Unity 5.x Game Development Blueprints

This book will help you to create exciting and interactive games from scratch with the Unity game development platform. We will build 6 action-packed games of different difficulty levels, and we’ll show you how to leverage the intuitive workflow tools and state of the art Unity rendering engine to build and deploy games for multiple platforms.

Through this book, you’ll develop a complete skillset with the Unity toolset. Using the powerful C# language, we’ll create game-specific characters and game environments. Each project will focus on key Unity features as well as game strategy development. This book is the ideal guide to help your transition from an application developer to a fully fledged Unity game developer.

Who this book is written for
This book is best suited for C# developers who have some basic knowledge of the Unity game development platform. If you are looking to create exciting and interactive games with Unity and get a practical understanding of how to leverage key Unity features, then this book is your one-stop solution.

What you will learn from this book
- Use Unity’s native 2D tools to create a twin-stick shooter, an endless mobile game, and a shooting range from scratch
- Explore both of the available GUI systems built into Unity building a clicker game, a reusable menu system, and an in-game level editor
- Create environments that are as realistic as possible, while maintaining peak performance and removing repetitiveness
- Get tips and tricks on how to create environments using both Terrains and a modular workflow
- Modify already existing code to fulfill your own game’s unique mechanics
- Craft AI that will move around the world and players can interact with
- Package your game for release, create an installer, and get your game out into the world
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 '2D Twin-stick Shooter'
- A synopsis of the book’s content
- More information on Unity 5.x Game Development Blueprints
John P. Doran is a technical game designer who has been creating games for over 10 years. He has worked on an assortment of games in teams consisting of just himself to over 70 people in student, mod, and professional projects in different roles, from game designer to lead UI programmer. He previously worked at LucasArts on Star Wars: 1313 as a game designer. He later graduated from DigiPen Institute of Technology in Redmond, WA, with a Bachelor of Science in game design.

John is currently a part of DigiPen's Research and Development branch in Singapore in addition to DigiPen Game Studios. He is also the lead instructor of the DigiPen-Ubisoft Campus Game Programming Program, instructing graduate-level students in an intensive, advanced-level game programming curriculum. In addition to that, he also tutors and assists students in various subjects and gives lectures on C#, C++, Unreal, Unity, game design, and more.

He is the author of Unreal Engine Game Development Cookbook, Building an FPS Game in Unity, Unity Game Development Blueprints, Getting Started with UDK, UDK Game Development, Mastering UDK Game Development, and he cowrote UDK iOS Game Development Beginner's Guide, all available from Packt Publishing. More information about him can be found at http://johnpdoran.com.
Preface

Unity, available in free and pro versions, is one of the most popular third-party game engines available. It is a cross-platform game engine, making it easy to write your game once and then port it to PC, consoles, and even the Web, which makes it a great choice for both indie and AAA developers.

Unity 5.x Game Development Blueprints will take you on an exploration of using Unity to the fullest extent, working on 3D and 2D titles, exploring how to create GUIs, and publishing the game for the world to see. Getting the opportunity to make a second edition of this book, I’ve updated each of the projects from the previous book to the latest version of Unity, making use of the latest features while also adding in three entirely new game projects for this book.

Using this book, you will be able to create a 2D twin-stick shooter, a clicker title, an endless runner, a shooting gallery, a side-scrolling platformer with an in-game level editor, a first-person survival horror shooter game, and a GUI menu system to use in all your future titles. In addition, you will learn how to publish your game with an installer to make your title look really polished and stand out from the crowd.

Each chapter either pushes your skills in Unity into new areas or pushes them to the very limits of what they can be used for.

What this book covers

Chapter 1, 2D Twin-stick Shooter, shows you how to create a 2D multidirectional shooter game. In this game, the player controls a ship that can move around the screen using the keyboard and shoot projectiles in the direction the mouse is pointing at. Enemies and obstacles will spawn towards the player, and the player will avoid/shoot them. This chapter also serves as a refresher for a lot of the concepts of working in Unity and gives an overview of Unity's native 2D tools.
Preface

Chapter 2, Creating GUIs, expands on our twin-stick shooter game, adding additional UI elements, including a main menu as well as a pause menu and options menu, and it gives us the ability to restart our project.

Chapter 3, GUIs Part 2 – Clicker Game, expands on our GUI knowledge with the implementation of a clicker game. Over the course of the chapter, you learn how to create and animate buttons, how to work with accessors (get/set functions), and how to build a simple shop for your projects dealing with UI assets such as Scrollviews and Masks.

Chapter 4, Mobile Endless Game – Procedural Content, has us building a game similar in gameplay to the popular mobile title Flappy Bird while learning how to create endless levels by creating content at runtime. In addition, we go over topics such as creating a repeating background, using sorting layers.

Chapter 5, Shooting Gallery – Working with Animations and Tweens, has you build a simple shooting gallery game in which players can shoot at targets before the time is up. In the meantime, you learn the various ways of animating things such as using Unity's built in animations or a tweening library such as iTween. Finally, we will use PlayerPrefs to set a new high score.

Chapter 6, Side-scrolling Platformer, shows you how to create a side-scrolling platformer. You learn the similarities between working in 2D and 3D and the differences, in particular, when it comes to physics.

Chapter 7, First Person Shooter Part 1 – Creating Exterior Environments, discusses the role of an environment artist who has been tasked to create an outdoor environment while learning about mesh placement. In addition, we will also learn some beginner-level design.

Chapter 8, First Person Shooter Part 2 – Creating Interior Environments, discusses the role of a level designer who has been tasked to create an interior environment using assets already provided to them by the environment artist.

Chapter 9, First Person Shooter Part 3 – Implementing Gameplay and AI, shows how we are going to be adding interactivity in the form of adding in enemies, shooting behaviors, and the gameplay to make our game truly shine. In addition, we'll also learn how to use an Xbox 360 controller to send input to our game.
Chapter 10, Building an In-Game Level Editor, talks about how we can add in functionality to our previously created side-scrolling platformer game in the form of an in-game level editor, which can be used for future projects. In addition, we'll also look at the Immediate Mode GUI System (IMGUI) to see how we can create GUI elements directly through code.

Chapter 11, Finishing Touches, talks about exporting our game from Unity and then creating an installer so that we can give it to all of our friends, family, and prospective customers!
2D Twin-stick Shooter

The shoot 'em up genre of games is one of the earliest kinds of game. In a shoot 'em up, the player character is a single entity fighting a large number of enemies. They are typically played using a top-down or side-scrolling perspective, which is perfect for 2D games. Shoot 'em up games also exist within many other categories, based upon their design elements.

Elements of a shoot 'em up were first seen in the 1961 Spacewar! game. However, the concept wasn't popularized until 1978 with Space Invaders. The genre was quite popular throughout the 1980s and 1990s and went in many different directions, including bullet hell games, such as the titles of the Touhou Project. The genre has gone through a resurgence in recent years with games such as Bizarre Creations' Geometry Wars: Retro Evolved, which is more popularly known as a twin-stick shooter.

Project overview

Over the course of this chapter, we will be creating a 2D multidirectional shooter game similar to Geometry Wars.

In this game, the player controls a ship. This ship can move around the screen using the keyboard and shoot projectiles in the direction that the mouse is pointing. Enemies and obstacles will spawn toward the player, and the player will avoid/shoot them. This chapter will also serve as a refresher for a lot of the concepts of working in Unity and give an overview of the native 2D tools in Unity.

Your objectives

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

- Setting up the project
2D Twin-stick Shooter

- Creating our scene
- Adding player movement
- Adding shooting functionality
- Creating enemies
- Adding GameController to spawn enemy waves
- Particle systems
- Adding audio
- Adding points, score, and wave numbers
- Publishing the game

Prerequisites

Before we start, we will need to get the latest Unity version, which you can always get by going to http://unity3d.com/unity/download/ and downloading it there:

From here, you’ll want to select the Choose your Unity + Download button and from the following menu select the edition of Unity you’d like to use and then download the Launcher.

Everything that we will be covering will work in the Personal Edition, unless I mention specifically otherwise. At the time of writing, the latest version is 5.3.4, but the project should work with minimal changes in future versions.
We will also need some graphical assets for use in our project. These can be downloaded from the example code provided for this book on Packt Publishing's website (http://www.PacktPub.com).

Navigate to the preceding URL, download the Chapter1.zip package, and unzip it. Inside the Chapter1 folder, there are a number of things, including an Art Assets folder, which will have the art, sound, and font files you'll need for the project as well as the Twinstick Shooter folder which contains the finished project for you to take a look at.

**Setting up the project**

At this point, I assume that you have Unity freshly installed and have started it up.

1. With Unity started and the launcher, select **New**. Select a **Name** and a **Project Location** of your choice somewhere on your hard drive, and ensure that you have **2D** set. Once completed, select **Create project**. At this point, we will not need to import any packages, as we'll be making everything from scratch. Have a look at the following screenshot:
2. From there, if you see the Welcome to Unity popup, feel free to close it out as we won't be using it. At this point, you will be brought to the general Unity layout, as follows:

Again, I'm assuming you have some familiarity with Unity before reading this book; if you would like more information on the interface, please visit http://docs.unity3d.com/Documentation/Manual/LearningtheInterface.html.

Keeping your Unity project organized is incredibly important. As your project moves from a small prototype to a full game, more and more files will be introduced to your project. If you don't start organizing from the beginning, you'll keep planning to tidy it up later on, but as deadlines keep coming, things may get quite out of hand.
This organization becomes even more vital when you're working as part of a team, especially if your team is telecommuting. Differing project structures across different coders/artists/designers is an awful mess to find yourself in.

Setting up a project structure at the start and sticking to it will save you countless minutes in the long run and only takes a few seconds, which we'll be doing now. Perform the following steps:

1. Click on the **Create** drop-down menu below the **Project** tab in the bottom-left corner of the screen.

2. From there, click on **Folder**, and you'll notice that a new folder has been created inside your **Assets** folder.

3. After the folder is created, you can type in the name for your folder. Once done, press **Enter** for the folder to be created. We need to create folders for the following directories:
   - Prefabs
   - Scenes
   - Scripts
   - Sprites

   If you happen to create a folder inside another folder, you can simply drag and drop it from the left-hand side toolbar. If you need to rename a folder, simply click on it once and wait, and you'll be able to edit it again.

   You can also use **Ctrl + D** to duplicate a folder if it is selected.
4. Once you’re done with the aforementioned steps, your project should look something like this:

Creating our scene
Now that we have our project set up, let's get started with creating our player:

1. From the Project tab, double-click on the Sprites folder. Once inside, right-click within the Sprites folder and select Import New Asset to select the playerShip.png file from the Chapter 1/Art Assets folder from our example code to bring it in. Once added, confirm that the image's texture type is Sprite by clicking on it and from the Inspector tab, confirm that the Texture Type property is Sprite (2D and UI). If it isn't, simply change it to that, and then click on the Apply button. Have a look at the following screenshot:
If you do not want to drag and drop the files, you can also right-click within the Sprites folder in Project Browser (bottom-left corner) and select Import New Asset to select a file from the Chapter 1/Art Assets folder to bring it in.

The art assets used for this tutorial were provided by Kenney. To see more of their work, please check out www.kenney.nl.
2. Next, drag and drop the ship into the scene (the center part that's currently dark gray). Once completed, set the position of the sprite to the center of the screen (0, 0) by right-clicking on the Transform component and then selecting Reset Position. Have a look at the following screenshot:

![Screenshot showing the Transform component and its options]

Now, with the player in the world, let's add in a background. However, instead of creating a huge image or copying and pasting a similar image over and over, we will learn how we can use a material with a repeating texture.

3. From the Project tab, bring the background sprite into the project and then select it and change the Texture Type in the Inspector tab to Texture, and click on Apply.

4. Now, let's create a 3D cube by selecting Game Object | Create Other | Cube from the top toolbar. Change the object's name from Cube to Background. In the Transform component, change Position to (0, 0, 1) and the scale to (100, 100, 1).

Since our camera is at 0, 0, -10 and the player is at 0, 0, 0, putting the object at position 0, 0, 1 will put it behind all of our sprites. By creating a 3D object and scaling it, we are making it really large, much larger than the player's monitor. If we scaled a sprite, it would be one really large image with pixilation, which would look really bad. By using a 3D object, the texture that is applied to the faces of the 3D object is repeated, and since the image is tileable, it looks like one big continuous image.
5. As we won’t be using it (and we’re a 2D game), remove the BoxCollider component by right clicking on it from the Inspector tab and then selecting Remove Component.

6. Next, we will need to create a material for our background to use. To do so, under the Project tab, select Create | Material, and name the material BackgroundMaterial. We can create a new folder called Materials to store this, but since this project will only use one it is OK to stay in Sprites. Under the Shader property, click on the drop-down menu and select Unlit | Texture. Click on the Texture box on the right-hand side and select the background texture. Once completed, set the Tiling property’s x and y to 25. Have a look at the following screenshot:

   ![Texture Screenshot]

   In addition to just selecting from the menu, you can also drag and drop the background texture directly onto the Texture box and it will set the property.

   Tiling tells Unity how many times the image should repeat in the x and y positions, respectively.

7. Finally, go back to the Background object in Hierarchy. Under the Mesh Renderer component, open up Materials by left-clicking on the arrow, and change Element 0 to our BackgroundMaterial material. Consider the following screenshot:
You can also add a material to an object by dragging and dropping the material onto the object in the Scene tab.

Well, at this point we have a color, but we don’t have the stars from our image repeating. This is due to how the Wrap Mode property is set. In Unity 5.2, the default mode is Clamp, which means that the edges of the image will be extended out rather than repeat. Let’s fix that now.

8. From the Project tab, select the background texture object. Once selected, go to the Inspector tab, change the Wrap Mode to Repeat, and then click on Apply. Finally, select the Cube object from Hierarchy and then in the Inspector tab at the top, rename the object to Background.

Now, when we play the game, you’ll see that we now have a complete background that tiles properly.

**Scripting 101**

In Unity, the behavior of game objects is controlled by the different components that are attached to them in a form of association called composition. These components are things that we can add and remove at any time to create much more complex objects. If you want to do anything that isn’t already provided by Unity, you’ll have to write it on your own through a process we call scripting. Scripting is an essential element in all but the simplest of video games.
Unity allows you to code in either C#, Boo, or UnityScript, a language designed specifically for use with Unity and modeled after JavaScript. For this book, we will use C#.

C# is an object-oriented programming language – an industry-standard language similar to Java or C++. The majority of plugins from Asset Store are written in C#, and code written in C# can port to other platforms, such as mobile, with very minimal code changes. C# is also a strongly typed language, which means that if there is any issue with the code, it will be identified within Unity and will stop you from running the game until it's fixed. This may seem like a hindrance, but when working with code, I very much prefer to write correct code and solve problems before they escalate to something much worse.

### Implementing player movement

Now, at this point, we have a great-looking game, but nothing at all happens. Let's change that now using our player. Perform the following steps:

1. Right-click on the **Scripts** folder you created earlier, click on **Create**, and select the **C# Script** label. Once you click on it, a script will appear in the **Scripts** folder, and it should already have focus and be asking you to type a name for the script – call it **PlayerBehaviour**.

   ![Note]
   
   Note that I'm using Behaviour instead of Behavior. Either is fine to use as long as you're consistent, but since Unity's base script is called MonoBehaviour, I use the same spelling to avoid confusion.

2. Double-click on the script in Unity, and it will open **MonoDevelop** or **Visual Studio Community 2015** depending on how you installed Unity and what OS your computer is running.

   ![Note]
   
   While code can be written in any program that can support text (such as Notepad), both of these pieces of software are integrated development environments (IDEs) that are created to make software development easier and more convenient to work with giving users access to useful features. MonoDevelop is open source, multi-platform, and is included with your Unity installation, but Visual Studio is a very popular tool with programmers and has some nice features that are exclusive to it. However, it can only be used for free by individual developers or for education and open source projects. For this book, we will not be doing anything specific to either IDE, so it's up to you which you'd like to use.
If you'd like to change which piece of software to use to open script files in Unity, you can go to the Edit | Preferences menu and then go to External Tools and under External Script Editor, click on the drop-down menu and select whichever you'd like to use.

After your IDE has loaded, you will be presented with the C# stub code that was created automatically for you by Unity when you created the C# script.

Let's break down what's currently there before we replace some of it with new code. At the top, you will see two lines:

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

The engine knows that if we refer to a class that isn't located inside this file, then it has to reference the contents within these namespaces for the referenced class before giving an error. We are currently using two namespaces.

The UnityEngine namespace contains interfaces and class definitions that let MonoDevelop know about all the addressable objects inside Unity.

The System.Collections namespace contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hash tables, and dictionaries. But we will not be using it in this file so you can remove that line.

The next line you'll see is:

```csharp
public class PlayerBehaviour : MonoBehaviour {
```
You can think of a class as a kind of blueprint for creating a new component type that can be attached to GameObjects, the objects inside our scenes that start out with just Transform and then have components added to them. When Unity created our C# stub code, it took care of that; we can see the result, as our file is called PlayerBehaviour and the class is also called PlayerBehaviour. Make sure that your .cs file and the name of the class match, as they must be the same to enable the script component to be attached to a game object. Next up is the : MonoBehaviour section of the code. The : symbol signifies that we inherit from a particular class; in this case, it is set to MonoBehaviour. All behavior scripts must inherit from MonoBehaviour directly or indirectly by being derived from it. This idea of having classes inherit behavior is known as inheritance.

Inheritance is the idea of having an object to be based on another object or class using the same implementation. With this in mind, all of the functions and variables that exist inside the MonoBehaviour class will also exist in the PlayerBehaviour class, because PlayerBehaviour is a MonoBehaviour with the ability to add more to it.

Checking Unity’s Script Reference can be quite useful when determining what has already been done for you in Unity. For more information on the MonoBehaviour class and all of the functions and properties it has, check out http://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

Directly after this line, we will want to add some variables to help us with the project. Variables are pieces of data that we wish to hold on to for one reason or another, typically because they will change over the course of a program, and we will do different things based on their values.

Add the following code under the class definition:

```csharp
// Movement modifier applied to directional movement.
public float playerSpeed = 4.0f;

// What the current speed of our player is
private float currentSpeed = 0.0f;

// The last movement that we've made
private Vector3 lastMovement = new Vector3();
```
Between the variable definitions, you will notice comments to explain what each variable is and how we'll use it. To write a comment, you can simply add a // to the beginning of a line and everything after that is commented upon so that the compiler/interpreter won't see it. If you want to write something that is longer than one line, you can use /* to start a comment, and everything inside will be commented until you write */ to close it. It's always a good idea to do this in your own coding endeavors for anything that doesn't make sense at first glance.

For those of you working on your own projects in teams, there is an additional form of commenting that Unity supports, which may make your life much easier: XML comments. They take up more space than the comments we are using, but also document your code for you. For a nice tutorial about that, check out [http://bit.ly/xmlComments](http://bit.ly/xmlComments).

One of the things you'll notice is the public and private keywords before the variable type. These are access modifiers that dictate who can and cannot use these variables. The public keyword means that any other class can access that property, whereas private means that only this class will be able to access this variable. Here, currentSpeed is private because we want our current speed not to be modified or set anywhere else. But, you'll notice something interesting with the public variable that we've created. Go back into the Unity project and drag and drop the PlayerBehaviour script onto the playerShip object. Before going back to the Unity project though, make sure that you save your PlayerBehaviour script. Not saving is a very common mistake made by people when starting to code. Have a look at the following screenshot:
You'll notice now that the public variable that we created are located inside Inspector for the component. This means that we can actually set those variables inside Inspector without having to modify the code, allowing us to tweak values in our code very easily, which is a godsend for many game designers. You may also notice that the name for Player Speed has been changed to be more readable. This is because of the naming convention that we are using where each word starts with a capital letter. This convention is called CamelCase (more specifically headlessCamelCase).

Now that we have our variables set, go back to MonoDevelop for us to work on the script some more.

The line after that is a function definition for a method called Start; it isn't a user method but one that belongs to MonoBehaviour. Where variables are data, functions are the things that modify and/or use that data. Functions are self-contained modules of code (enclosed within braces { and }) that accomplish a certain task. The nice thing about using a function is that once a function is written, it can be used over and over again. Functions can be called from inside of other functions:

```csharp
void Start () {
}
```

Start is only called once in the lifetime of the behavior when the game starts and is typically used to initialize data.

If you're used to other programming languages, you may be surprised that initialization of an object is not done using a constructor function. This is because the construction of objects is handled by the editor and does not take place at the start of gameplay as you might expect. If you attempt to define a constructor for a script component, it will interfere with the normal operation of Unity and can cause major problems with the project.

However, for this behavior, we will not need to use the Start function. Perform the following steps:

1. Delete the Start function and its contents.

The next function that we see included is the Update function. Also inherited from MonoBehaviour, this function is called for every frame that the component exists in and for each object that it's attached to. We want to update our player ship's rotation and movement every turn.
2D Twin-stick Shooter

2. Inside the `Update` function (between `{` and `}`), put the following lines of code:
   ```csharp
   // Rotate player to face mouse
   Rotation();
   // Move the player's body
   Movement();
   
   Here, I called two functions, but these functions do not exist because we haven't created them yet (and will possibly show up as red in your IDE because of that). Let's do that now!

3. Below the `Update` function and before the `}` that closes the class, put the following function to close the class:
   ```csharp
   // Will rotate the ship to face the mouse.
   void Rotation()
   {
      // We need to tell where the mouse is relative to the player
      Vector3 worldPos = Input.mousePosition;
      worldPos = Camera.main.ScreenToWorldPoint(worldPos);

      /*
      * Get the differences from each axis (stands for deltaX and deltaY)
      */
      float dx = this.transform.position.x - worldPos.x;
      float dy = this.transform.position.y - worldPos.y;

      // Get the angle between the two objects
      float angle = Mathf.Atan2(dy, dx) * Mathf.Rad2Deg;

      /*
      * The transform's rotation property uses a Quaternion,
      * so we need to convert the angle in a Vector
      * (The Z axis is for rotation for 2D).
      */
      Quaternion rot = Quaternion.Euler(new Vector3(0, 0, angle + 90));

      // Assign the ship's rotation
      this.transform.rotation = rot;
   }
   ```
In this code, we are changing the rotation of the object we attached this script to (our player) based on where the player has put the mouse. Now, if you comment out the `Movement` line (by putting `//` in front of it) and run the game, you'll notice that the ship will rotate in the direction in which the mouse is. Have a look at the following screenshot:

If you did comment out the line, be sure to remove the `//` before continuing.

4. Below the `Rotation` function, we now need to add the following code in our `Movement` function:

```csharp
// Will move the player based off of keys pressed
void Movement()
{
    // The movement that needs to occur this frame
    Vector3 movement = new Vector3();

    // Check for input
    movement.x += Input.GetAxis("Horizontal");
    movement.y += Input.GetAxis("Vertical");

    /*
     * If we pressed multiple buttons, make sure we're only
     * moving the same length.
     */
    movement.Normalize();

    // Check if we pressed anything
    if(movement.magnitude > 0)
    {
        // If we did, move in that direction
        currentSpeed = playerSpeed;
        this.transform.Translate(movement * Time.deltaTime * 
        playerSpeed, Space.World);
        lastMovement = movement;
    }
    else
    {
```
2D Twin-stick Shooter

```csharp
// Otherwise, move in the direction we were going
this.transform.Translate(lastMovement * Time.deltaTime * currentSpeed, Space.World);
// Slow down over time
currentSpeed *= .9f;
}
```

Now, inside this function, we use Unity's built in Input properties in order to set the x and y aspect of our movement vector and then move (or Translate) the ship in that direction. Edit | Project Settings | Input will open up Unity's Input Manager, which will have a list of different Axes that can be used for input.

As you can see here, left and a are used in the Negative Button and Alt Negative Button properties respectively, so if we press either of those two it will give you a number between -1 and 1, with positive giving you a value between 0 and 1.
For more information on the Input Manager, check out: http://docs.unity3d.com/Manual/class-InputManager.html.

5. Now, save your file and move back into Unity. Save your current scene as Chapter_1.unity by going to File | Save Scene. Make sure to save the scene to our Scenes folder we created earlier.

6. Run the game by pressing the play button on the top toolbar. And you should see something like the following screenshot:

![Game screenshot](image)

Now, you'll see that we can move using the arrow keys or the WASD keys, and our ship will rotate to face the mouse. Great!

If you'd like to make your ship move faster and/or slower, increase or decrease the PlayerBehavior component's Player Speed variable inside the Inspector tab.
Shooting behavior

The next thing we will do is give our player the ability to shoot:

1. Open up the PlayerBehaviour script. In the top section where the other variables are present, we need to add some additional ones that we'll use:

   // The laser we will be shooting
   public Transform laser;

   // How far from the center of the ship should the laser be
   public float laserDistance = .2f;

   // How much time (in seconds) we should wait before
   // we can fire again
   public float timeBetweenFires = .3f;

   // If value is less than or equal 0, we can fire
   private float timeTilNextFire = 0.0f;

   One thing you may have noticed is that we have a laser variable that is of the type Transform. This is the laser we'll fire, which we will create shortly.

   In our game, the player may want to shoot using either my clicking or the Space key. You may even want to use something else. Rather than restricting the player to just having one button or using an Axis like we did previously, we will store all of the possible ways to shoot in their own container. To do this, we are going to use a list, which is a holder for multiple objects that we can add or remove while the game is being played. However, to use a list, we will need to add the following to the top of our code:

   using System.Collections.Generic; // List


   Afterwards, add the following variable below the others:

   // The buttons that we can use to shoot lasers
   public List<KeyCode> shootButton;
2. Inside our `Update` function, we will need to add some additional code, which is as follows:

```csharp
// a foreach loop will go through each item inside of
// shootButton and do whatever we placed in {}s using the
// element variable to hold the item
foreach (KeyCode element in shootButton)
{
    if(Input.GetKey(element) && timeTilNextFire < 0)
    {
        timeTilNextFire = timeBetweenFires;
        ShootLaser();
        break;
    }
}

timeTilNextFire -= Time.deltaTime;
```

In this code we check each of the keys we allow the player to shoot with (such as the spacebar and `Enter` keys). If they press any of these keys and can fire again, then we will reset our timer and shoot a laser. However, we haven't made the `ShootLaser` function. Let's do that now.

3. Underneath the functions, add the following function:

```csharp
// Creates a laser and gives it an initial position in
// front of the ship.
void ShootLaser()
{
    // We want to position the laser in relation to
    // our player's location
    Vector3 laserPos = this.transform.position;
    // The angle the laser will move away from the center
    float rotationAngle = transform.localEulerAngles.z - 90;
    // Calculate the position right in front of the ship's
    // position laserDistance units away
    laserPos.x += (Mathf.Cos((rotationAngle) *
        Mathf.Deg2Rad) * -laserDistance);
    laserPos.y += (Mathf.Sin((rotationAngle) *
        Mathf.Deg2Rad) * -laserDistance);

    Instantiate(laser, laserPos, this.transform.rotation);
}
```
4. Save your file, and go back into Unity. You'll now see a number of additional variables that we can now set.

If, for some reason, your Inspector window doesn't update, save your project and restart Unity. Upon reset, it should be updated.

5. Now, change the Size of each of the Shoot Button variables to 2, and fill in the Element 0 value with Mouse0 and Element 1 with Space. When you've finished, it should look something like the following screenshot:

![Inspector window](image)

6. Next, we will need to create our laser to fill in the Laser variable. Go back into our Assets folder from the example code, and move the laser.png file into our Project tab's Sprites folder.
7. Following that, drag and drop it into your scene from the **Scene** tab to place it in the level.

8. Right-click the **Scripts** folder you created earlier, click on **Create**, and select the **C# Script** label. Call this new script **LaserBehaviour**. Go into your IDE, and use the following code:

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

public class LaserBehaviour : MonoBehaviour
{
    // How long the laser will live
    public float lifetime = 2.0f;

    // How fast will the laser move
    public float speed = 5.0f;

    // How much damage will this laser do if we hit an enemy
    public int damage = 1;

    // Use this for initialization
    void Start ()
    {
        // The game object that contains this component will be
        // destroyed after lifetime seconds have passed
        Destroy(gameObject, lifetime);
    }

    // Update is called once per frame
    void Update ()
    {
        transform.Translate(Vector3.up * Time.deltaTime * speed);
    }
}
```
9. Attach `LaserBehaviour` to the laser object. Finally, add a `Box Collider 2D` component by first selecting the laser object and then going to `Component` | `Physics 2D` | `Box Collider 2D`. The collision box, by default, will be the size of the image, but I want to shrink it to fit what is inside of it. To do that, we will change the `Size` attribute's `X` property to `.09` and `Y` to `.5`. To see exactly what we're modifying, you can zoom in the camera by using the mouse wheel in the `Scene` tab:

![Collider Set up](image)

Now, the laser will move in the direction that it's facing and die after a period of two seconds! Next, let's make it so that the player can shoot them.

10. To differentiate this object from the `laser` sprite that we used to create it, select your laser object in the `Hierarchy` tab and then under the `Inspector` change its name to `Laser`.

11. In the `Project` tab, go to the `Assets` | `Prefabs` folder, and drag and drop the Laser object from our `Hierarchy` tab into it. You'll notice that the object `Hierarchy` will turn blue to show that it is a prefab.
Prefabs or prefabricated objects are the objects that we set aside to make copies during runtime, such as our bullets and eventually enemies that we'll spawn into the world, and we can create as many as we want. When you add a prefab to a scene, you create an instance of it. All of these instances are clones of the object located in our Assets. Whenever you change something in the prefab located in our Prefab folder, those changes are applied to all of the objects that are already inside of your scene. For example, if you add a new component to Prefab, all of the other objects we have in the scene will instantly contain the component as well. We can also apply any of the ones in our scene to be the blueprint for the others as well, which we will do later on. However, it is also possible to change the properties of a single instance while keeping the link intact. Simply change any property of a prefab instance inside your scene, and that particular value will become bolded to show that the value is overridden, and they will not be affected by changes in the source prefab. This allows you to modify prefab instances to make them different (unique) from their source prefabs without breaking the prefab link.

Have a look at the following screenshot:
12. Now, because we have the prefab created, we don't need the original in the scene because we'll be spawning them instead. With that in mind, delete the Laser object from our scene, and then go to the playerShip object. For clarity, rename it to Player Ship and then drag and drop the Laser prefab from the Project tab into the Laser property of the PlayerBehavior component.

13. Finally, we want our player to have collision as well, so let's add a circle collider to our ship by going to Component | Physics 2D | Circle Collider 2D. Change the Radius property to 0.3.

Generally, in games, we want to be as efficient as possible toward calculations. Polygon collision is the most accurate collision, but it is much slower than using a box or a circle. In this case, I wanted to use a circle, because not only is it more efficient but also allows the player some leeway in how close they can get to enemies without being hurt. Players will always think it's their skill if they get away, but if the collider is too big they will think the game is broken, which we want to avoid.

Have a look at the following screenshot:

Now, our ship can shoot in the direction that the mouse is currently facing.
Creating enemies

Now, it's really cool that we have a player, but it'll get really boring if all we can do is move around and shoot some lasers in the dark. Next, we'll introduce some simple enemies that will move toward the player that we'll be able to shoot later. Perform the following steps:

1. Leave the game by pressing the Play button again and then access our example code's Assets folder; move the enemy.png file into our Sprites folder.
2. Following that, drag and drop it into your scene from the Scene tab to place it in the level.
3. Right-click on the Scripts folder you created earlier, click on Create, and select the C# Script label. Call this new script MoveTowardsPlayer. Go to MonoDevelop and use the following code:

```csharp
using UnityEngine;
using System.Collections;
public class MoveTowardsPlayer : MonoBehaviour
{
    private Transform player;
    public float speed = 2.0f;

    // Use this for initialization
    void Start ()
    {
        player = GameObject.Find("playerShip").transform;
    }

    // Update is called once per frame
    void Update ()
    {
        Vector3 delta = player.position - transform.position;
        delta.Normalize();
        float moveSpeed = speed * Time.deltaTime;
        transform.position = transform.position + (delta * moveSpeed);
    }
}
```

In the beginning of the game, I find the player ship and get his Transform component. Then, in every frame of the project, we move the enemy from where it currently is to the direction where the player is at.
The `GameObject.Find` function is very useful as it allows us to access objects at any time, but is computationally expensive, so don’t put it in a function that gets called often (such as `Update`) when you can just save a reference to it. It’s also important to note that the spelling of the parameter has to be exactly the same as the object’s name in the hierarchy; so be sure to double check when you use it.

![Warning](https://i.imgur.com/3Q5Z5.png)

If you ever want to have objects run away from the player, use a negative speed.

4. Drag and drop this newly added behavior onto our enemy object and rename it to `Enemy`.

5. Next, add a circle collider to our enemy by going to `Component | Physics 2D | Circle Collider 2D`. Change the `Radius` property to 0.455, and run the game. Have a look at the following screenshot:
Now, you'll see that the enemy will always move toward you! But if we shoot it, nothing happens. Let's fix that as follows:

1. Right-click on the **Scripts** folder you created earlier, click on **Create**, and select the **C# Script** label. Call this new script **EnemyBehaviour**. Go to MonoDevelop (your IDE of choice), and use the following code:

   ```csharp
   using UnityEngine; // MonoBehaviour
   public class EnemyBehaviour : MonoBehaviour
   {
     // How many times should I be hit before I die
     public int health = 2;

     void OnCollisionEnter2D(Collision2D theCollision)
     {
       // Uncomment this line to check for collision
       // Debug.Log("Hit" + theCollision.gameObject.name);

       // this line looks for "laser" in the names of
       // anything collided.
       if (theCollision.gameObject.name.Contains("laser"))
       {
         LaserBehaviour laser =
         theCollision.gameObject.GetComponent("LaserBehaviour") as LaserBehaviour;
         health -= laser.damage;
         Destroy (theCollision.gameObject);
       }

       if (health <= 0)
       {
         Destroy (this.gameObject);
       }
     }
   }
   ```

Now, you will notice that we have commented a line of code calling the function **Debug.Log**. This function when called will print something onto your console (in this case, name the object that we collided with), which may help you when debugging your own code in the future.

It's also important to note the reason we ask if the name contains Laser instead of is equal to is because when you create an object using Instantiate, by default it will add "(Clone)" to the end of the name of the object.
2. We also want to tell the objects to actually react to collision events so we will need to add **Rigidbody 2D** physics to the object, which is required one of the two objects involved in a collision in order for **OnCollisionEnter2D** to be called.

3. Select the **Enemy** object and then select **Component | 2D Physics | Rigidbody 2D**. From there, change the **GravityScale** value to 0 so that the object will not fall down by default due to gravity.

4. Save your script, and then go back into Unity. Attach the **EnemyBehaviour** behavior to your enemy object, save your project, and run the game. If all went well, we should have something like the following screenshot:

![Screenshot of Unity game](image)

Now, whenever we hit the enemy with our bullets twice, it will die. Nice!
Adding GameController to spawn enemy waves

We have all of the mechanics of our game completed at this point. Now, we need to actually create the game or manage what happens in the game. This game controller would be required to run our game, keep track of and display the game's score, and finally end the game whenever the player dies. Later on, we'll discuss a game state manager, which we can use for larger projects with multiple states, but for the sake of this simple project, we will create a simple game controller, and that's what we'll do now:

1. First, create an empty game object by going to Game Object | Create Empty. From there, with the object selected, go to Inspector and set its name to Game Controller, and optionally, for neatness sake, set its Position to (0, 0, 0). As this is our main game object, I'm also going to drag it up on the Hierarchy tab so that it is the top object on the list.

2. Underneath the name, you'll see the Tag property. Change it from Untagged to GameController.

A Tag is a way to link to one or more game objects in a collected group. For instance, you might use Player and Enemy tags for players and enemies respectively; a Collectable tag could be defined for power-ups or coins that are placed in the scene, and so on. This could also have been used with EnemyBehaviour to check whether something was a bullet or not. One thing to note is the fact that GameObject can only have one tag assigned to it. Tags do nothing to the scene but are a way to identify game objects for scripting purposes.

3. Next, let's see another way that we can create scripts quickly. From the Inspector tab, select Add Component | New Script, and once you are brought to the next menu, change the language to C#, and set the name of the script to GameController.
4. Select the newly created script, and move it to the *Assets\Scripts* folder. Go to your IDE of choice by double-clicking on the script file.

While our game does many things, the most important thing is the spawning of enemies, which is what we'll be adding in first. Let's create a variable to store our enemy.

5. Inside of the class definition, add the following variable:

```csharp
// Our enemy to spawn
public Transform enemy;
```

6. Now, we can set the enemy that we currently have in the scene, but we should instead make the enemy a prefab and use it. To do so, drag the enemy from *Hierarchy* into your *Assets\Prefabs* folder. Once we've created the prefab, we can remove the enemy object from our scene by deleting it.

7. Next, drag and drop the enemy prefab into the *Enemy* variable in the *GameController* component.
8. Next, go back into our GameController script by double-clicking it to go into MonoDevelop. Add the following additional variables to the component:

```
[Header("Wave Properties")]
// We want to delay our code at certain times
public float timeBeforeSpawning = 1.5f;
public float timeBetweenEnemies = .25f;
public float timeBeforeWaves = 2.0f;

public int enemiesPerWave = 10;
private int currentNumberOfEnemies = 0;
```

Now, if we save the script and go back into the editor, you'll notice a couple of interesting things. First of all, you'll see that due to the Header that we added to our script, the script has been separated into sections to make it easier to compartmentalize our code. In addition, we also see a warning on the bottom-left of the screen saying that we haven't used our variables yet. You can click on it to open up the Console window to look at it, but with that in mind, let's make it so that we are actually using the variables.
We now need a function to spawn enemies; let's call it `SpawnEnemies`. We don't want to spawn all of the enemies at once. What we want is a steady stream of enemies to come to the player over the course of the game. However, in C#, to have a function pause the gameplay without having to stop the entire game, we need to use a coroutine that looks different from all of the code that we've used so far.

9. Inside the `Start` method, add the following line:
   ```csharp
   StartCoroutine(SpawnEnemies());
   ```

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

10. Now that we are already using the function, let's add in the `SpawnEnemies` function as follows:
    ```csharp
    // Coroutine used to spawn enemies
    IEnumerator SpawnEnemies()
    {
        // Give the player time before we start the game
        yield return new WaitForSeconds(timeBeforeSpawning);
        // After timeBeforeSpawning has elapsed, we will enter
        // this loop
        while(true)
        {
            // Don't spawn anything new until all of the previous
            // wave's enemies are dead
            if(currentNumberOfEnemies <= 0)
            {
                // Spawn 10 enemies in a random position
                for (int i = 0; i < enemiesPerWave; i++)
                {
                    // We want the enemies to be off screen
                    // (Random.Range gives us a number between the
                    // first and second parameter)
                    float randDistance = Random.Range(10, 25);
                    // Enemies can come from any direction
```
Vector2 randDirection =
    Random.insideUnitCircle;
Vector3 enemyPos =
    this.transform.position;

    // Using the distance and direction we set the
    // position
    enemyPos.x += randDirection.x * randDistance;
    enemyPos.y += randDirection.y * randDistance;

    // Spawn the enemy and increment the number of
    // enemies spawned
    // (Instantiate Makes a clone of the first
    // parameter
    // and places it at the second with a rotation of
    // the third.)
    Instantiate(enemy, enemyPos, this.transform.rotation);
    currentNumberOfEnemies++;
    yield return new
    WaitForSeconds(timeBetweenEnemies);
}

    // How much time to wait before checking if we need to
    // spawn another wave
    yield return new WaitForSeconds(timeBeforeWaves);
}

The ++ operator will take the current value of a number and increment it by 1.

11. Now, when we destroy an enemy, we want to decrement the number of
    currentNumberOfEnemies, but it's a private variable, which means that it
    can only be changed inside the GameController class or one of the methods
    inside of the class. Simple enough? Now, let's add a new function in our
    GameController class:
    // Allows classes outside of GameController to say when we
    // killed an enemy.
    public void KilledEnemy()
    {
        currentNumberOfEnemies--;
    }
2D Twin-stick Shooter

12. Finally, let’s go back into our EnemyBehaviour class. Inside the OnCollisionEnter2D function under the Destroy function call, add the following lines:

```
GameController controller = GameObject.FindGameObjectWithTag("Game Controller").GetComponent<GameController>();
controller.KilledEnemy();
```

The preceding line gets the script GameController from the game object that has the GameController tag.

This will call the KilledEnemy function from GameController, onto which we set the GameController tag in step 2.

13. With all those changes, save both script files and run the game! Have a look at the following screenshot:

![Screenshot of the game with waves of enemies moving towards the player](image)

We now have waves of enemies that will now move toward the player! When we kill all of the enemies inside a wave, we will spawn the next wave. In such a short period of time, we already have so much going on!
Particle systems for enemy explosion

Now that we have the basis for our game, let's spend some time to make the project look nicer. Particle systems are one of my go-to things to add juiciness to a game project and helps to set your project apart from others. Particle systems are composed of two separate parts: a particle and the thing that emits it (called an emitter). A particle is a small object that stores properties; generally, we try to make these objects as simple as possible, as we want to spawn a large number of them. The emitter's job is to spawn a number of these and initialize their properties. Thankfully, Unity has a fully featured particle editor that's included with the engine, and we're going to use it in this section. Perform the following steps:

1. Create a new particle system by going to **GameObject | Particle System**. Have a look at the following screenshot:

![Particle System](image)

Once you do this, you should see a default particle system show up. Do note that the system will only animate if it is the object selected and Unity is the active window.

2. Change the object's name to **Explosion**. First, under the **Particle System** tab, change **Duration** to **1.00**.
3. Click on the arrow on the right-hand side of **Start Lifetime**, and change the values to be **Random Between Two Constants**. Change those values to 0 and 1. Do the same with **Start Speed**. Make **Start Size** use random values between 0 and .5.

4. Next, we will set the object's **Start Color** value to the same color as the UFO ship (you can use the eyedropper tool or set it to 210, 224, 230) and an **Alpha** value of 125. Have a look at the following screenshot:

![Color settings](image)

5. Open the **Emission** tab, and change **Rate** to 200. This is how many particles are spawned at a time.

6. Open the **Shape** tab, change the **Shape** property to **Sphere**, and then set **Radius** to .35 to fit the rim of the ship. Enable the **Random Direction** option.

7. Go back to the top **Explosion** section and then change **Simulation Space** to **World**; that way, if this object moves, the already-spawned particles will not move. After this, still in the **Explosion** section, uncheck **Looping**.

8. At this point, you'll need to click on the **Stop** and **Simulate** buttons to see how the properties change while we make adjustments.

9. Now, make this object a prefab by dragging and dropping it into the **Prefabs** folder. After that, delete the **Explosion** object in your **Hierarchy** object.
10. Go back to your EnemyBehaviour script file. We will first want to add in a new variable for us to use to spawn this explosion when it dies:

```csharp
// When the enemy dies, we play an explosion
public Transform explosion;
```

11. Back in the Project tab, drag and drop your new Explosion prefab to fill in the explosion variable slot in our Enemy prefab.

12. Coming back to the EnemyBehaviour script, let's spawn an explosion whenever we die. Inside your `if(health <= 0)` section of `OnCollisionEnter2D`, add in the following lines:

```csharp
// Check if explosion was set
if(explosion)
{
    GameObject exploder = ((Transform)Instantiate(explosion, this.transform.position, this.transform.rotation)).gameObject;
    Destroy(exploder, 2.0f);
}
```

As you can see here, we use the Instantiate function in order to create an explosion and we call the Destroy function to have it automatically delete itself (the first parameter) after 2 seconds as passed (the second parameter).

13. Save your script and scene files, and run your project! Have a look at the following screenshot:

And now, whenever an enemy dies, it will spawn an explosion for us to see!
Adding in sound effects/music
Another thing that we can do to give the project a little more polish is add in sound effects and background music. Let's do that now. Perform the following steps:

1. Select your **enemy** prefab, and add an **Audio Source** component to it by selecting **Component | Audio | Audio Source**.
2. From there, go into the **Inspector** and uncheck the **Play On Awake** property. Otherwise, if there is something attached to **AudioClip** it will play that sound automatically. An **audio source** lets the audio listener attached to the **Main Camera** object know that this is an object that can play sounds.
3. After this, let's go into our **EnemyBehaviour** script! As usual, we'll need to add in a new variable for us to use to play whenever we're hit:
   ```csharp
   // What sound to play when hit
   public AudioClip hitSound;
   ```
4. Next, go into the **CollisionEnter2D** function. After the `Destroy(theCollision.gameObject)` line, add the following code:
   ```csharp
   // Plays a sound from this object's AudioSource
   GetComponent<AudioSource>().PlayOneShot(hitSound);
   ```

   In this instance, I used the `GetComponent` function to access the **AudioSource** component that is attached to the game object this is attached to. It is important to note that `GetComponent` is a fairly expensive function to call. Since this will only be used once, it's OK to use like this; but if you were to use this more than once, I would suggest creating an **AudioSource** variable and setting it in the **Start** function instead like we will be doing in the next section.

   For more information about the **PlayOneShot** function, check out [http://docs.unity3d.com/ScriptReference/AudioSource.PlayOneShot.html](http://docs.unity3d.com/ScriptReference/AudioSource.PlayOneShot.html).

   For more information on the **Audio Source** component (audio), check out [http://docs.unity3d.com/ScriptReference/AudioSource.html](http://docs.unity3d.com/ScriptReference/AudioSource.html).

5. Now, we need some actual sounds to play. I've set aside a folder of assets for you in the **Example Code** folder, so open your project folder via your **Explorer** and move the folder into your project's **Assets** and then return to Unity.
6. Back in the inspector for our enemy, let’s set the Hit Sound variable in the EnemyBehaviour script to the hit sound that we’ve imported by using drag and drop. Now, if we play the game, when we hit an enemy, the sound will be played! Now, let’s have a sound if we destroy the enemy!

7. Go to the Explosion prefab, and add an AudioSource component in the same way we did in step 1; however, make sure that the Play On Awake property is still checked. After this, set the Audio Clip property in the component to the explode sound.

Now, if you play the game, hitting the object will play one sound, and when the object is destroyed, the explosion will play a sound. Because the sound is in the Audio Clip property, we can just check Play On Awake or call the Play function. However, if you want an object to play multiple sounds, it’s better to have separate AudioClip variables just as we did with EnemyBehaviour.

8. Finally, I want to play a sound whenever we fire a shot. To do that, let’s go to playerShip and add an audio source.

9. Next, go into PlayerBehaviour, and add in two new variables, as follows:

```csharp
// What sound to play when we're shooting
public AudioClip shootSound;

// Reference to our AudioSource component
private AudioSource audioSource;
```

10. Then, we need to initialize the audio property, so add a Start function with the following:

```csharp
void Start()
{
    audioSource = GetComponent<AudioSource>();
}
```

11. After this, whenever we shoot a bullet, let’s play the new sound at the beginning of the ShootLasers function:

```csharp
audioSource.PlayOneShot(shootSound);
```

12. Coming back to Inspector, set the Shoot Sound property in the PlayerBehaviour component to the shoot sound effect.

13. Finally, let’s add in our background music. Go to your Main Camera object in Hierarchy. Add an AudioSource component. Change Audio Clip to bgm, check the Loop option, and set Volume to .25.
14. Save everything, and run the game!

Although we won’t see any changes at this point for those of you actually running the game, you’ll notice quite a change when the game is started. It’s already feeling much more like a game.

**Adding in points, score, and wave numbers**

One of the most important things to do in a game is reward the player and give them a sense of progression. For now, let’s reward the player with a score we will display for them and also let the player know exactly which wave he is on. Perform the following steps:

1. Create a new **Text** object by going to the **Hierarchy** tab and selecting **Create | UI | Text**.

   You’ll notice that three objects get created at this point: **Canvas**, **Text**, and **Event System**. Right now, we only need to worry about **Text**; however, the three objects are all needed in order for Unity’s new UI system to function properly. We will be discussing this in much more detail in a later chapter.

2. Next, select the **Text** object again and notice that the top of it has a **Rect Transform** component instead of a **Transform**. This is a special transform that is used to position UI elements on the screen relative to other objects that are also part of the UI.

The background music is provided for this project by Stratkat (Kyle Smith). If you are interested in more of his work, check out his website at http://daydreamanatomy.bandcamp.com/.
3. In this case, we want the text to be located in the top-left, but currently you may not be able to see the text at all. You can double-click on the object to zoom out and see it more clearly.
As you can see, the text is much larger than the player, and there seems to be a white box surrounding the object. This is a representation of what the UI will look like, with the white box being the screen. Rect Transform is currently set up to represent a position of (0,0) to be the center of the world and currently the text is -21 pixels away from the center in the x axis and -97 in the y. If we were to change it to (0,0), you'd notice that the text would now be centered.

4. Rename the Text object to Score Counter and then change the Anchors Min Max and Pivot properties to (0, 1); then reset Pos X to 10 and Pos Y to -10 to move them slightly away from the edge of the screen. Lastly, change the Color to white to make it easier to see.

5. Now that we have the text set up, let's set the font. Bring the Font folder into your project. Then, set Font to OSP-DIN and Font Size to 25.

The font used in this project was created by the OSP Foundry. For more information about their stuff, check out http://ospublish.constantvzw.org/foundry/.
6. Next, duplicate the **Score Counter** object by right-clicking and selecting **Duplicate**. Set this duplicate's name to **Waves Counter**, and change its text to **Wave: 0**.

7. Set the **Waves Counter** object's **Anchors Min Max** and **Pivot** properties to (1, 1) and then reset **Pos X** to -10 and **Pos Y** to -10 to move them slightly away from the edge of the screen. Afterward, in the text component, change the **Alignment** to be right aligned horizontally.

8. Now that we have our text files created, let's now have them function correctly! First, let's go into the **GameController** class. Inside, we will first need to let the script know to use Unity's new UI system:

```csharp
using UnityEngine.UI;
```

9. Afterward, we need to create some new variables as follows:

```csharp
[Header("User Interface")]
// The values we'll be printing
private int score = 0;
private int waveNumber = 0;

// The actual GUI text objects
public Text scoreText;
public Text waveText;
```
10. Next, we will need to add a function to call whenever our score increases, as follows:

```csharp
public void IncreaseScore(int increase)
{
    score += increase;
    scoreText.text = "Score: " + score;
}
```

The `+=` operator will take the current value of the variable and add something additional to it.

11. Then, we’ll need to call this function inside our `EnemyBehaviour` component. After the `controller.KilledEnemy()` line, add the following line:

```csharp
controller.IncreaseScore(10);
```

12. Finally, whenever we increase the wave number, we need to change its text as well. Back in the `GameController` class after the opening `{` of the `if(currentNumberOfEnemies <= 0)` block, add the following lines:

```csharp
waveNumber++;
waveText.text = "Wave: " + waveNumber;
```

13. Save all of the script files, go back to `Inspector`, and set the `Score Text` and `Wave Text` objects to the proper variables in the `Game Controller` object. After that, save your project and run the game. Have a look at the following screenshot:
And with that, you can see that everything is working together, killing enemies will reward the player points, and killing all of the enemies in a wave triggers the next wave to start!

**Publishing the game**

The final thing that we are going to touch on for the project is actually publishing it:

1. Go to File | Build Settings. From here, you can decide which platforms and/or scenes to include with your project.
2. Click on the Add Current... button to add our current scene to the game, as follows:
3. After that, since we are just publishing to our current platform, confirm that the settings are correct, and click on the Build and Run button.

4. Once you press the button, you'll be brought to a menu to name your application that you are going to save. Give it a name, save it, and wait. If all goes well, you should be brought to a menu allowing you to set some options before the game starts:
5. After that, click on the **Play!** button to see your completed project. Have a look at the following screenshot:

![Screenshot of a game project](image.png)

**Summary**

And there we have it! Within this first chapter, you've already completed an entire game project and learned how to publish it. Moving on, in the next chapter, we will tackle more advanced game types, learn additional things about Unity, and do more to push Unity to do as much as possible.
Challenges
For those of you who want to do more with this project, there are still plenty of things you can do, especially after finishing the rest of this book. Here are some ideas to get your mind thinking:

- Add in feedback whenever the player hits an enemy – perhaps an animation or change the sprite used.
- **Give the player lives**, and each time he is hit by an enemy, have him lose one life. Add Text to display lives as well.

Once you learn how to use Unity's GUI system even more, create a main menu, pause screen, and restart button using UI as well!

- Add in Xbox controller and mobile touch support!
Where to buy this book
You can buy Unity 5.x Game Development Blueprints from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.