprite. The level sprite stays on the screen and doesn't change. There can be multiple instances of ghost and bullet sprites on the screen at the same time, which means we'll set them up as prefabs in order to create the multiple instances easily. The character sprite will have only one instance on the screen, but the character can be killed by the ghosts, ending the game. We'll remove the sprite when this happens. If the player decides to play the game again, we'll recreate the sprite—we'll make it a prefab, too.

Creating a prefab from a sprite is very easy. Drag each sprite from the Sprite folder into the Hierarchy, then drag each one into the Prefabs folder. Delete the sprites from the Hierarchy and we're done.

Creating instances of the prefabs takes a bit more work than creating the prefabs themselves, but it's not very difficult. We'll deal with the character first. In the Game script file, add the code from Code Listing 8.

Code Listing 8: Player Creation Code

```
public GameObject Player;

private GameObject _player;

void Start ()
{

   _player = (GameObject)GameObject.Instantiate(Player, new Vector3(0, 0,
0), Quaternion.identity);
   ResetGame();
}

void ResetGame()
{
   _player.transform.position = new Vector3(0, 0, 0);

}
```

The public **Player** member will let us reference the character prefab in our code. The **Player** member is our instance of the prefab. When the scene loads, the **Start** method is called automatically. We then create the instance of the prefab, placing it in the center of the screen.

We'll have a bit more code that deals with the character instance, but we'll add that later. Let's move on to creating the instances of the ghost prefab. We'll need more code because we have multiple instances of the sprite, and they can appear in different locations at different times. Code Listing 9 shows how we'll handle this.

Code Listing 9: Ghost Creation Code

```
public GameObject Ghost;

private void SpawnGhost()
{

    int index = Globals.RandomNum.Next(Globals.StartingPoints.Length);

    Vector3 vec = Globals.StartingPoints[index];
    GameObject ghost = (GameObject)Instantiate(Ghost, vec,
Quaternion.identity);
    ghost.GetComponent<Ghost>().Player = _player;

    //rotate as necessary
    switch (index)
    {
        //rotate 180
        case 0:
        case 1:
        case 2:
```

```
        {
            ghost.transform.Rotate(0.0f, 0.0f, 180.0f);

            break;
        }
        // rotate 90 counterclockwise
        case 3:
        case 4:
        {
                ghost.transform.Rotate(0.0f, 0.0f, 90.0f);

                break;
        }
        // rotate 90 clockwise
        case 8:
        case 9:
        {
                ghost.transform.Rotate(0.0f, 0.0f, -90.0f);

                break;
        }
    }

}
```

Because there are 10 places a ghost can appear, we generate a random number that is an index into a list of possible start positions. These are determined simply by looking at the level sprite in the scene.

When we have the starting position, we create an instance of the ghost prefab and give it a reference to the instance of the player's character so that the ghost prefab can chase after the player's character.

Depending on the starting position of the ghost instance, we might need to rotate the ghost instance so that it's facing the opening it's supposed to come through.

This method relies on some global variables. Add the following code to the **`Globals`** class.

Code Listing 10: Ghost Creation Global Variables

```
public static System.Random RandomNum;


public static Vector3[] StartingPoints = new Vector3[] {
   new Vector3(-2.575f, 3.4f, 0f), new Vector3(0f, 3.4f, 0f), new
Vector3(2.575f, 3.4f, 0f),
   new Vector3(5.0f, 1.2f, 0f), new Vector3(5.0f, -1.14f, 0f),
   new Vector3(2.575f, -3.4f, 0), new Vector3(0f, -3.4f, 0f),new Vector3(-
2.575f, -3.4f, 0f),
   new Vector3(-5.0f, -1.14f, 0f), new Vector3(-5.0f, 1.2f, 0f)};
```

The random number variable will be initialized in the **Start** method of the Game script. Add the following code to the beginning of the method.

Code Listing 11: Random Number Initialization

```
if (Globals.RandomNum == null)
     Globals.RandomNum = new System.Random();
```

We have a character and ghosts to chase him, so now we need a way for the player to destroy the ghosts. Add the bullet graphic file to the Sprites folder, drag it into the Hierarchy window, then drag that into the Prefabs folder. Delete the bullet in the Hierarchy window, and we're done. Creating instances of the bullet prefab is done by responding to input from the player. Read on and we'll see how to do that.

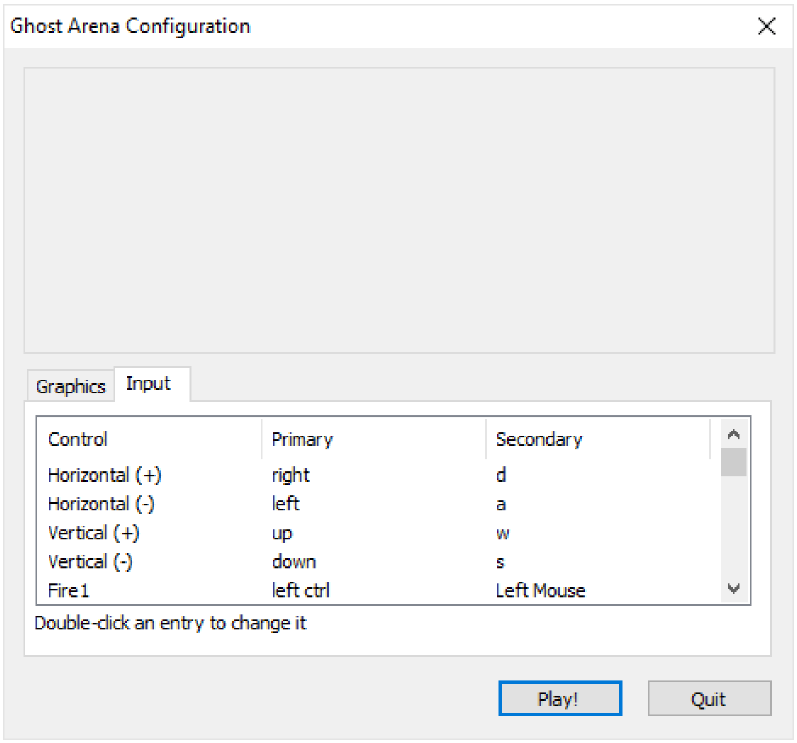
Chapter 5 Input

If the player can't control the game, we don't have much of a game. The main types of input your game must handle will be mouse and keyboard (PC), gamepad (PC and console), and touch (mobile). You might also consider voice and tilt inputs, although we won't cover those here. And, touch is difficult to implement correctly—it would require more explanation than we have room for here. If you're creating only mobile games, you'll want to look into both touch and tilt input methods. The **`Touch`**, **`TouchScreenKeyboard`**, and **`Gyroscope`** classes in the API will be good places to start.

Fortunately, Unity handles a lot of the input automatically. For PC games, the mouse works fine. No need to draw the pointer every frame and calculate its location. Unity also handles a lot of the UI work for you. Nearly all you need to do is write code for whatever a button does when you click on it, what happens when a check box is checked or unchecked, etc. We'll see that in the UI section.

Input Manager

Before dealing with getting input, we need to see how Unity handles peripheral configuration. The Input Manager tracks which keys/mouse control/etc. is used for actions in the game and allows the player to configure the input. The screenshot in Figure 27 shows the Input tab of the window being displayed when a game is run.

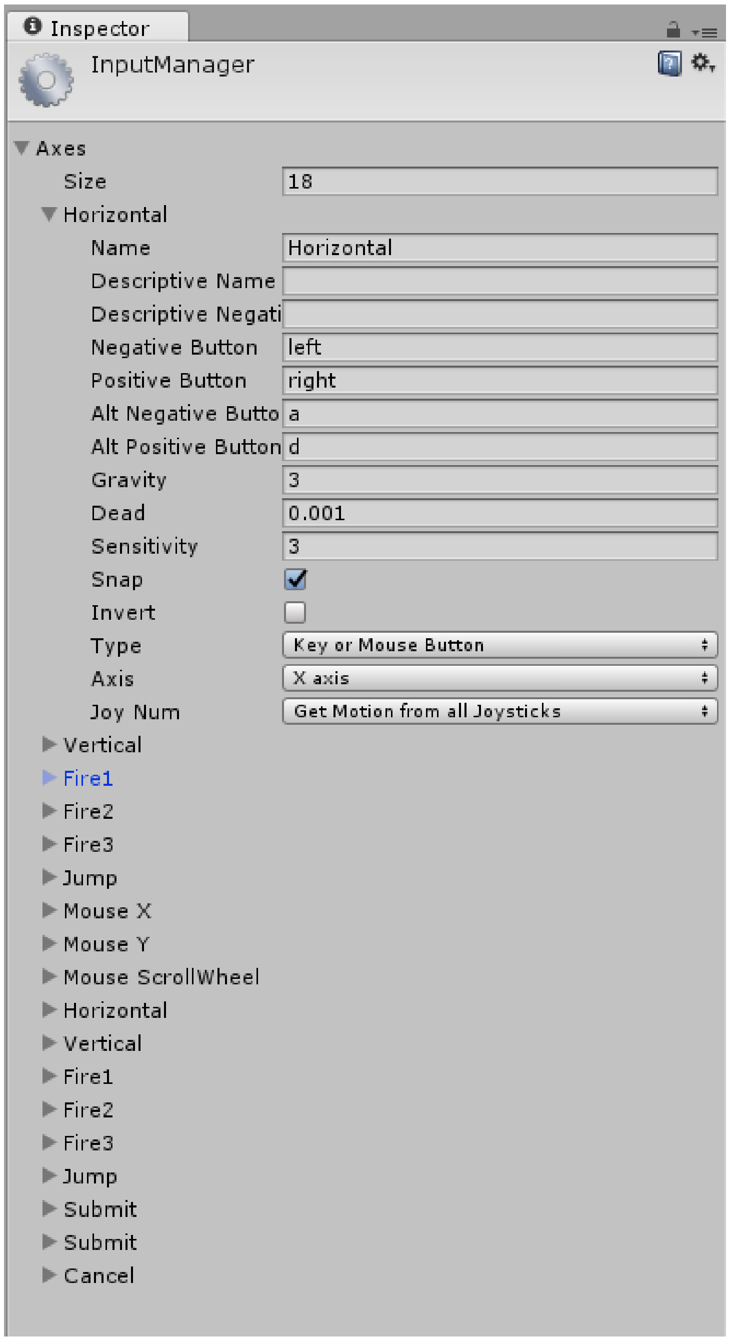


Control	Primary	Secondary
Horizontal (+)	right	d
Horizontal (-)	left	a
Vertical (+)	up	w
Vertical (-)	down	s
Fire1	left ctrl	Left Mouse

Double-click an entry to change it

Figure 27: Game Configuration Window

This window shows the input values that were set when the game was compiled. These values can be changed in the InputManager window in the Unity editor, as shown in Figure 28.


Inspector

InputManager

Axes

Size	18

Horizontal

Name	Horizontal
Descriptive Name	
Descriptive Negati	
Negative Button	left
Positive Button	right
Alt Negative Butto	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
Vertical
Fire1
Fire2
Fire3
Jump
Submit
Submit
Cancel

Figure 28: InputManager Window

Note that some actions, such as Horizontal, Vertical, Fire, Submit, and Jump, are listed multiple times. If you open all the versions of an action, you'll see that they are defined for multiple types of input devices. The first Horizontal type is Key or Mouse button—the left/right or a/d keys, as well as the x-axis on the mouse, will trigger horizontal movement. The second Horizontal type is Joystick Axis. You might wonder why they didn't simply define the Joystick type with the mouse. A PC will almost always have a mouse and keyboard, but a joystick is generally pretty rare.

In order to make this more confusing, take a look at the Fire1 actions. Both are listed as Key or Mouse button, one is left Ctrl/mouse 0 and the other is joystick button 0. This means that configuring your own actions will take some trial and error. Or you can just use the Get… methods of the Input class to check for specific inputs for specific actions and not allow the player to custom configure those actions. This will limit your game somewhat, however.

Input class

The main class you'll be dealing with is the **`Input`** class. Everything in the class is static, which means you don't need an instance of the class. For PC, the variables you'll be dealing with are **`inputString`**, **`mousePosition`**, and possibly **`anyKey`** and **`anyKeyDown`**. There are a number of methods in the class for dealing with mouse and keyboard (and joystick for those who still use them). Table 3 shows a partial list of input methods.

Table 3: Input Class Methods for PC

`GetAxis`	Returns the value of the virtual axis identified by axisName.
`GetAxisRow`	Returns the value of the virtual axis identified by axisName with no smoothing filtering applied.
`GetButton`	Returns true while the virtual button identified by buttonName is held down.
`GetButtonDown`	Returns true during the frame the user pressed down the virtual button identified by buttonName.
`GetButtonUp`	Returns true the first frame the user releases the virtual button identified by buttonName.
`GetJoystickNames`	Returns an array of strings describing the connected joysticks.
`GetKey`	Returns true while the user holds down the key identified by name. Think autofire.
`GetKeyDown`	Returns true during the frame the user starts pressing down the key identified by name.
`GetKeyUp`	Returns true during the frame the user releases the key identified by name.
`GetMouseButton`	Returns whether the given mouse button is held down.

`GetMouseButtonDown`	Returns true during the frame the user pressed the given mouse button.
`GetMouseButtonUp`	Returns true during the frame the user releases the given mouse button.

Fortunately, for our purposes we'll only need to use three methods and the **`mousePosition`** variable. We don't have any text input to deal with. If you want to enhance the game by adding a login for items such as a multiplayer mode and tracking high scores, you still won't necessarily need to use any more functionality in the **`Input`** class because the UI controls will be able to handle most of your needs.

We'll use the **`GetKeyDown`** method for determining when to pause and unpause the game. We'll use the **`GetAxis`** method for moving the player. We'll use the **`GetButton`** method for handling firing bullets. Lastly, we'll use the **`mousePosition`** for determining where the character is pointing in order to rotate the character sprite, which will affect where the bullet is fired.

In the Scripts folder, add a class that we'll use for controlling the character or player. I have called it PlayerController because that's the name you'll find in a lot of Unity samples and tutorials. Add the code from Code Listing 12.

Code Listing 12: PlayerController Class

```
public float Speed;
private float _fireRate = .5f;
private float _nextFireTime = 0.0F;

public GameObject Bullet;

private bool _bulletFired = true;

void OnTriggerEnter2D(Collider2D collider)
{
    if (Globals.CurGameState == GameState.PlayingGame)
    {
        if (collider.gameObject.tag == "Ghost")
        {
            Globals.Health--;

            Destroy(collider.gameObject);
        }
    }
}

void Update()
{
    if (Globals.CurGameState == GameState.PlayingGame)
    {
        Vector3 vel = new Vector3();
        vel.x = Input.GetAxis("Horizontal");
        vel.y = Input.GetAxis("Vertical");

        Vector3.Normalize(vel);
        vel *= Speed;

        float x, y;


        GetComponent<Rigidbody2D>().velocity = vel;


        x = Mathf.Clamp(transform.position.x, -4.5f, 4.5f);
        y = Mathf.Clamp(transform.position.y, -3.0f, 3.0f);

        transform.position = new Vector3(x, y, 0);


        //rotation
        Vector3 mouse = Input.mousePosition;
        Vector3 screenPoint = 
Camera.main.WorldToScreenPoint(transform.localPosition);
        Vector2 offset = new Vector2(mouse.x - screenPoint.x, mouse.y - 
screenPoint.y);
        float angle = Mathf.Atan2(offset.y, offset.x) * Mathf.Rad2Deg - 
90.0f;
        transform.rotation = Quaternion.Euler(0, 0, angle);

        if (_bulletFired)
        {
            _nextFireTime += Time.deltaTime;

            if (_nextFireTime >= _fireRate)
                _bulletFired = false;
        }

        if (Input.GetButton("Fire1") && !_bulletFired)
        {
            //spawn bullet
            GameObject bullet = (GameObject)Instantiate(Bullet, 
transform.position, transform.rotation);
            _nextFireTime -= _fireRate;
            _bulletFired = true;
        }

    }
}
```

The **`Update`** method will be called with every frame. We'll use the current mouse and keyboard values to determine where the character script will be drawn and which way it will face. The Horizontal and Vertical axes are used along with a constant speed, which is set in the Unity editor, and the current position of the sprite from the built-in Transform object to determine the new position. The **`mousePosition`** variable of the Input class is used along with the position of the sprite in order to determine how much the sprite needs to be rotated in order for it to face the mouse—so that the sprite will appear to be firing toward where the mouse is pointed.

The last action of the method will check to see if the player is pressing the control mapped to the Fire1 action. If so, and if the player is allowed to fire, a bullet is spawned and variables are set to ensure the player can't fire again before he's allowed.

The code uses a couple of globals that we need to add to the Globals script. The **`CurGameState`** and **`Health`** members go in the **`Globals`** class, with the enum outside of it.

Code Listing 13: Input-Related Global Code

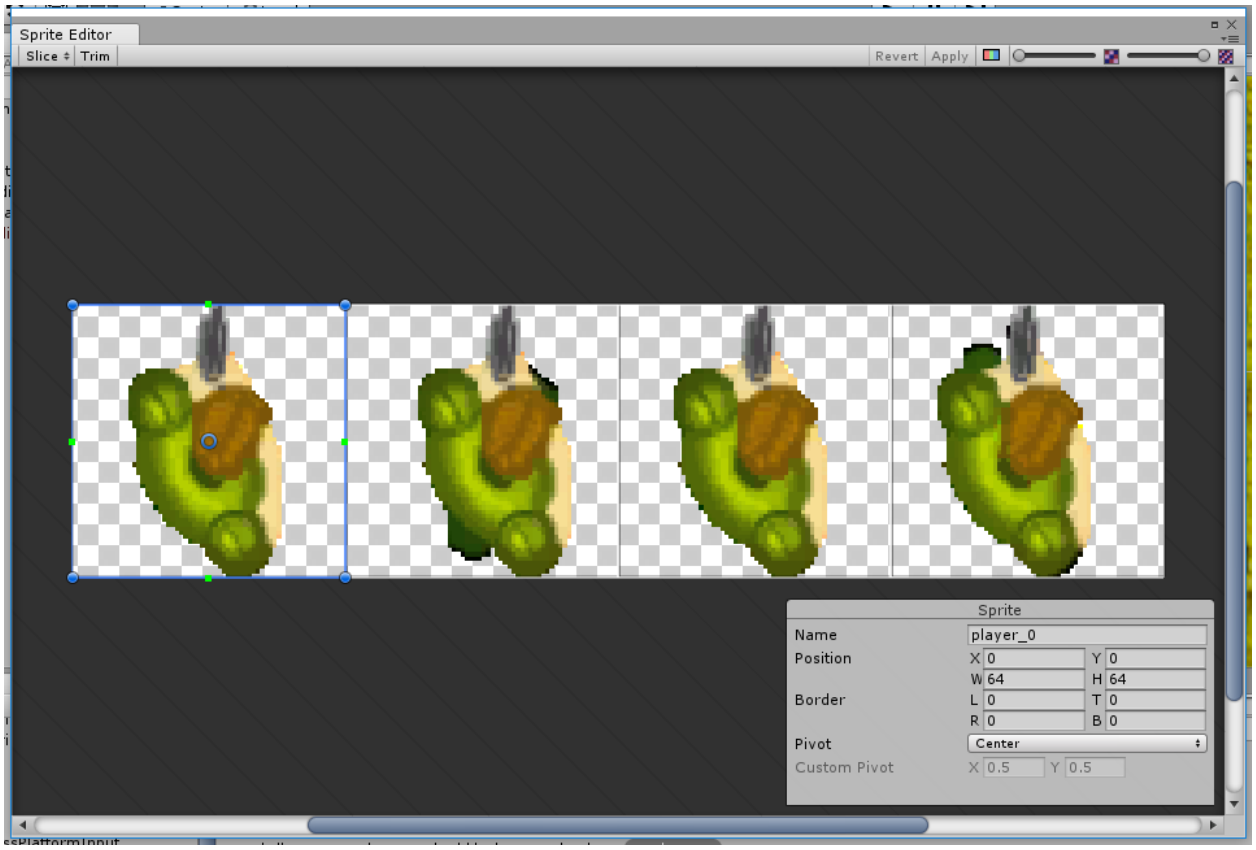
```
public static GameState CurGameState;
public static int Health;


public enum GameState
{
    PlayingGame,
    PauseGame,
    GameOver

}
```

You might want to consider allowing the player to use a gamepad in order to control the character. Even on a PC, some players prefer to use a gamepad for games. Allowing your game to handle a gamepad will also make it easier to convert to a console version.

Chapter 6 Animation

Now that we have our character moving around, let's next make him look like he's moving instead of merely floating (which is how it will currently appear). This is where sprite animation comes in.

First, we need to separate our character and ghost sprites into frames so that they'll animate properly. Select the player sprite in the Project window, then click **Sprite Editor** in the Inspector. You should see the screenshot in Figure 29.

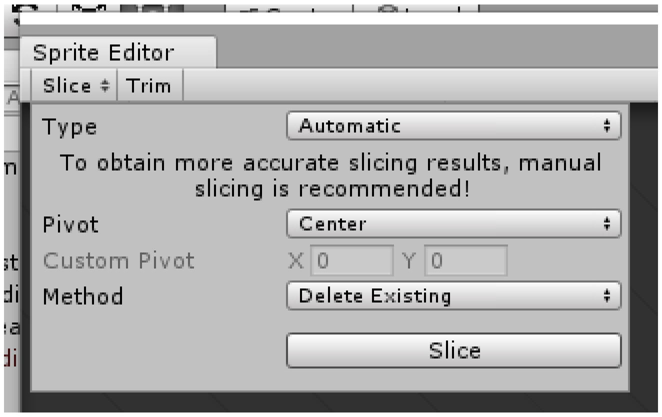


Sprite

Name	player_0
Position	X 0 Y 0
	W 64 H 64
Border	L 0 T 0
	R 0 B 0
Pivot	Center
Custom Pivot	X 0.5 Y 0.5

Figure 29: Sprite Editor Window

Use the **Slice** menu item to display the drop-down window and click **Slice**, as in Figure 30.


Sprite Editor

Slice Trim

Type: Automatic

To obtain more accurate slicing results, manual slicing is recommended!

Pivot: Center

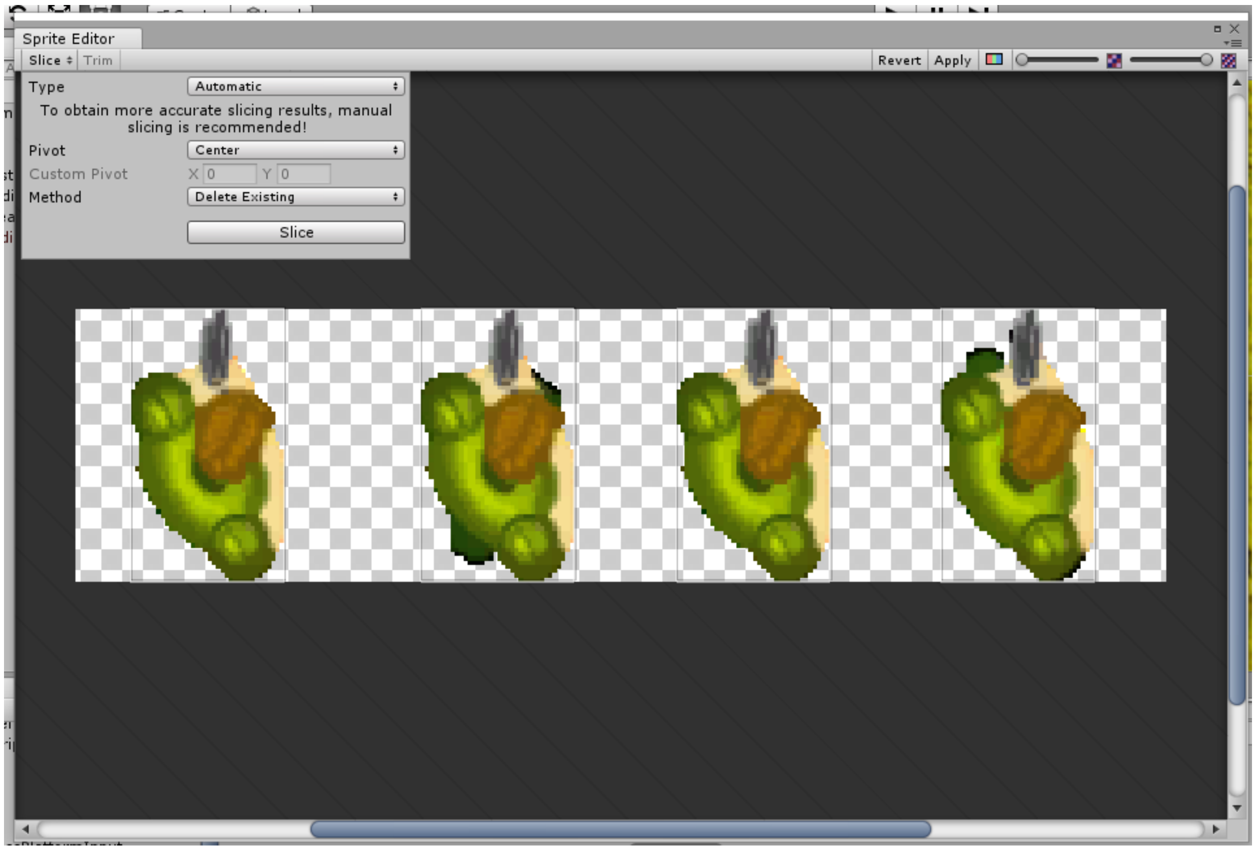
Custom Pivot: X 0 Y 0

Method: Delete Existing

Slice

Figure 30*: Sprite Editor Slice Window*

The image should show the sprite now sliced into four separate frames (you should see boxes around each image).



Type: Automatic

To obtain more accurate slicing results, manual slicing is recommended!

Pivot: Center

Custom Pivot: X 0 Y 0

Method: Delete Existing

Slice

Figure 31*: Sliced Sprite*

Close the Sprite Editor window and you should get a confirmation dialog that will apply the changes to the sprite. Click **Apply**, then select the ghost sprite in the Project window and perform the same steps.

Now that we have the frames for our animations, it's time to create the animation for each sprite. We want to apply the animation to our existing prefab, so let's drag the player prefab from its folder in the project into the Hierarchy window. If the frames for the player sprite in the Project window are not visible, click the icon on the right of the sprite in order to display them. Select the four frames of the player sprite (click the first frame and hold the **Shift** key while clicking the fourth frame) and drag them onto the player in the Hierarchy. You should get a Create New Animation file window. Create a new folder in the Assets folder called Animations and save the animation in the new folder. The Animator window should also open automatically. If it doesn't, open it from the Window menu. You'll see something like Figure 32.

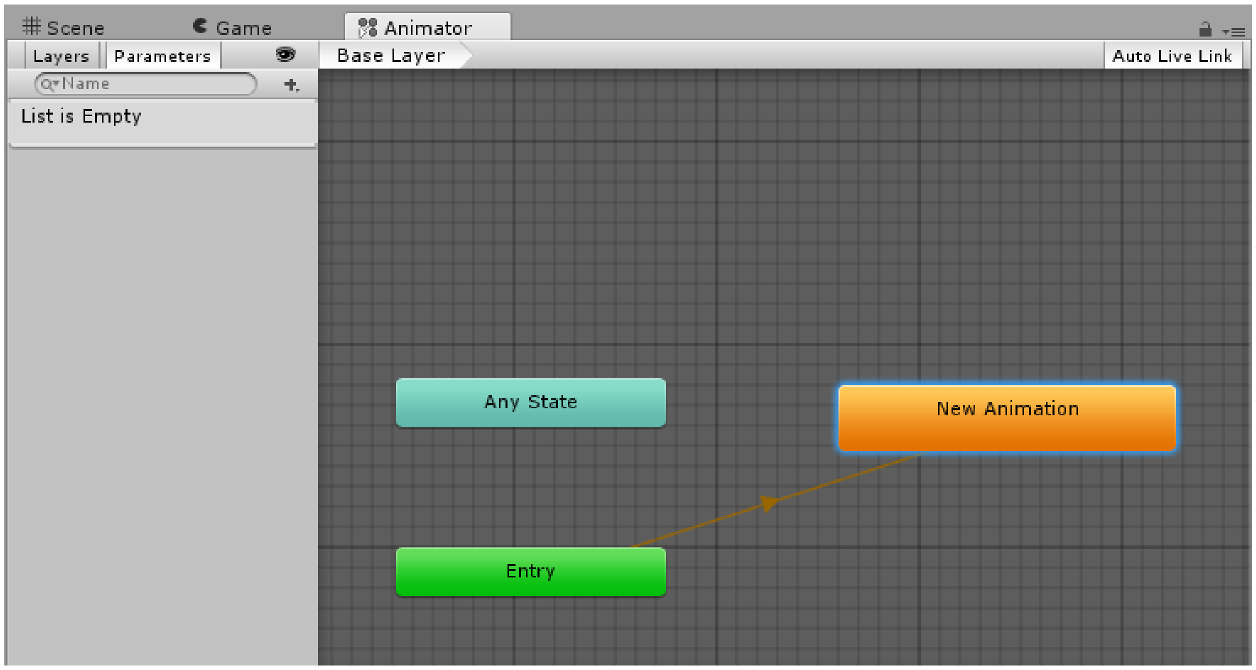


Figure 32: New Animation in Animator Window

With only the one state in the Animator, the animation will start immediately. While that will work fine for the ghosts, we don't want that for the player. We don't want the player to have the walking animation playing when standing still. In order to solve that, right-click in an empty area of the window and select the **Create State | Empty** menu item. A new state will be added with no animation. Rename this state Idle by clicking it and changing the name in the Inspector (make sure to press **Enter** after changing the name to ensure it actually changes), then right-click it and select **Set as Layer Default State**. You'll see the line move to the new Idle state from the Entry state. Rename the first animation Walking.

In order to change states, transitions need to be added between the Idle and Walking states. Transitions are used for indicating when to switch to a different animation. They have parameters that you set with code, and a change in the parameter will be detected automatically. The animation for that change becomes the current animation.

Right-click on the **Idle** animation and select the **Make Transition** menu item. Move the cursor to the Walking animation. As you do this, you'll notice that a line is attached to the cursor from the Idle animation. Click on the **Walking** animation and you'll have set a transition between the two. An arrow in the middle of the line shows which direction the transition will go. Right-click the **Walking** animation, then click on the **Idle** animation. You should now have transitions to and from both animations, as shown in Figure 33.

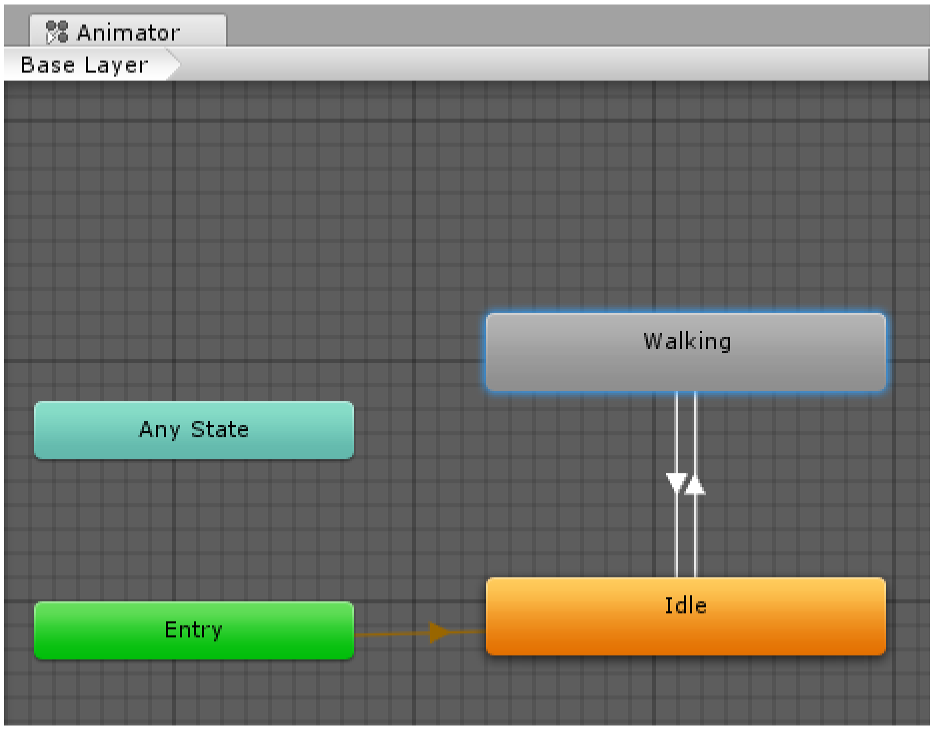


Figure 33: Walking and Idle Player Animations with Transitions

Now we need to set up the parameter for the transitions. Click the **+** to the left of the animation grid and select **Bool**, as in Figure 34.

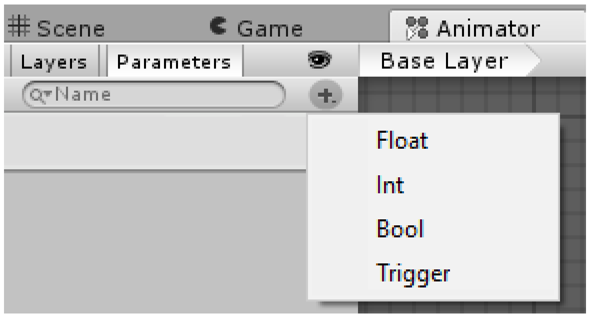
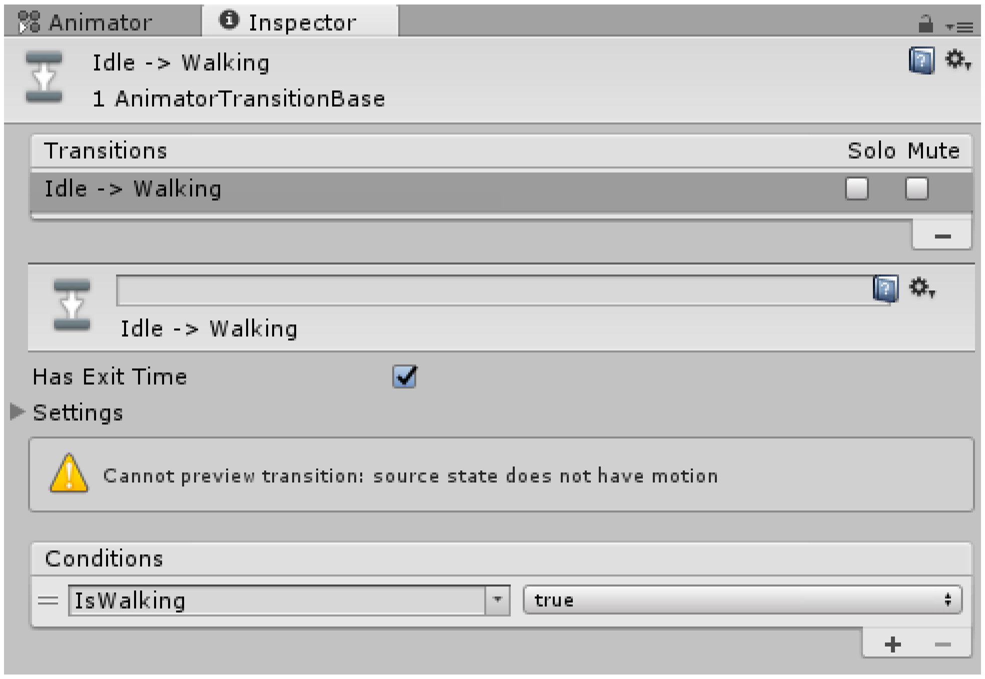


Figure 34: Addition Animation Transition Condition

Name the parameter IsWalking. Click on the transition line from Idle to Walking and you'll see the parameter in the Conditions part of the Inspector window. Set the drop-down item next to it to true. Click on the transition from **Walking to Idle** and set the IsWalking value to false, as in Figure 35.


Animator | Inspector

Idle -> Walking
1 AnimatorTransitionBase

Transitions	Solo	Mute
Idle -> Walking	☐	☐

Idle -> Walking

Has Exit Time ✓

Settings

Cannot preview transition: source state does not have motion

Conditions

IsWalking true

Figure 35: Animator Parameter Condition Setting

In order to save the changes to the prefab, we need to drag the Player object from the Hierarchy back onto the prefab in the Project window. Anytime you want to make changes to a prefab, you'll need to do this. Delete the Player object in the Hierarchy once you've saved the changes back to the prefab.

Now that we have everything set up in the Animator, we need only to add a few lines of code to change the value of the IsWalking parameter based on the input from the player. That's done in the **`PlayerController`** class **`Update`** method. After the line that sets the **`vel`** variable, add the following from Code Listing 14.

Code Listing 14: Animation Transition Code

```
if (vel != Vector3.zero && !CharacterAnimator.GetBool("IsWalking"))
    CharacterAnimator.SetBool("IsWalking", true);
else if (CharacterAnimator.GetBool("IsWalking") && vel == Vector3.zero)
    CharacterAnimator.SetBool("IsWalking", false);
```

The **`CharacterAnimator`** object gets added to the members of the class in Code Listing 15.

Code Listing 15: CharacterAnimator Declaration Code

```
    public Animator CharacterAnimator;
```

Select the player prefab and drag the Animator into the CharacterAnimator member of the script. The script will now update the Animator, which will trigger the animations when the player moves the character around the screen, and you should see the animation in the Game window while holding down the movement keys. The animation should stop when the character isn't moving.

If you select the Player object in the Hierarchy while the game is running and the Animator window is open, you should see the Parameter being changed and the animations being triggered, as in Figure 36.

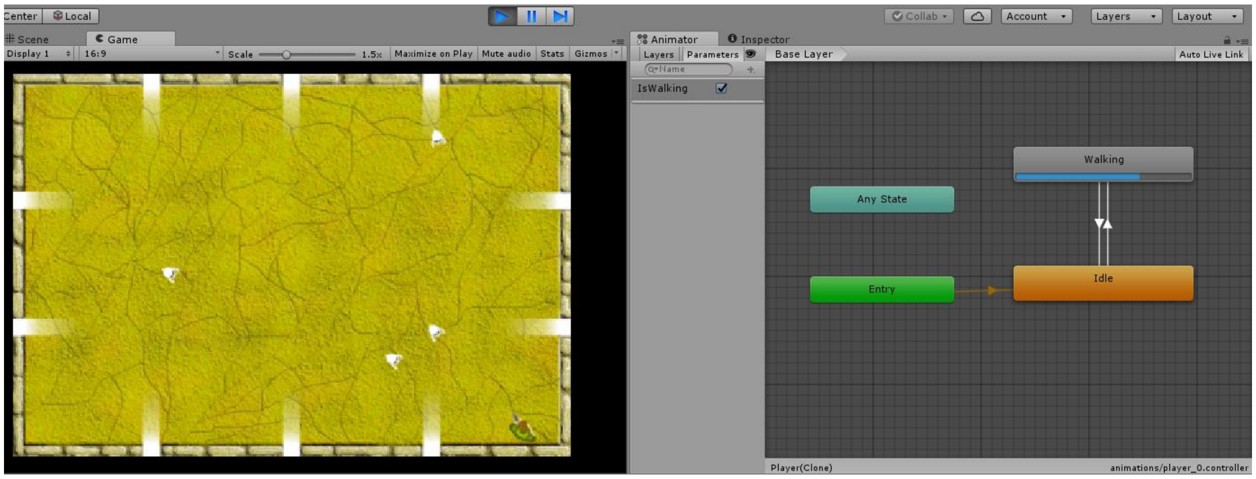


Figure 36: Animator Updates While Running Game

This takes care of the player animation. We need to do the same thing with the ghost. Fortunately, the ghost is a little easier—there's no need for any conditions to the ghost animation and no need to have an Idle state. The ghost will be moving constantly.

Drag the ghost prefab back into the Hierarchy, select the frames of the ghost animation, and drag that onto the Ghost object. Save the animation, drag the Ghost object back onto the prefab, and you're done. Every ghost that spawns will be animated.

With the addition of animation, the game is a step toward being more professional, but it still doesn't include a feature that you might have noticed is missing—sound.

Chapter 7 Audio

Very few games are complete without some kind of sound effects, background noises, or music. Even the simplest of games like *Bejeweled* or *Tetris* have sounds. Many AAA games have such good sounds that if you closed your eyes, you might think the game was the real world.

At its simplest level, sounds in Unity involve two objects—the AudioClip and the AudioSource.

The AudioSource has a ton of properties for dealing with sounds in both 2-D and 3-D games, but we don't need to use any of them. AudioSource is simply a container for the audio file we need to play.

We'll have four different audio files that we'll be playing—one music file that will play in the menus and three sound effects in the game. You can use either the audio files provided for this chapter or use your own files. Unity supports the .aif, .wav, .mp3, and .ogg audio file formats.

Add the files to the Audio folder. In the Menu scene, right-click the Hierarchy and select **Audio | Audio Source**. Drag the file "warg_-_kibelebassrif.wav" onto the AudioClip property of the AudioSource. Make sure the Play On Awake and Loop check boxes are checked. The Play On Awake property starts the file playing when the scene starts. While the scene is open, the Loop property will restart the file each time it finishes. None of the other check boxes should be checked. The Mute check box would cause the music to mute, and the other check boxes are for functionality that's not used.

After adding the AudioSource and selecting it, you'll notice that there is a circle in the scene that encompasses the entire Canvas, as shown in Figure 37.

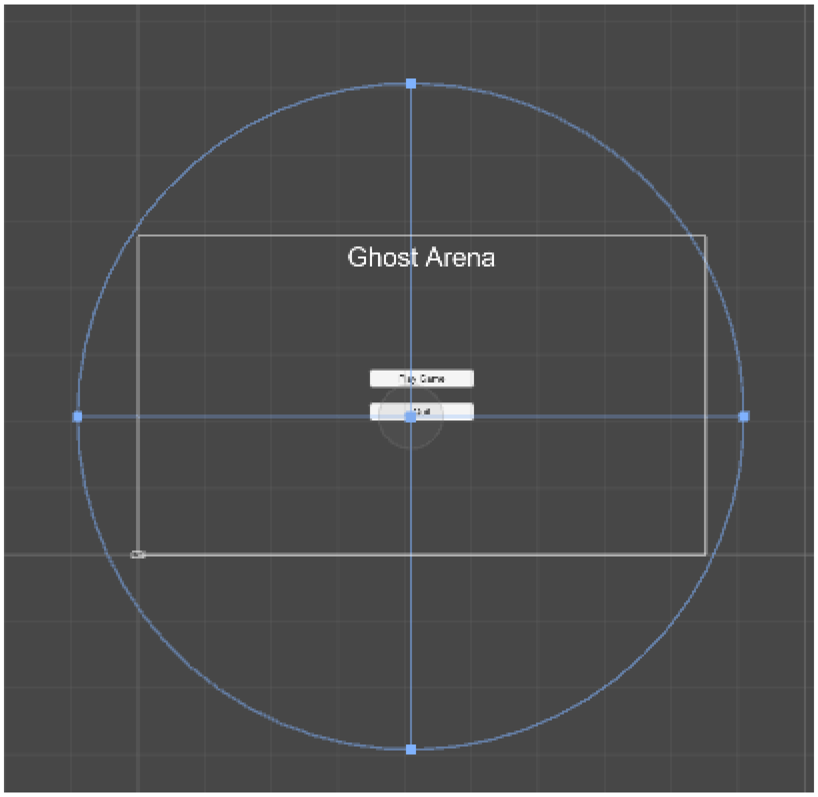


Figure 37: Audio Source Area

The circle defines the area in which the sound will be heard for a 3-D game. If you open the 3-D Sound Settings section of the AudioSource in the Inspector, you'll see the settings that define how the sound will be heard in a 3-D game. For our purposes, they don't matter. The sound will be heard no matter what.

Even the settings that can affect the sound in a 2-D game, which you'll find above the 3-D Sound Settings section, will rarely need modification. Perhaps you might change the Volume setting if you want to give the player the option of disabling sounds or modifying the volume.

For the main part of the game, we'll need three AudioSource objects. Change the scene to the Game scene and add them. Unlike with the menu, you'll want to change the names of the objects. I used BulletAudioSource, GhostAudioSource, and PlayerDeathAudioSource. Drag the appropriate sound files to the AudioSource objects. None of these will be set to loop or play immediately. We'll control when they will play with script.

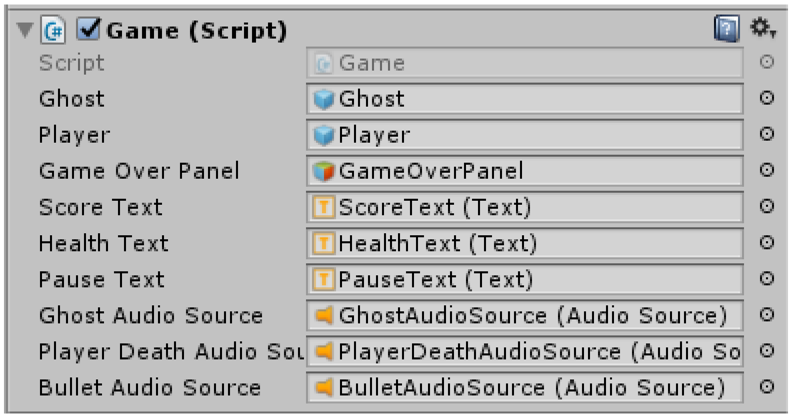
Note: Playing sounds from a prefab does not work.

Now that we have the AudioSources added to the scene, we need to modify the Game script in order to recognize the sources. Add the code from Code Listing 16 to the declarations section.

Code Listing 16: Game Audio Declarations

```
    public AudioSource GhostAudioSource;
    public AudioSource PlayerDeathAudioSource;
    public AudioSource BulletAudioSource;
```

Save the script and go back to the Unity IDE. Select the object to which you've added the Game script and you should see the three AudioSource objects that you just added in the Inspector window. Drag each AudioSource object from the Hierarchy window into its corresponding object in the Inspector, as shown in Figure 38.



Game (Script)

Script	Game
Ghost	Ghost
Player	Player
Game Over Panel	GameOverPanel
Score Text	ScoreText (Text)
Health Text	HealthText (Text)
Pause Text	PauseText (Text)
Ghost Audio Source	GhostAudioSource (Audio Source)
Player Death Audio Sou	PlayerDeathAudioSource (Audio So
Bullet Audio Source	BulletAudioSource (Audio Source)

Figure 38: Audio Source Script Members

Next, add the following line to the Globals script.

Code Listing 17: Global Bullet Audio Source Declaration

```
public static AudioSource BulletAudioSource;
```

Now add the following line after the **`If`** statement in the **`Start`** method.

Code Listing 18: Bullet Audio Source Assignment

```
Globals.BulletAudioSource = BulletAudioSource;
```

Because the code that spawns the bullets is in a different script file, we'll need to get to the AudioSource object in the code. Making it a global will allow us to do that. In order to actually play the bullet sound, insert the following code at the beginning of the **`If`** statement at the end of the **`Update`** method in the PlayerController script.

Code Listing 19: Playing Bullet Audio

```
Globals.BulletAudioSource.Play();
```

Next, we'll handle the ghost spawning. In the Game script at the end of the **`SpawnGhost`** method, add the code from Code Listing 20.

Code Listing 20: Playing Ghost Audio Logic

```
//If hard difficulty, only play sound every other time.
        if (Globals.DifficultyLevel == 2)
            _playSound = !_playSound;

        if(_playSound)
            GhostAudioSource.Play();
```

Playing a sound when the ghost spawns is a nice touch, but you must always think about what can happen when a sound plays. In this case, with the different difficulty levels, the ghost could spawn at different times. In the case of the hard difficulty level (DifficultyLevel == 2), that means every half-second. If you haven't already tried the game without the code, do so now. You'll see the problem almost immediately. The sound plays a bit too often, playing over itself. This will get annoying very quickly. The code we add will play the spawn sound every other time a ghost spawns, which makes it a bit more tolerable. You could change the frequency so that the sound plays even less, but that will mean a bit more code and you'd have to change from using a Boolean to track when the sound plays. Not difficult, but I chose to go the easy route. Feel free to modify the code to limit the sound further if you prefer.

The above code uses a switch variable we need to declare. Put the following code at the end of the declarations section of the **`Game`** class.

Code Listing 21: Ghost Sound Switch

```
private bool _playSound;
```

We'll need to set the variable to true in the **`ResetGame`** method. We can't do it in the declaration because the value might end up being false if the game is played on the hard difficulty level. Add the following code to the end of the method.

Code Listing 22: Ghost Sound Switch Initialization

```
_playSound = true;
```

The last sound is played when the character dies as a result of his health reaching 0. Insert the following code at the beginning of the **`Update`** method.

Code Listing 23: Playing Code for Player Death Sound

```
//PlayerDeathAudioSource.Play();
```

We will add logic late in order to figure out when to play this sound, but we're adding the placeholder now so that we'll know where the logic goes.

That's all the code we need to play our sounds. These are very easy sounds that don't require a lot of logic to handle. You can add sounds that play when they hit a collider. You can trigger sounds with other types of logic. There are a number of other scenarios in which playing sounds can happen. Feel free to experiment. One idea is to play sounds when you click UI buttons, for example. The source code contains a sound file you might use for that purpose.

Chapter 8 Implementing Gameplay

We're just about done with the game. We have only a little more code to add in order to make it playable.

We currently have no logic to handle the bullet that destroys a ghost. A simple class will take care of that. Add a new class called **`PlayerBullet`** to the scripts file and enter the following code.

Code Listing 24: PlayerBullet Class

```
public class PlayerBullet : MonoBehaviour
{

    void OnTriggerEnter2D(Collider2D collider)
    {
        if (Globals.CurGameState == GameState.PlayingGame)
        {
            if (collider.gameObject.tag == "Ghost")
            {
                Globals.Score += (Globals.DifficultyLevel + 1) * 5;
                Destroy(collider.gameObject);
                Destroy(gameObject);
            }
        }
    }

    void Update ()
    {
        if (Globals.CurGameState == GameState.PlayingGame)
        {
            transform.Translate(Vector2.up * 0.1f);
            //destroy if at arena wall
            if (transform.position.x >= 5 || transform.position.x <= -5 ||
                transform.position.y >= 3.4 || transform.position.y <= -
3.4)
            {
                Destroy(gameObject);
                //Debug.Log("destroying bullet");
            }
        }
    }
}
```

Select the **Bullet** prefab, click the **Scripts** folder, and drag the new script into the Inspector.

The **OnTriggerEnter2D** method is built into Unity for GameObjects that have a 2-D collider attached to them. We simply add the logic we need. If the game hasn't been paused and the bullet has hit a ghost, we add to the global score variable and destroy both the ghost and bullet instances.

In the **Update** method, if the game hasn't been paused, we move the bullet appropriately for the elapsed time since the last frame. If it has hit a wall of the arena, we destroy the instance of the prefab.

Next, the ghost needs some code to handle moving it toward the player. Add a new script called Ghost and add the following code.

Code Listing 25: Ghost Class

```
public class Ghost : MonoBehaviour
{
    public GameObject Player;

    void Start ()
    {
    }

    void Update ()
    {
        if (Globals.CurGameState == GameState.PlayingGame)
        {
            float step = Time.deltaTime;
            transform.position = Vector3.MoveTowards(transform.position,
Player.transform.position, step);

            Vector3 player =
Camera.main.WorldToScreenPoint(Player.transform.position);
            Vector3 screenPoint =
Camera.main.WorldToScreenPoint(transform.localPosition);
            Vector2 offset = new Vector2(player.x - screenPoint.x, player.y
- screenPoint.y);
            float angle = Mathf.Atan2(offset.y, offset.x) * Mathf.Rad2Deg -
90.0f;
            transform.rotation = Quaternion.Euler(0, 0, angle);
        }
        else if (Globals.CurGameState == GameState.GameOver)
            Destroy(gameObject);
    }
}
```

If the player hasn't paused the game, we move the ghost toward the player based on the elapsed time since the last frame, rotating the sprite if necessary. If the game is over, we destroy the ghost.

We will need a few more global variables.

Code Listing 26: Remaining Global Variables

```
public static float[] SpawnTimes = new float[] { 2, 1, 0.5f };

public static int Score;
```

Lastly, the **`Game`** class needs a bit more code.

Code Listing 27: Game Timer Member

```
private float _timer;
```

The timer is used to determine if a ghost is to be spawned. We'll initialize it in the **`ResetGame`** method along with a couple of other variables, as shown in Code Listing 28.

Code Listing 28: ResetGame Method Code

```
Globals.Health = 100;
Globals.Score = 0;
Globals.CurGameState = GameState.PlayingGame;
_timer = 0.0f;
```

Code Listing 29 needs to be added to the beginning of the **`Update`** method, replacing the commented-out code to play the sound when the character dies.

Code Listing 29: Game Update Code

```
if (Globals.CurGameState == GameState.PlayingGame)
{
    _timer += Time.deltaTime;

    if (_timer >= Globals.SpawnTimes[Globals.DifficultyLevel])
    {
        SpawnGhost();

        _timer -= Globals.SpawnTimes[Globals.DifficultyLevel];
    }

    ScoreText.text = "Score: " + Globals.Score.ToString();
    HealthText.text = "Health: " + Globals.Health.ToString();

    if (Globals.Health == 0)
    {
        PlayerDeathAudioSource.Play();
        Globals.CurGameState = GameState.GameOver;
        //kill player, game over
        GameOverPanel.SetActive(true);
    }
}
```

If the player hasn't paused the game, we increment the timer variable with the elapsed time since the last frame. If that time is equal to or greater than the spawn time for the selected difficulty, we call the **`SpawnGhost`** method and decrement the timer by the time amount of that spawn.

We set the score and health UI objects to their current amounts. Because the score and health are changed in other scripts, we must set global variables that we access from the game script.

If the character has been killed, we play his death sound, change the current game state so that other code will know not to run (ghosts spawning and moving, for example), and show the panel notifying the player that the game is over.

Next steps

While we have a playable game at this point, some things could be added to make it better. We'll cover a few of them here. I'm sure everyone can come up with their own unique features—that's part of what makes indie game development so great. There are as many ideas as there are game developers.

Leaderboards

In any game that keeps scores, leaderboards are a way for players to measure their progress against themselves and other gamers. Many games with leaderboards will have multiple ways of tracking scores and other information. Even a game as small as ours has several different types of information we can track:

- Most ghosts destroyed
- Longest survival time
- Highest accuracy percentage
- Lowest ghost survival average

You will notice that some of these types of information will require additional code because the information isn't currently tracked. You'll probably want to make that information global. Doing so will enable you to keep the code to save and retrieve the information separate from the main game code. Separating functionality this way is almost always a good idea.

Despite the different data types, we'll use one class to handle all of them.

Code Listing 30*: LeaderboardData Class*

```
public enum LeaderboardType
{
    GhostsDestroyed,
    PlayerSurvivalTime,
    Accuracy,
    GhostSurvivalTime
}

public enum LeaderboardAmountType
{
    FloatAmount,
    IntAmount
}

public class LeaderboardData
{
    public string PlayerName;
    public LeaderboardType Type;
    public LeaderboardAmountType AmountType;
    public float FloatAmount;
    public int IntAmount;
}
```

Because we have a distinct value for each leaderboard, we can use a **Dictionary** to hold the data for all of the leaderboards.

Code Listing 31: Leaderboards Dictionary

```
public static Dictionary<LeaderboardType, List<LeaderboardData>>
Leaderboards;
```

In order to track the information for these leaderboards, you'll need to implement some kind of system to recognize when the events for the leaderboards occur. You could hard code everything by keeping global variables and updating them from whatever code is applicable, which is quick and easy, but it's not very flexible.

Another method would be to implement events in the classes in which the data for leaderboards gets updated while keeping track of the data in a more central part of your code, such as a **LeaderboardsManager** class. This takes a little more time, but adding leaderboards will be much quicker to implement and all of your code will be in one central location, which means updating it is much easier.

For the Ghosts Destroyed leaderboard, for example, all you need is a few lines of code in the **PlayerBullet** class.

Code Listing 32: Ghosts Destroyed Leaderboard Event

```
    public delegate void GhostDestroyedEvent();
    public static event GhostDestroyedEvent GhostDestroyed;

//Add this after the Destroy(gameObject) line in the OnTriggerEnter2d
method.
GhostDestroyed();
```

The Player Survival Time leaderboard requires an update to the **PlayerController** class.

Code Listing 33: Player Survival Leaderboard Event

```
    public delegate void PlayerSurvivalTimeEvent(float time);
    public event PlayerSurvivalTimeEvent PlayerSurvival;

    private float _aliveTime;

//Add this as the last line in the inner If statement in the OnTrigger2D
method.
if (Globals.Health <= 0)
    PlayerSurvival(_aliveTime);

//Add this as the first line in the If statement in the Update method.
_aliveTime += Time.deltaTime;
```

The Accuracy leaderboard requires an update to the **`PlayerBullet`** class, as shown in Code Listing 34.

Code Listing 34: Accuracy Leaderboard Event

```
    public delegate void AccuracyEvent(BulletEventType type);
    public event AccuracyEvent Accuracy;

    private float _aliveTime;

//Add this as the last line in the inner If statement in the OnTrigger2D
method.
Accuracy(BulletEventType.Hit);

//Add this as the after the Destroy(gameObject) line in the Update method.
Accuracy(BulletEventType.Miss);
```

Code Listing 35 shows the **`BulletEventType`** enum being added to the Globals file.

Code Listing 35: BulletEventType enum

```
public enum BulletEventType
{
    Hit,
    Miss
}
```

The Ghost Survival Time leaderboard requires an update to the **`Ghost`** class.

Code Listing 36: Ghost Survival Time Leaderboard Event

```
    public delegate void GhostSurvivalTimeEvent(float time);
    public event GhostSurvivalTimeEvent GhostSurvival;

    private float _aliveTime;

void Start ()
```

```
{
    _aliveTime = 0.0f;
}

// Add this as the first lines in the If statement in the Update method.
float step = Time.deltaTime;
_aliveTime += step;

// Add this as the last line in the else If statement in the Update method.
GhostSurvival(_aliveTime);
```

The **LeaderboardsManager** class will then hook into these events and handle them when they're raised.

Code Listing 37: LeaderboardsManager Class

```
public class LeaderboardsManager : MonoBehaviour
{
    public static LeaderboardsManager Instance;

    void Awake()
    {
        Instance = new LeaderboardsManager();
        PlayerBullet.GhostDestroyed += GhostDestroyedHandler;

    }

    private void GhostDestroyedHandler()
    {

    }

    private void PlayerSurvivalHandler()
    {

    }

    private void AccuracyHandler()
    {

    }

    private void GhostSurvivalHandler()
    {

    }
}
```

Inside each event handler, you would add the code to track and/or display the necessary data.

Multiplayer

On every platform, with the possible exception of mobile, most single-player games will only do so well without a multiplayer feature. For almost every type of game, having the ability to play with or against (or, even better, with *and* against) other gamers will make the game more attractive. In most cases, it's worth the effort to implement multiplayer functionality, especially if you plan on selling your game and trying to make a living from game development.

There are several solutions for implementing multiplayer in your game:

- Photon
- Lidgren
- Unity Networking

All three solutions have some level of free functionality. Photon and Unity Networking also offer paid levels with increasing numbers of features, such as more concurrent players.

Photon is a multiplatform network library created by Exit Games. It has functionality for text, voice chat, and setting up a server, along with a matchmaking API. More than 100K developers use it. One of its main draws is that servers are handled by Unity Cloud functionality, which eliminates the need for client servers.

Lidgren is an open-source networking library. It's not as full-featured as the other two options. It's a message-based system which, while allowing a bit more customization, means getting it to work specifically with your game will require some effort. You'll need to take a few steps in order to use it with Unity. Those steps are documented at https://github.com/lidgren/lidgren-network-gen3/wiki/Unity3D.

Unity has its own built-in networking functionality. Although not as old as the other two, it's still robust enough for use with most indie-level games. The main attraction is the ease of integration with a Unity project, and that functionality, such as matchmaking, is handled by a Unity service, which you can read about at https://unity3d.com/services/multiplayer. Much of the code is also available through a Bitbucket project at https://bitbucket.org/Unity-Technologies/networking, which means you can customize the functionality to fit your unique needs.

Adding multiplayer functionality after you've completed development on the single-player version is usually very difficult unless your code is extremely well organized. If you think you might want to have multiplayer functionality, plan for it from the start, even if you don't implement it immediately.

Other gameplay features

While the game we have can be challenging on hard difficulty, it can be a little boring on easy. Implementing other gameplay features could make it much more interesting. We'll talk about a couple here, but feel free to go even further and make your game uniquely your own.

Gradually increasing difficulty

There may be times when your game will be too easy for some players on a particular difficulty level, but too hard on the next. If this happens with a lot of players, you might consider implementing a gradual increase in the difficulty at the easier level. This will make the game more attractive to players who might otherwise not buy/play it.

For our game, increasing the speed and frequency of the spawning of the ghosts makes sense. We can set this up at intervals of destruction of the ghosts by the player—for example, every 25, 50, etc. You'll have to playtest this with players of varying ability levels in order to find the sweet spot that makes the game challenging without being so difficult that players won't want to play it.

Power-ups

What was the last game you played that didn't have some kind of power-up feature? Why didn't it have that feature? Would that have made it a better game? While power-ups aren't appropriate for some types of games, shooters are a prime candidate for power-ups.

Some power-ups are typical for our type of game:

- Health
- Other weapons
- Alternate fire modes
- Shields, invulnerability, or other damage reduction
- Speed increase

Most power-ups will have some properties that are the same no matter their type—delay between spawning, an amount or time the power-up is effective, an amount the power-up increases the character or weapon it's used for. It makes sense, then, to have a base **`PowerUp`** class to hold these properties and either inherit from this class or use some other technique to implement all the different power-ups you want in your game.

Code Listing 38: PowerUp Class

```
public enum PowerUpType
{
    Health,
    Shield,
    Speed
}


public class PowerUp : MonoBehaviour
{
    public PowerUpType Type;
    public int Amount;
    public float EffectiveTime;
    public bool Active;
    public Sprite Icon;
    public bool Rechargeable;
    public float RechargeTime;
    public bool DestoryAfterUse;

    private float _activeTime;
    private float _elapsedRechargeTime;

    public bool CanActivate;

    public PowerUp(PowerUpType type, int amount = 0, float effectiveTime = 
0.0f, bool destroyAfterUse, bool rechargeable = false, bool active = false)
    {
        Type = type;
        Amount = amount;
        EffectiveTime = effectiveTime;
        DestroyAfterUse = destroyAfterUse;
        Rechargeable = rechargeable;
        Active = active;
        CanActivate = !active;
    }

    public void Activate()
    {
        Active = true;
        CanActivate = false;
    }

    void Update()
    {
        if (Active)
        {
            _activeTime += Time.deltaTime;

            if (_activeTime >= EffectiveTime)
            {
                Active = false;

                if (DestoryAfterUse)
                    Destroy(this);
            }
        }
        else if (Rechargeable && !CanActivate)
        {
            _elapsedRechargeTime += Time.deltaTime;

            if (_elapsedRechargeTime >= RechargeTime)
                CanActivate = true;
        }
    }
}
```

The members are public so that the class can be used in the editor and the members set. The constructor takes values for the members as parameters for flexibility.

Because the class inherits from **`MonoBehaviour`**, the **`Update`** method can be used to handle updating the **`_activeTime`** member and disabling the power-up if the **`EffectiveTime`** has passed.

Some power-ups might stay around once they are picked up and used multiple times—possibly with a recharge time between them. Players of MMOs are used to this with abilities that have a cooldown period between uses. A couple of members can give power-ups that same functionality—the **`Rechargeable`**, **`RechargeTime`**, and **`_elapsedRechargeTime`**. The first two are public, which means they can be set in the editor if you want to put power-ups in the level. The last is only used internally in the class, so it's private. If you want to give some kind of indicator to the player of when the power-up will be available again, you can add a public method to get the value.

Achievements

If you want to get your game on a console, you will probably need to think about achievements that the game will award a player for certain accomplishments or progress made in the game. For our game, there's not any real progression, other than surviving. In this case, it would make more sense to have the achievements be for accomplishing things such as surviving for more than a certain time or for killing a certain number of ghosts. If you add other gameplay features or implement other types of functionality we've addressed, the kinds of achievements you are able to award can be more varied.

Achievements for accomplishments such as killing a number of ghosts can be awarded for increments of kills—say 100, 200, etc. Achievements for surviving can be marked in a similar way—5 minutes, 10 minutes, and so on.

Each console has its own API for implementing and awarding achievements/trophies/whatever. In order to officially use their systems, you'll have to go through the developer application process for that console. Appendix A includes URLs for the Xbox One and Playstation programs, as well as Valve's Steamworks.

If you implement your own achievement system, you'll want to store the information as to whether or not an achievement has been unlocked either on the player's machine or on a server somewhere. The former will be easier and will let you avoid communicating with a server that you may or may not be responsible for maintaining and paying for, but it lacks security. Anything stored on a player's machine will be easily editable by the player. Storing on a server will be harder to implement, but the data will be more secure, as the player only has as much access to the data as you give him.

Representing an achievement in code takes only a few pieces of data—the name and description and a date the achievement was unlocked. If the date variable is null, the achievement is still locked. Optionally, you can have a sprite represent the achievement graphically.

Code Listing 39*: Achievement Class*

```
public class Achievement
{
    public string Name;
    public string Description;
    public DateTime? DateUnlocked;
    public Sprite Icon;
}
```

As with leaderboards, ideally you would store the achievement information for players on a server that you maintain or have a level of admin access to, so that you can maintain the information easily. Optionally, storing the information in the registry (in Windows or a system file or folder location, depending on the platform) using the **`PlayerPrefs`** class in the Unity API on the player's machine is an easy-to-implement system. See the documentation on the class at https://docs.unity3d.com/ScriptReference/PlayerPrefs.html.

An **`AchievementManager`** class to track when a player unlocks an achievement would work almost exactly like the **`LeaderboardsManager`** class.

Wrapping up

You should now have a complete game. What do you do with it, however? Ideally, you would compile it for every platform you can and put it in the digital distribution service (Apple AppStore, Google Play Store, etc.) for that platform, if it exists. Some platforms will take a bit of work to prep your game for—touch for mobile, for example. For a small game like this, some platforms might not be ideal. Console players will probably want a few more features than we've implemented. Multiplayer, in this case, would be a good feature to add. There are some things you'll need to consider before putting the game onto a store, however:

- Cost—what should you charge for the game, if you charge anything? Free-to-play is the norm for most mobile platforms, usually with in-app purchases added to help make money. Implementing those will require a good bit of work.

- Platform-specific features—consoles have specific requirements before being allowed in a store. You'll need to implement the requirements for each console, but before doing that you'll typically need to establish a relationship with the console's company.

- Polishing—even in its current state, you'll probably encounter some bugs that will need to be fixed before your game is truly ready to be published. Adding features for specific platforms will require much more testing, both by you and playtesters. Getting at least a handful of people to play your game before putting it out for the world is a must. Make sure the testers are people who want to help you and provide constructive feedback, both good and bad, and who have experience with testing games, if possible. Simply playing the game as a typical player might play isn't sufficient. You'll want people who will do things that are unexpected and try their best to break your game.

- Feedback—you will want to give players a way to contact you, to report bugs, and provide feedback on your game. Giving them a way to provide their contact information so that you can give them news about this game and other games you may create is important if you plan to try making a living in game development. You should also set up a website and an email subscription list.

You've taken your first step into game development. Hopefully, it's been fun and made you want to learn more, so that you can create new, more complex games. Game development is something that requires almost constant learning in order to keep up with the industry. It's a lot of work, but it's very satisfying, especially when you see players having fun with your game. Making a living doing game development is icing on the cake.

If you really want to make game development your job, make sure you take time to play games as well as create them. It's probably safe to say that virtually every game developer is a game player. Playing games is one of the best ways to get your creative juices going and inspire you to create awesome games. I truly wish you the best of luck on your game development journey and hope to one day play some great new games that you created.

Appendix—Resources

ID@Xbox Developer Website: http://www.xbox.com/en-US/developers/id

Playstation Developer Program: https://www.playstation.com/en-us/develop/

Steamworks: https://partner.steamgames.com/

Photon Networking Library: https://www.photonengine.com/en-US/Photon

Lidgren Networking Library: https://github.com/lidgren/lidgren-network-gen3