Community Experience Distilled

# Learning Unity iOS Game Development

Build exciting games with Unity on iOS and publish them on the App Store

Kyle Langley

[PACKT] PUBLISHING

# Learning Unity iOS Game Development

# Table of Contents


- Learning Unity iOS Game Development
- Credits
- About the Author
- About the Reviewers
- www.PacktPub.com
  - Support files, eBooks, discount offers, and more
    - Why subscribe?
    - Free access for Packt account holders
- Preface
  - What this book covers
  - What you need for this book
  - Who this book is for
  - Conventions
  - Reader feedback
  - Customer support
    - Downloading the example code
    - Downloading the color images of this book
    - Errata
    - Piracy
    - Questions
- 1. Requirements and Preparation Work
  - Requirements
    - Downloading and installing Unity 5
    - Installing the iOS SDK plugin
  - Creating and logging in to the iTunes Connect profile
  - Creating the Bundle ID and Game App
    - Setting up App pricing and territories
    - Creating an in-app purchase
    - Game Center, leaderboards, and achievements

The Main App settings
Enabling iAds
Setting up iOS Native in Unity
Certificates and provisions on Mac
Why iTunes Connect is important
Summary
2. Player Input for Mobile Devices
Creating the C# input class
Including system
What is Touch?
Our own Touch
Global variables
Creating the Input GameObject
Calculating the TouchInput function
The Start function
Starting to work with Update
The in-editor input logic
Testing in editor
Input for the Touch device
Wrapping up
Summary
3. Player Character, Obstacles, and Pickups
What is a character?
What is a pickup?
What is an obstacle?
Creating the character
Creating the character controller
Creating the Character GameObject
Adding Rigidbody 2D
Adding Box Collider 2D
Writing the character code and connecting the PlayerInput class to it

- Character attributes
- PlayerInput
- The KillCharacter function
- Creating the coin pickup
- Creating the obstacle
- Summary

4. Editor Tool, Prefabs, and Game Level
- Making the sprite tile editor tool
  - The SpriteTiler C# class
  - Global variables
  - The MenuItem creation
  - The OnGUI function
    - The GUILayout and OnGUI setup
  - The OnGUI create tiled button
    - The CreateSpriteTiledGameObject function
    - Testing the SpriteTiler file
    - Sprite Tiler wrap-up
- The prefab creation
  - Coin prefabs
  - Level piece prefabs
  - Further setup of the level pieces
  - The level piece code
  - The level piece manager code
  - Setting up the LevelPieceManager GameObject
  - Adding coins and obstacles to LevelPieces
  - Managing coins and obstacles with the LevelPiece code
    - Resetting coins
    - Resetting the coin in LevelPiece
- Summary

5. Scene Background, the Game User Interface, and the Save Profile
- Creating the scene background

- Adding movement to SceneBackgroundElement
- Creating the game user interface
  - The GameInfo class
  - Importing UI images
    - Creating the UI Canvas
      - The coin background
      - The distance background
      - The pause button
    - Updating code to show coins and distance
      - Updating the coin class
      - The restart button
    - The fade object
      - The restart functionality
        - The Awake function
      - The Update function
      - The RestartGame function
      - The RestartLevelAndCharacter function
      - FadeToBlack and FadeToNormal functions
      - The HideRestartButton function
    - The ResetLevelPieces function in LevelPieceManager
      - Updating Character To Reset
      - Connecting RestartGame to the RestartButton click
  - PlayerPrefs game save object
- Summary

6. Main Menu, iAds, Leaderboards, Store Purchases, and Achievements
- Building the main menu UI
  - Adding button images
    - PlayButton
    - LeaderboardButton
    - AchievementButton
    - RemoveAdsButton

- RestorePurchaseButton
- The Purchase Remove iAds panel
  - RemoveAdsBackgroundScreen
  - RemoveAdsScreen
  - RemoveAdsTextOption
  - ForCoin
  - ForCoinText
  - ForCash
  - ForCoinText
  - CloseRemoveAds
- The Purchase Succeeded panel
  - PurchaseSucceededBackgroundScreen
  - PurchaseSucceededScreen
  - PurchaseSucceededText
  - PurchaseSucceededAccept
  - PurchaseSucceededAcceptText
  - PurchaseSucceededClose
- The Purchase Failed panel
- Connecting the MainMenuUI reference to GameInfo

The menu code
- The GameInfo menu code
- The LevelPieceManager menu code
- The character menu code
- Completing changes in Unity

Adding the BackToMenu button in GameUI

Code for leaderboard and achievement buttons

Code for achievements
- Game center events
  - SubmitAchievementProgress
  - SubmitAchievementAsWhole
- SubmitLeaderboardScore

The character achievement code
Displaying iAds
Purchasing remove iAds
New RemoveiAds Purchase for coins
The In App Purchase code
The Purchase RemoveIAds functions
Purchasing buttons
Restoring purchases
Summary
7. Game Builds and Submission to Apple
Setting up Unity for iOS builds
Creating a test build
Making a release build
Submitting the game to Apple
Congratulations!
What next?
Testing
I want to learn more about programming
I want to create a 3D game
3D programs
Creating textures
Furthermore
Using the Internet
Summary
Index

# Learning Unity iOS Game Development

# Learning Unity iOS Game Development

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2015

Production reference: 1230915

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-980-1

www.packtpub.com

# Credits

**Author**

Kyle Langley

**Reviewers**

Chris Avellis

Luís Pedro Fonseca

Fabian Rastorfer

Ranpariya Ankur J. {PAHeartBeat}

**Commissioning Editor**

Ashwin Nair

**Acquisition Editor**

Vinay Argekar

**Content Development Editor**

Sumeet Sawant

**Technical Editor**

Humera Shaikh

**Copy Editor**

Relin Hedly

**Project Coordinator**

Shweta H Birwatkar

**Proofreader**

Safis Editing

**Indexer**

Monica Ajmera Mehta

**Production Coordinator**

Nilesh R. Mohite

**Cover Work**

Nilesh R. Mohite

# About the Author

**Kyle Langley** is a self-taught game designer who has worked on games such as Transformers: Fall of Cybertron released by High Moon Studios and Warm Gun, Carnival of Bullets, and Maximum Overdrive released by Emotional Robots. He is currently developing a multiplayer PC title for Vex Studios called Jeklynn Heights. Kyle has also tutored many game design students and has previously released a beginners' programming book for Unreal Development Kit called *Learn Programming With Unreal Script*. His website is, www.dotvawxgames.com.

# About the Reviewers

**Chris Avellis** has loved games ever since the Atari era. He began with Neopets way back in the early 2000s, making games and applications in Flash before moving on to working for a Facebook game company. Chris then proceeded to web development and eventually came back to games using Unity. He is currently working for codeSpark, Inc. on The Foos, an award-winning game that teaches kids computer science. In his spare time, Chris enjoys studying Japanese, playing board games, doing art and animation, playing on his PlayStation, and working on mobile games with Unity. You can check out his work at http://www.chrisavellis.com/.

**Luís Pedro Fonseca** is an interactive developer from Oporto, Portugal with over a decade of experience in working with a multitude of technologies ranging from Unity to Flash and HTML5. For several years now, he has focused strongly on gaming and mobile. At the same time, Luís worked for high-profile companies such as FOX Broadcasting Company, Microsoft, Riot Games, EMC, and many others, creating a wide range of interactive projects that are experienced by millions of users.

You can find more about Luís' work on his personal blog at www.luispedrofonseca.com.

I would like to thank everyone at Packt for the chance to work on this book. I would also like to thank my family, especially my fiancé, for supporting me through the long nights and weekends of work that go into my many side projects, such as this book.

**Fabian Rastorfer** is the founder and CEO of Fabraz, a company that excels in digital design and development. Fabian graduated in design and technology from Parsons School of Design and was the creator behind the hit iOS game Cannon Crasha which was critically acclaimed and featured by Apple numerous times. His second game, The Wild Wild Pixel, also gained significant appraisal from the press and was given the green light on Steam. His newest title, Planet Diver, will be released in late 2015. Fabian is also the steady hand behind The Tale of Aeria, which is currently gaining over 2,000 views a day. He has done design work for Zaha Hadid, Baz Luhrmann, Conan O'Brian, Wired, ProHelvetia, Photoshop Creative, Samsung, Citibank, Maybelline, Sony Music Entertainment, Google, and more. Fabian is all about entertainment and experimenting in every possible media.

**Ranpariya Ankur J. {PAHeartBeat}** is a programmer by heart and passion and a game programmer by profession. He has a vast experience in working with different programming languages and another 6 years of experience in programming, where he worked with various technologies and languages. Ankur is a techno freak and is always keen on trying his hand at something new. He believes that "we never lose; we either WIN, or we LEARN something". Ankur has also reviewed *Creating E-Learning Games with Unity*, *Packt Publishing*.

I am very thankful to my wife and younger brother for giving me the space and time to review this book, even when it meant that they had to compromise on their own work. Without their support, it would have been very hard to complete the reviewing of all the

chapters.

I am also thankful to the team at Packt Publishing and the author of this book for giving me this opportunity.

**www.PacktPub.com**

# Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

# Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

# Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

# Preface

*Learning Unity iOS Game Development* covers everything that is needed to create and publish an iOS game using Unity 5. The process includes creating an Apple Developer account, creating a new App using iTunes Connect, which includes the Game Center integration of leaderboards and achievements, and creating an in-app purchase and displaying iAds to the user. Furthermore, we will go through many of the introductory steps within Unity 5 to create a fully functional game with the C# programming language. We will also discuss its application in creating a user interface that takes touch input, stores saved data, and has a full game loop so that the player can navigate between the menu and game states. Finally, we will go through the process of creating a test build before finally making a distribution build to submit to Apple for approval.

# What this book covers

Chapter 1, *Requirements and Preparation Work*, teaches you how to create an iTunes Developer account and the app with the settings for leaderboards, achievements, in-app purchases, and iAds. It also teaches you how to download and install Unity 5 in order to create a new 2D game project.

Chapter 2, *Player Input for Mobile Devices*, writes the C# code needed to both test inside of Unity 5 and take user input on the iOS device.

Chapter 3, *Player Character, Obstacles, and Pickups*, uses Unity 5 to create GameObjects that visually show a 2D sprite and animation before writing the C# code so that they are interactive in the scene.

Chapter 4, *Editor Tool, Prefabs, and Game Level*, uses the Unity 5 EditorWindow class and C# to write an editor tool that will help create the GameObjects that make up the game level before discussing how to create and use prefabs to put together the game level.

Chapter 5, *Scene Background, the Game User Interface, and the Save Profile*, writes a C# class that manages background sprite images before discussing the basics of the Unity 5 UI system and how to use the PlayerPrefs class to save game data.

Chapter 6, *Main Menu, iAds, Leaderboards, Store Purchases, and Achievements*, shows further development using the Unity 5 UI system to create the main menu and how to set up the main menu to show leaderboards and achievements, handle in-app purchasing, and display iAds to the device screen.

Chapter 7, *Game Builds and Submission To Apple*, creates a test build to play on your device before going through the process of creating a distribution build that you can then submit to Apple for review.

# What you need for this book

In order to compile and submit an iOS build, you *must* have a Macintosh computer running the most current OS X version and have the most recent version of Xcode installed. You will also need a computer that meets the minimum requirements to run Unity 5, which includes Windows XP SP2+, Windows 7 SP1+, Windows 8+, or Mac OS X 10.8+, and a graphics card that supports DirectX 9.

You will also need a plugin that allows you to access native iOS functions that can handle in-app purchases. In the book, I use IOS Native, which is available through Unity Asset Store.

To only follow the process and not build or submit the game to Apple, you simply need any computer that has the minimum requirements to run Unity 5.

# Who this book is for

This book is for those who are either brand new to game development or those who have made games for other platforms and are looking to explore how to make games for iOS. You need to have basic computer knowledge, including how to install, open, and navigate programs through instruction, as well as the desire to follow along and write computer code.

# Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: “This means that if the game is running at 30 frames-per-second, `Update()` will be called 30 times in that second.”

A block of code is set as follows:

```
using UnityEngine;
using System.Collections;
public class PlayerInput : MonoBehaviour
{
  /* Use this for initialization */
  void Start ()
  {
  }
  /* Update is called once per frame */
  void Update ()
  {
  }
}
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: “When this folder is created, right-click on it. Then, navigate to **Create** and select **C# Class**.”

## Note

Warnings or important notes appear in a box like this.

## Tip

Tips and tricks appear like this.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

# Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

# Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/9801OT_Graphics.pdf.

# Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

# Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

# Questions

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

# Chapter 1. Requirements and Preparation Work

Unity 3D is a game engine that specializes in allowing developers to produce games for almost all gaming platforms. For iOS games, we must first prepare the engine so that it can publish builds that Apple will approve of. We will overcome some technical issues, such as how to get the users touch input to control the game from an Apple device, how to unlock achievements that show up in Apple's Game Center, and how to show advertisements in the game.

In this chapter, we will cover the following topics:

- Downloading and installing Unity 5
- Using the Unity Asset Store to download and install the iOS SDK plugin
- Creating and logging in to iTunes Connect
- Creating the game app in iTunes Connect
- Setting up Game Center achievements and leaderboards
- Enabling iAds for the game

# Requirements

The bare minimum requirements to follow along with this book is a Windows or Macintosh computer that is able to run Unity 5. You also *must* have a Macintosh computer to compile, test, and submit the game to iTunes and have the most recent operating system for the Macintosh computer to run Xcode. It is important to have the most recent Xcode because the App Store now requires 64-bit compiling for an app to be accepted.

If you have plans on publishing a game to the iTunes Store, you will need an iOS developer account and an iOS license, which costs $99.00 USD per year. For the Unity packages, you will be given an option to use either a free or paid version. I will let you know about the package that I will be using. If you opt to not use the package that I am using, there will be some more work on your end to figure out some technical differences between them, although I will do my best to point you in the right direction.

If you do want to publish a game to the iTunes store, you will need a Macintosh computer with the current Xcode version installed. The most recent version is 6.4 and is available to all OS X Yosemite users. This will be used to create the proper credentials to upload the game to iTunes Connect for review.

# Downloading and installing Unity 5

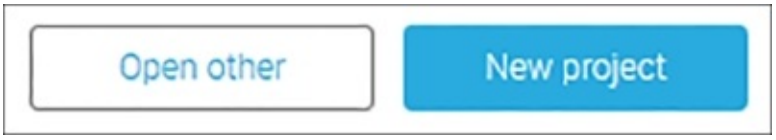
Unity 3D is offered free of charge by Unity Technologies via its website at http://unity3d.com/get-unity.

This page also displays the differences between the professional and free versions.

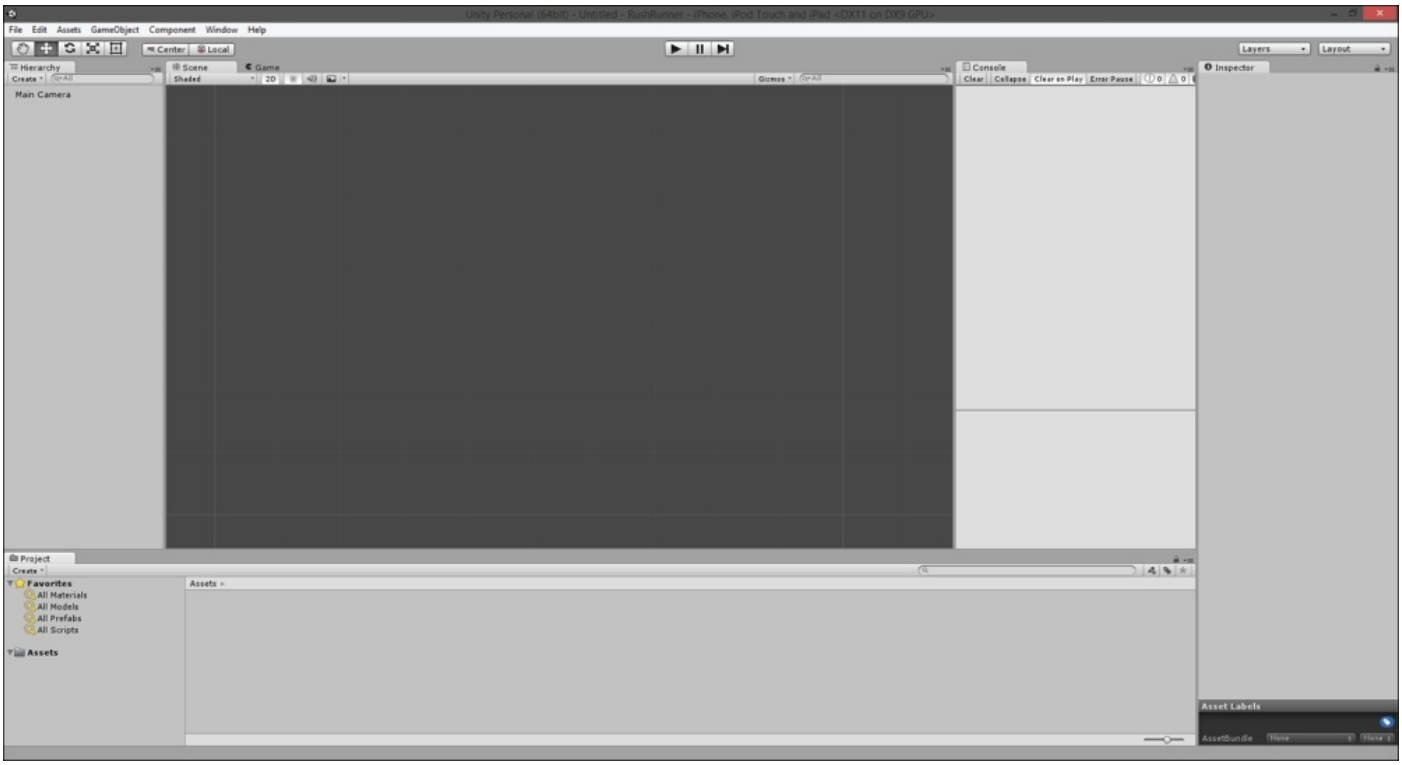
As of today, the most recent version is 5.0.0. Click on the **FREE DOWNLOAD** or **FROM $75/MONTH** button on the web page mentioned before. The current Unity licensing states that if you are a noncommercial entity or make less than $100,000 a year, you are free to use the entire engine without any royalties paid. Due to this reason, I recommend that you use **FREE DOWNLOAD**. When you click on download, you will get a very small installer. Run this and allow it to download what is needed for the engine. The engine requires 4 GB of HDD space and is about 1.5 GB to download.

Once the download is complete, open the installer. The first window, **Download Assistant**, will be called. It will ask you what to install. Make sure that you keep **Unity 5** checked. You can uncheck **Web Player** and **Standard Assets** if you want.

After installing and opening Unity, you will not see any projects. To create one, click on the **New Project** button in the top-right corner of the Unity window. You can call it anything you desire. At the bottom of the window, there is a selection to change from **3D** to **2D**. Switch to **2D** and click on **Create project**.




At this point, the editor will open up and give you a blank screen to work with:


Unity Personal (64bit) - Untitled - RushRunner - iPhone, iPod Touch and iPad <DX11 on DX9 GPU>
File Edit Assets GameObject Component Window Help
Center
Local
Layers
Layout
Hierarchy
Create
Main Camera
Scene
Shaded
2D
Gizmos
Game
Console
Clear
Collapse
Clear on Play
Error Pause
Inspector
Project
Create
Favorites
All Materials
All Models
All Prefabs
All Scripts
Assets
Assets ›
Asset Labels
AssetBundle
None
None

# Installing the iOS SDK plugin

We can use the free or nonfree version of a plugin. I will use the paid version called **IOS Native**.

The reason for me not using the free version called **iOS SDK Essentials** is that its version has not had any care for a while (at the time of writing this book). This means that there could be general update concerns by the time you are reading this.

To get **iOS Native** for $20.00 or **iOS SDK Essentials**, open the Unity Asset Store by clicking on **Window** and then **Unity Asset Store** or press *Ctrl + 9*. With the Asset Store window open, use the top-right search field to find either of these packages. From here, you can either buy and/or download it.

## Tip

The iOS SDK plugin allows you to trigger events for achievements, leaderboards, and in-app purchases. It also allows you to show iAd banners.

# Creating and logging in to the iTunes Connect profile

For Apple to approve our game, we must first create the development account for iTunes Connect. If you have no plans of publishing a game to the iTunes Store, you do *not* have to follow these steps.

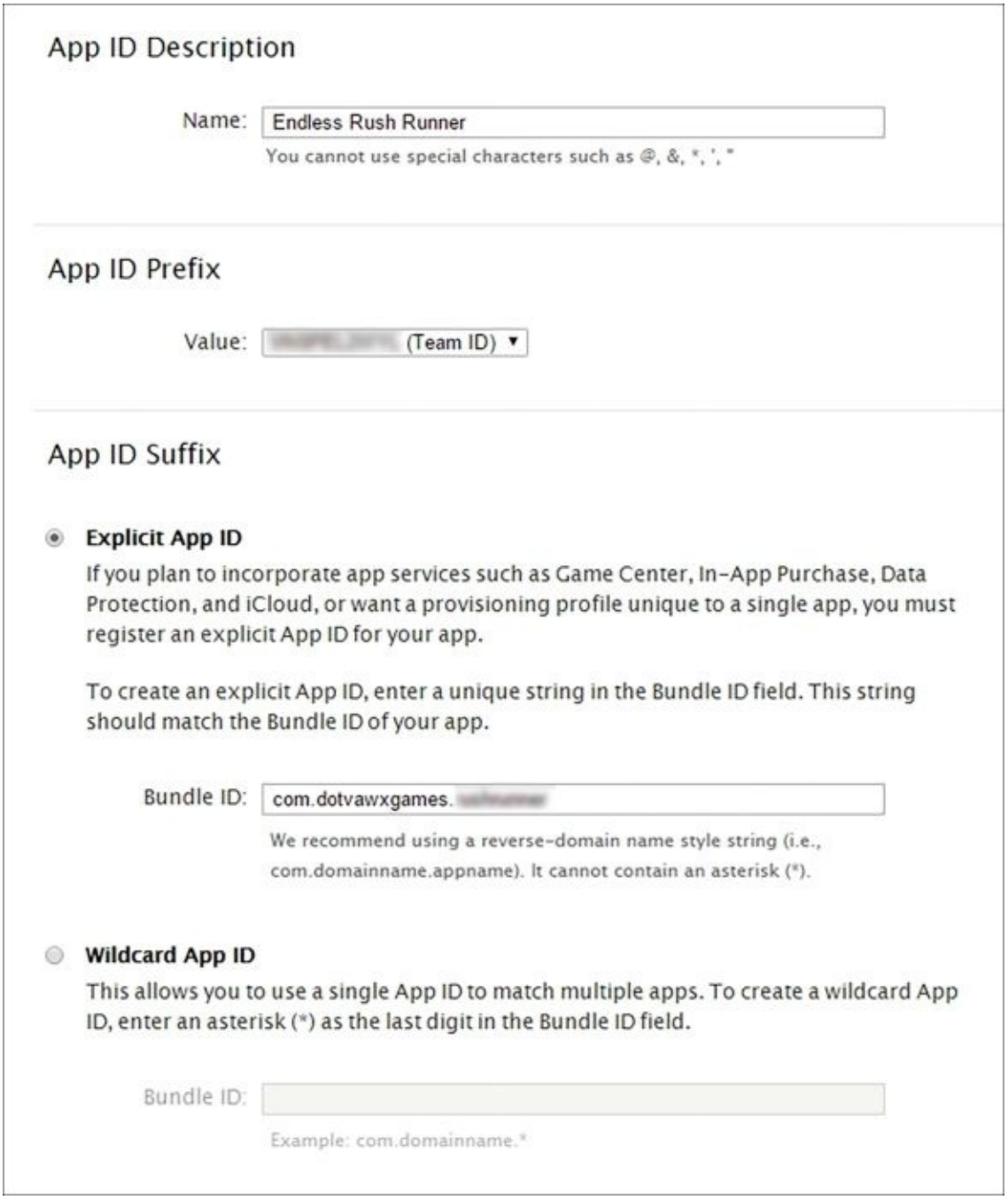
1. Navigate to https://developer.apple.com/.
2. At the bottom of this page, under **Download the latest development tools and SDK**, click on the **iOS Apps** button.
3. The following page will have a **Register for free** hyperlink at the top of the page. Click on it.
4. On the next page, if you have a current Apple ID, click on the **Sign In** button. If you need to create the Apple ID, click on the **Create Apple ID** button and follow the steps to create the Apple ID.
5. When you are logged in to your developer account, navigate to the iOS Dev Center. On the right-hand side, you will find **Join the iOS Developer Program**. Click on the **Learn More** hyperlink.
6. As stated before, you have to pay $99.00 USD per year to join the developer program. Click on the **Enroll now** button and follow the steps, including the payment information and identity requirements. Again, if you do not wish to publish games to the iTunes store, you do not need to pay $99.00 USD per year.
7. If you have completed the payment process, go to https://itunesconnect.apple.com.
8. Log in to iTunes Connect with the account information you created. You will probably see a prompt to enter the required contracts and payment information at the top of this screen. This is required by Apple to sell your game on their store.
9. You will also see many icons. Click on **My Apps**.

# Creating the Bundle ID and Game App

In the top-left corner of the screen, you will see a blue + sign; click on it.

Before you enter any of the information, click on **Developer Portal** at the bottom of the window. This allows you to set up the Bundle ID before creating the application.

Once at **Developer Portal**, the **App ID Description** can be the name of the application. The **App ID Prefix** should be filled out for you. I will use **Team ID**. I used **Explicit App ID** and used the **reverse-domain name** style: `com.mycompanyname.appname`. For **App Services**, I will use **Game Center** and **In-App Purchase**:




Click on **Continue**.

On the next page, **Confirm your App ID** will display the information that you have entered. If correct, click on **Submit**. It should follow with the **Registration Complete** page.

Go back to the **My Apps** page in iTunes Connect. Again, click on the blue **+** sign and then **New iOS App**.

In the popup window that shows, enter the name, version, language, SKU, and bundle ID. If you are curious about what any of these are, you can click on the question mark sign next to the name.

## Note

SKU stands for Stock keeping unit. It is a unique tracking number that is used in the app store, which is specific to your application. You can use anything you like as long as it's unique from any of your other apps. For example, you can use `com.mycompanyname.appname` or your birth date `01012015`.

If the name you chose is not taken, you will be taken to the **My Apps** page for your new game app.

# Setting up App pricing and territories

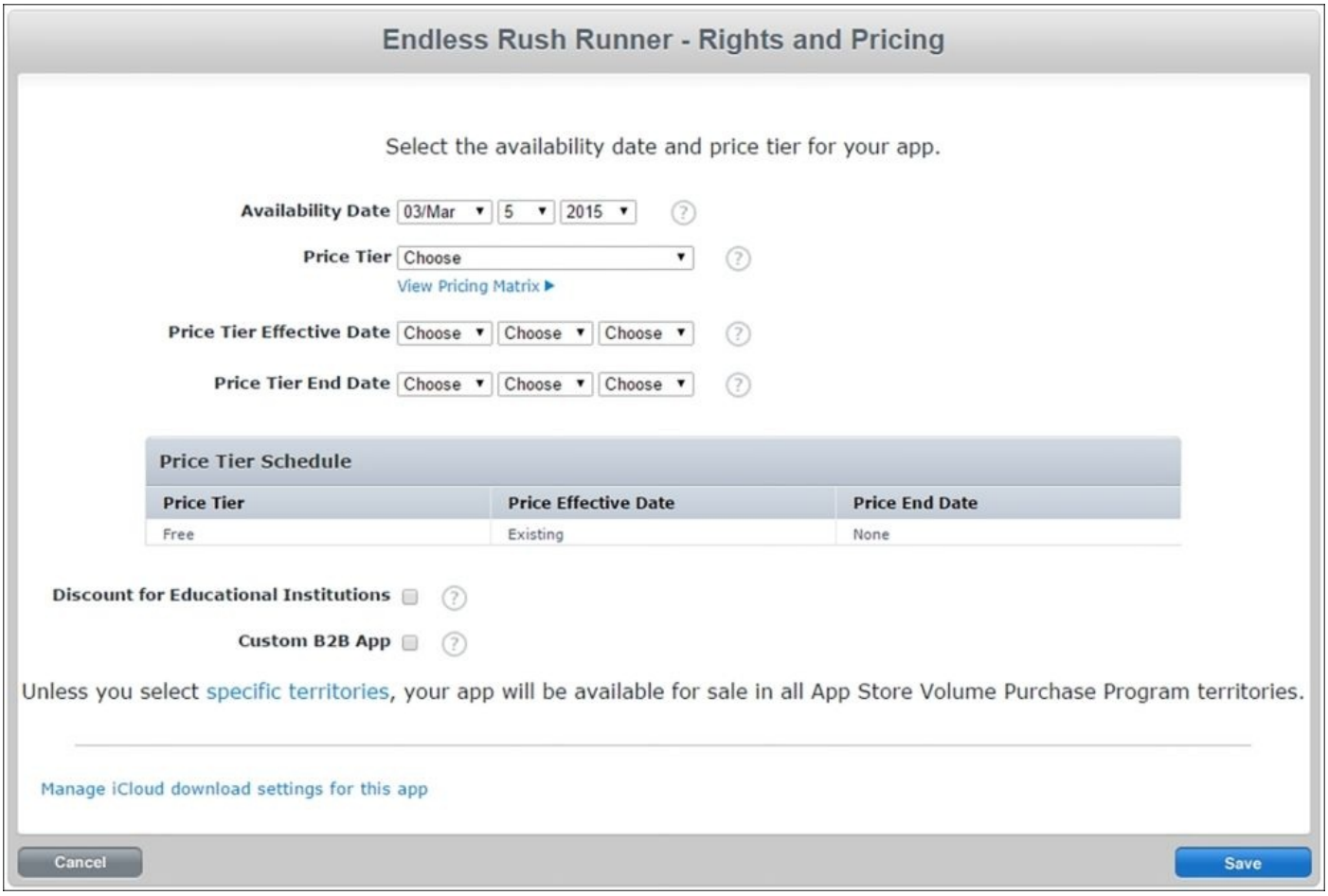
On the App settings page, click on **Pricing**. You can get here by clicking on **My Apps** and then opening your game by clicking on its icon.

**Pricing** is the page where you can choose the initial cost of your application and the territories that will be available.

**Availability Date** should be the date, in which you will create this application. You can leave it as default for now. In the future, if you want to change the price for something, such as a sale, **Availability Date** will dictate how long this pricing tier can last.

For my game, it will be free. To set this or any other price, click on the **Price Tier** drop-down list and select the price.

To use **Discount for Educational Institutions**, use your own discretion:




The last thing you need to do on this page is click on **specific territories** at the bottom of the settings page. Everything is defaulted to be selected. If there are any territories you don't wish to have your game available on, click on the checkbox next to it. When done, click on **Save** at the bottom of the page.

The page should load back to **Rights and Pricing** and have the **Completed** box at the top of the page.

# Creating an in-app purchase

First, click on the **My Apps** drop-down list at the top of the page and select **My Apps**. Then, click on your game to go back to the settings page.

Next, click on **In-App Purchases**.

The page will read **No In-App Purchases have been set up for this app**. To set one up, click on **Create New**.

Then, click on **Non-Consumable**.

There are five purchase types:

- **Consumable**
- **Non-Consumable**
- **Auto-Renewable Subscriptions**
- **Free Subscription**
- **Non-Renewing Subscription**

**Reference Name** is the name that will be displayed in iTunes Connect and in Sales Trend reports. It is not displayed on the App Store. It's a way for us to have a reference to what it is we will sell when we look at reports.

Name it `Remove_Ads` with an underscore.

The product ID is a unique ID that is used to report the purchase. Typically, I would name this as `gamename.referencename`. Generally, this would be written as `<<gamename>>.<<referencename>>`, which symbolizes that you can put anything between these fields; however, as not all of you are aware that these symbols represent a dynamic field—meaning that it is about the formatting and not the literal text—`gamename.referencename` is the literal view of how it's inputted.

As I don't know the name of your games and we will both have to use this in the future chapters, the naming will have to match.

Name the product ID as `G_RemoveAds` with an underscore.

Under **Pricing and Availability**, select **Yes** for **Cleared for Sale**. For **Price Tier**, I have selected **Tier 1**, which is $0.99USD. If you are curious about the pricing matrix, click on **View Pricing Matrix**.

At the bottom of the page, click on **Add Language**.

A pop-up window will appear. Select **English** as the language if not already selected. **Display Name** is the name of the in-app purchase that will be seen by customers.

Enter `Remove Ads`.

**Description** is the description of the in-app purchase that will be used by **App Review** during the review process. It is used to describe what the purchase will do.

Enter `Stops displaying iAds in the game`.

Then, click on **Save**.

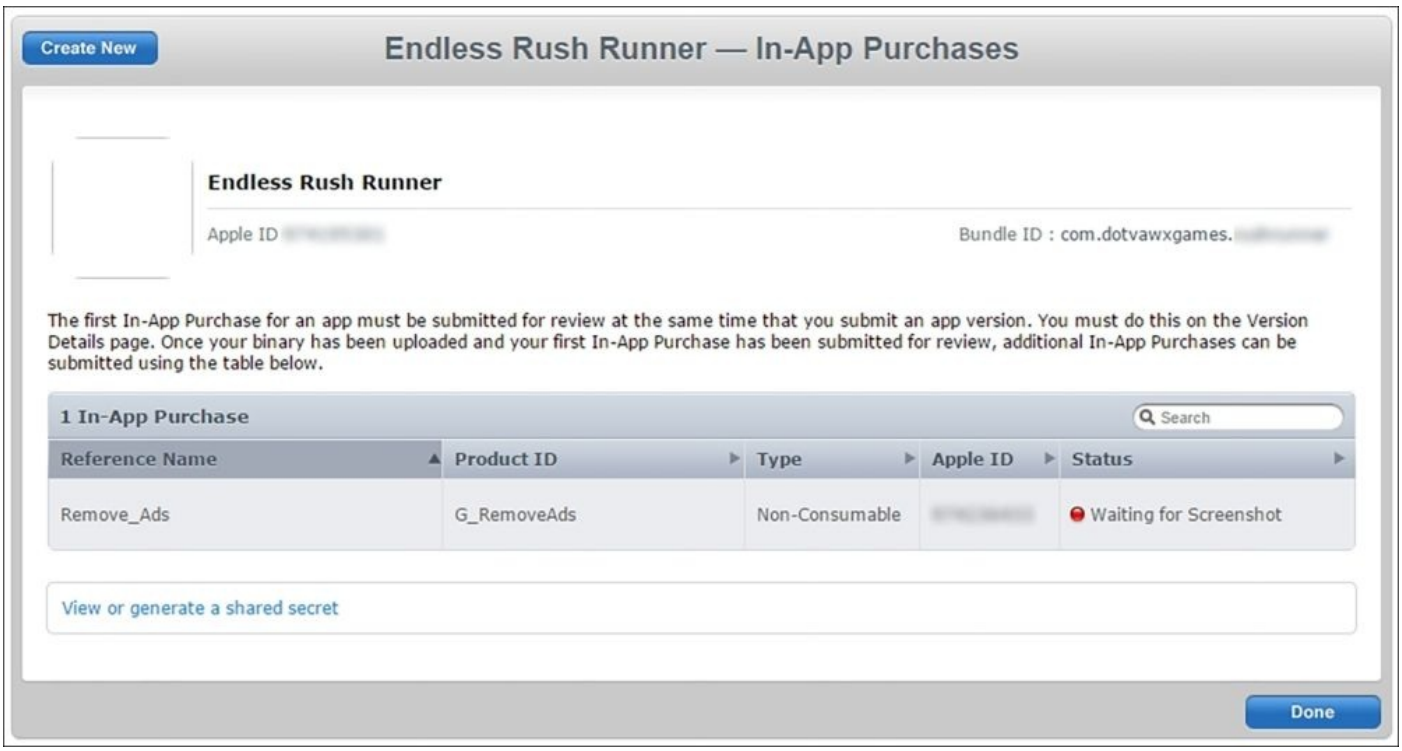
For **Hosting Content With Apple**, select **No**.

There is no need to have **Review Notes**.

We will add a screenshot of the purchase in the game when it is done in a later chapter. As a note, without screenshots, Apple will not be able to put the game on the App Store.

Now, click on **Save**.

The page should load to show the purchase we just made and have the status of **Waiting for screenshot**. We will add a screenshot in a later chapter.




Click on **Done**. This should load you back to the App page. If not, click on **My Apps** and then select **My Apps**. Now, select your game.

# Game Center, leaderboards, and achievements

When you are back at your game settings page, click on **Game Center**. The **Enabled** bar should already be set. If you see **Disabled**, click on it. The bar should turn to **Enabled**, and the color should turn to blue.

The next tab has two options: **Single Game** and **Group of Games**. Select **Enable For Single Game**.

Under **Leaderboards**, click on **Add Leaderboard**.

On the next page, under **Single Leaderboard**, click on **Choose**.

The next page will load with similar settings as the in-app purchase. For **Leaderboard Reference Name**, enter **Distance**.

For **Leaderboard ID**, enter `G_Distance`.

The **Type** of **Score Format** is **Integer**. This allows you to have a value that keeps track of the distance the player has moved. By whole distance, I mean a nondecimal-based value, such as 1, 2, 3, 4, and so on.

The **Type** of **Score Submission** should be set to **Best Score**. Leave this as it is. This way, we can save the distance with the player, save the profile locally, and only submit the score when its distance is greater than what they had done previously.

**Sort Order** is the order in which the leaderboards will appear. As the distance we are looking for is the highest value, change this to **High to Low**. This way, the player with the farthest distance will be at the top of the list.

As we have not programmed the value to calculate the distance, go ahead and leave **Score Range** blank. This will give us plenty of room to work with.

At the bottom of the window, click on **Add Language**.

Set **Language** to **English** and **Name** to **Distance**.

**Score Format** is a way to set a difference in the formatting of the score. For example, if our score was based on money, we would want to use a comma to separate the numbers. However, as we will use this value as a distance, we need to separate the values with a decimal point. Select **Fixed Point (10.000.012,218 )**.

The **Score Format** suffix is what will be added to the end of the score. Like before, if we were using money, we could end the formatting with “Dollars “or “Coins”. As we will use this as a distance traveled, enter **Yards**.

Leave the **Image** field blank.

Then, click on **Save**.

The page should load back to the **Add Leaderboard** page with the information you have set.

Now, click on **Save**.

## Tip

For the purposes of this book, one leaderboard is enough. Feel free to add more though! You could include things, such as “Deaths”, “Pickups Collected”, or “Attempts”.

The page should load back to the Game Center page. If not, go back to **My Apps**, select your app and then **Game Center**.

For the achievements, we will set three of them. One for surviving “100 yards”, one for “collecting 100 pickups”, and one for “Play 10 Rounds”. At the bottom of the page, click on **Add Achievement**.

For **Reference Name** of **Achievement**, enter `100Yards`.

For **ID** of **Achievement**, enter `G_100Yards`.

Again, usually games would refer to these IDs with a format such as `<<gamename>>.G_100Yards` or literally `gamename.G_100Yards`, but because these names have to match the code that we are writing together and the game name is going to vary, you can use the direct achievement name: `G_100Yards`.

**Point Value** is the number of achievement points the Game Center account will be credited with when the achievement is collected. The value is completely arbitrary; you can enter anything you want. I will use `10`.

## Tip

Make sure to leave room for the next two achievements because if you use all 1000 points with this one, you won’t be able to make any more achievements.

Select **No** for **Hidden**. If you use **Yes**, the achievement won’t be in the list of achievements for our game until they have earned it if the user was to look at Game Center. As I would like the players to see their achievements, they can earn before earning them; selecting **No** will allow them to do this.

For **Achievable More Than Once**, select **No**. This is a one time achievement.

At the bottom of the page, click on **Add Language**.

The **Title** is `100 Yards`.

The **Pre-earned Description** tab is what will appear in Game Center if the user is to look it up. You can enter what you like to describe this achievement. I will use `Survive for 100 yards`.

The **Earned Description** tab is what will display on Game Center after the user collects the achievement. Again, this is open. I will use `You survived 100 yards!`.

A package of art is included with this book. In the `iTunes_Connect_Achievements` folder, use the image called `100Yards.png` for **Choose File**.

## Tip

### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

The three files look similar to the following image:

Click on **Save**.

Back on the **Add Achievement** page, it should list the information for the `100 Yards` achievement. Then, click on **Save**.

Follow the same method for the next two achievements with the following settings:

- **[100 Pickups]**:
  - **Reference Name**: `100Pickups`
  - **Achievement ID**: `G_100Pickups`
  - **Points**: `10`
  - **Hidden**: **No**
  - **Achievable More Than Once**: **No**

  The following is the **Language** setting for `[100 Pickups]`:

  **Language**: **English**

  **Title**: `100 Pickups`

  **Pre-earned Description**: `Collect 100 Pickups`

**Earned Description**: `You have collected 100 Pickups!`

**Image**: `100_Pickups.png`

- **[Play 10 Rounds]**:
  - **Reference Name**: `10Rounds`
  - **Achievement ID**: `G_10Rounds`
  - **Points**: `10`
  - **Hidden**: **No**
  - **Achievable More Than Once**: **No**

  The following is the **Language** setting for `[Play 10 Rounds]`:

  **Language**: **English**

  **Title**: `10 Rounds`

  **Pre-earned Description**: `Play 10 rounds`

  **Earned Description**: `You have played 10 rounds!`

  **Image**: `10_Rounds.png`

Go back to the main page for your app and scroll down to where you see Game Center. The button should be grayed out. Click on it and it should turn green.

Click on the + sign next to **Leaderboards**. In the window that pops up, select the single leaderboard that we set up and then click on **Done**.

Now, click on the + sign next to **Achievements**. Select all the achievements we set up and then click on **Done**.

At the top of the screen, there is a **Save** button. Click on it to save the settings.

## Tip

Once your game goes live, the achievement IDs and the point values for them is impossible to change. Make sure that during testing, you decide on these options as soon as the game is on the store; you will *never* be able to alter them.

# The Main App settings

After you are done with the achievements, go back to the game app main page. This page is used to show all the public information for the game; the information users will see when the game appears on the store:

- **App video Preview and Screenshots**: This is the area where you can place screenshots and a video preview of the game. Apple requires different image sizes for all their devices. If you want your app to show images, you will need to create, scale, and upload all the images here. The video preview also has to be in a Mac-friendly codec, such as M4V. The screenshots for the app are required, and your game will not be able to go live until you have them included in these slots; the video is up to you and not required by Apple to submit your game.

  ### Tip

  If you want to convert a AVI file to a M4V file, you can use a program called **HandBrake** (https://handbrake.fr/).

- **Description**: This specifies the description of your app. This area is open for you to describe the game and why the user may be interested in it.
- **Keywords**: This denotes the list of keywords that describe your app. When the user searches, these keywords are used to match your app to that search.
- **Support URL**: This specifies the URL that a user can go to get help with the app. This is generally the studio website or the creator's personal e-mail.
- **Marketing URL**: This is the URL that would be used as marketing for your app. This is optional.
- **Privacy Policy URL**: This would link the user to the privacy policy of your game. This is required for apps that are made for kids, if you offer autorenew in-app purchases, or if you have free subscriptions. If you include account registration, an app that would access the users existing account, or if (by chance) your app is required by law, this is also required.
- **App Icon**: This is the icon of your app. It is what the user will see when it appears on the store.
- **Apple ID**: This specifies the ID of this app.
- **Version**: This is the current version of the app.
- **Category**: This specifies the category that the app falls into. For us, it is "Games".
- **Rating**: This denotes the rating of this app.
- **License Agreement**: This specifies the standard license agreement between you and Apple.
- **Copyright**: This denotes the name of the person or entity that owns the rights to this app. In the example of this book, you would probably just use your name.
- **Trade Representative Contract Information**: This allows you to display additional information that would be displayed on the Korean App Store.
- **Name, Address, and Country**: This specifies your personal information about who owns the app.

- **Routing App Coverage File**: This allows you to specify the geographic regions supported by your app.
- **Build**: This is the current build and is uploaded with Xcode 5.1.1 version or later or the Application Loader 3.0 version.
- **Game Center**: This denotes the achievements and leaderboards used for this game. You set these earlier.
- **Multiplayer Capability**: This specifies the games that are multiplayer-capable. Keep this as it is.
- **Newsstand**: Keep this unused.
- **App Review Information**: This specifies the information that the Apple review team will use to contact you if they need any information during the review process. You'd generally use the name of the person in charge of quality assurance. In my case, I will use my name and e-mail.
- **Demo Account:** This is used if the game needs an extra account to play, such as if you had a multiplayer service that required a log in, you would give the reviewers a demo account so that they could test the game with the extra service. For us, we can leave this blank.
- **Notes**: This specifies the additional information that the review team may need to test the app. As our game will be on the simple side, you can leave this blank as well.
- **Version Release**: This is the setting you can use to either automatically release the game as soon as it clears review or to set it up, so Apple lets you know when the game has cleared review. This is a personal choice. I generally use **Manually release this version** to make sure that everything is correct before it goes to the store.

At the very top of the page, click on **Save**.

# Enabling iAds

At the top of the page, click on **My Apps**, and in the window that pops up, click on **Agreements, Tax, and Banking**.

On this page, you will have to fill out and accept the contracts for **iOS Free Application**, **iOS Paid Applications**, and **iAd App Network**. The process includes the personal banking information and the identity verification. When you have completed these, the game will be set up to sell and display iAds.

# Setting up iOS Native in Unity

As Unity does not support the native functionality of iOS, we need a plugin to call the native iOS functions from Unity. For example, if we want to provide the user with the in-app purchase, we need this plugin to handle the code that takes care of this.

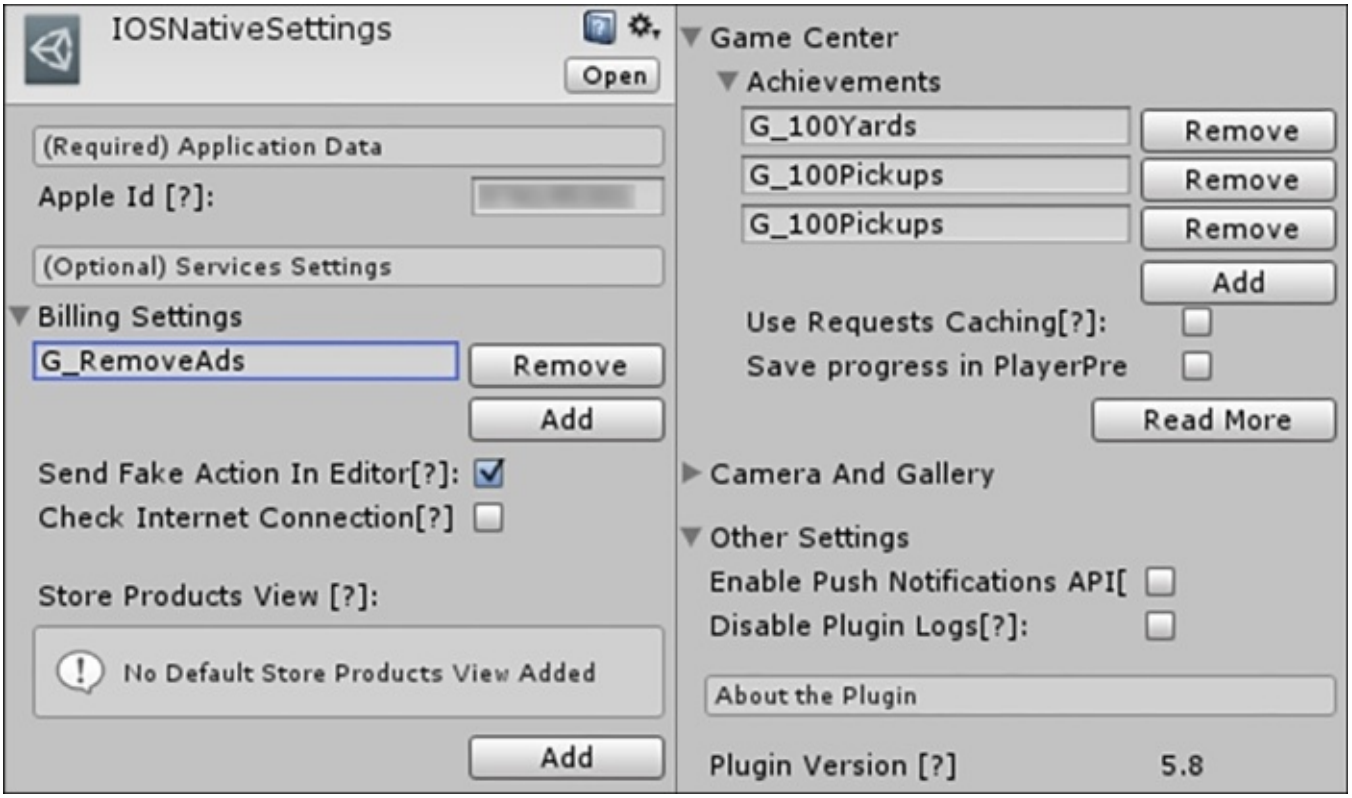
For our plugin to know about the specifics of our game, we must first include the Apple ID and the Bundle ID.

In Unity, click on **Window** at the top of the editor screen, navigate to **iOS Native**, and select **Edit Settings**.

The **Inspector** window should update to show **iOS Native Settings**.

In iTunes Connect, navigate to **My Apps** and click on your game. This will open the main settings page. About halfway down, you will find the text for **Apple ID**. This is a big integer value. Copy this and go back to Unity; you can select the text and click on *Ctrl + C*, or you can write it down.

In **iOS Native Settings** in Unity, under **Apple ID**, paste or enter the **Apple ID** from iTunes Connect:




While still in Unity, click on **File** in the top-left corner of the editor window. Then, navigate to **Build Settings**.

If your project is set to **PC, Mac & Linux Standalone**, click on **IOS**. In the bottom-left corner of the window, click on **Switch Platform**.

Next, in the same window, click on **Player Settings**. This will update the **Inspector**

window. Under **Other Settings**, find the section under **Identification** and then the textbox labeled **PlayerSettings.bundleIdentifier**. In this field, enter the Bundle ID that you set up earlier. I used the **reverse-domain name** style, such as `com.mycompanyname.appname`.

If you do not remember your Bundle ID, navigate to **iTunes Connect**, select **My Apps** and then select your game. On the right-hand side of the window, click on **More** and then **About this app**. In this window, you will see the game information, including the Bundle ID of the game.

You can leave **PlayerSettings.bundleVersion** set to `1.0`.

## Tip

iOS Native is recommended even with the extra cost. If you do opt not to use it and try a free version, you will have to find some of these settings for this plugin.

# Certificates and provisions on Mac

Apple has come a long way when it comes to making the provisions and certificates for games. This used to be a headache, but now, we can use Xcode to generate these for us.

The first step is that you will need to make a build and open it in Xcode. This can be done from your Windows or Macintosh machine, but it has to be opened on your Mac.

To create the build, open Unity and navigate to **File** in the top-left corner of the editor window. Then, go to **Build Settings**.

Make sure that the build is **iOS** and **Development Build** is checked.

As there is no reason to run this build on a device, you do not have to have a scene selected in the **Scenes in build** window.

Open **Player Settings** and make sure that **PlayerSettings.bundleVersion** is `1.0`, and **PlayerSettings.bundleIdentifier** is set to the Bundle ID of your app.

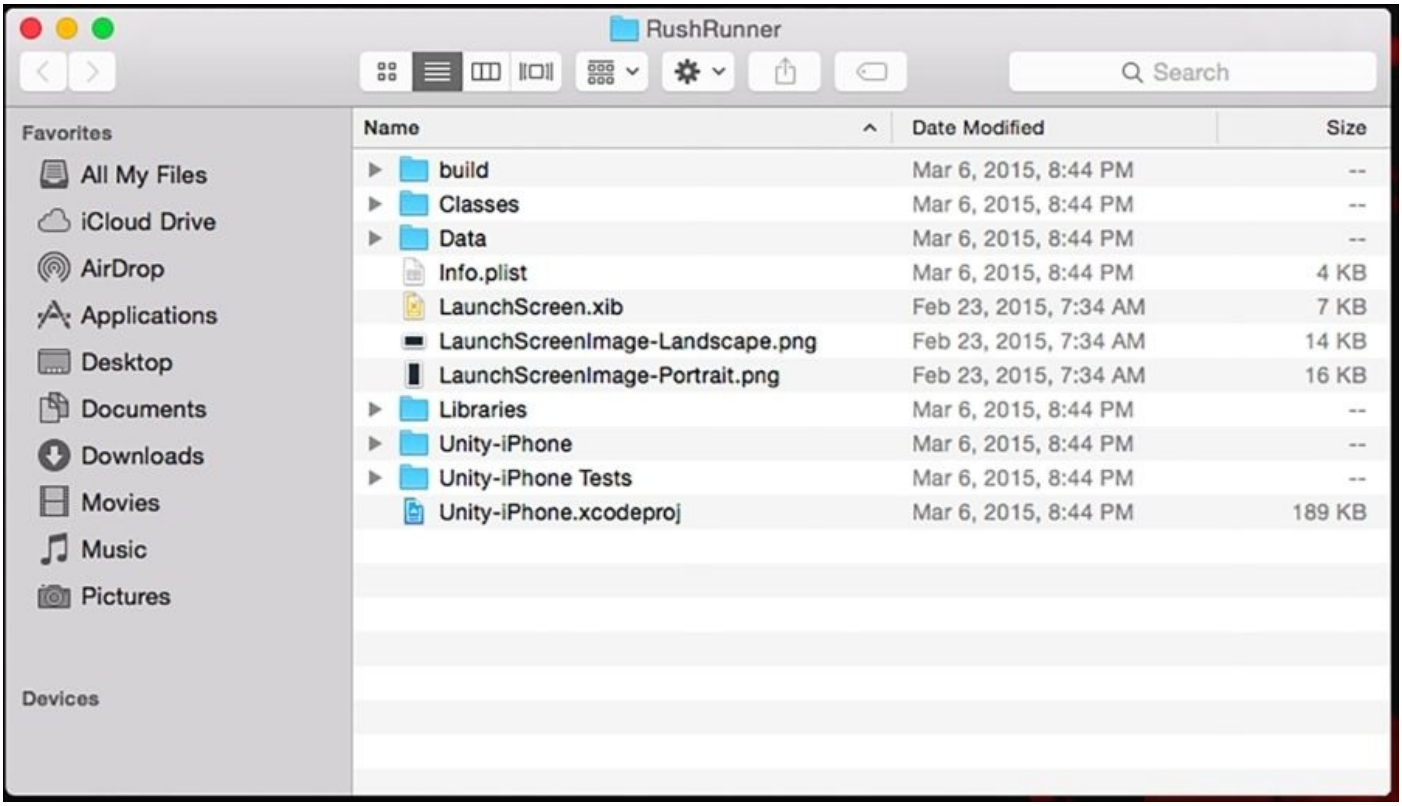
Under **Optimization**, there are a few settings that you can keep as default so that you know what the important ones do:

- **API Compatibility Level**: This allows you to set the .NET API profile. The .NET 2.0 API is the full .NET compatibility, whereas the .NET 2.0 subset is the subset of the full .NET compatibility and has smaller file sizes.
- **Target iOS Version**: This selects the iOS version you want to target your game to. The lower the version number you use, the more devices can run your game. Using a higher version means that you have the option to utilize the features of this higher version, but users who have not upgraded their device to run this version will not be able to run your game.
- **Stripping Level**: This option allows you to have the engine remove the unused aspects of the code to optimize the file size of your game.
    - **Strip assemblies level**: This will look at the scripts by the code and remove any classes and methods that are not being used.
    - **Strip ByteCode level**: This takes any of the .NET DLL files and strips them down to metadata only.
    - **Use micro mscorlib level**: This is a special option to use a small version of mscorlib. This does remove some features, including security, Reflection.Emit, remoting, and nonGregorian calendars.

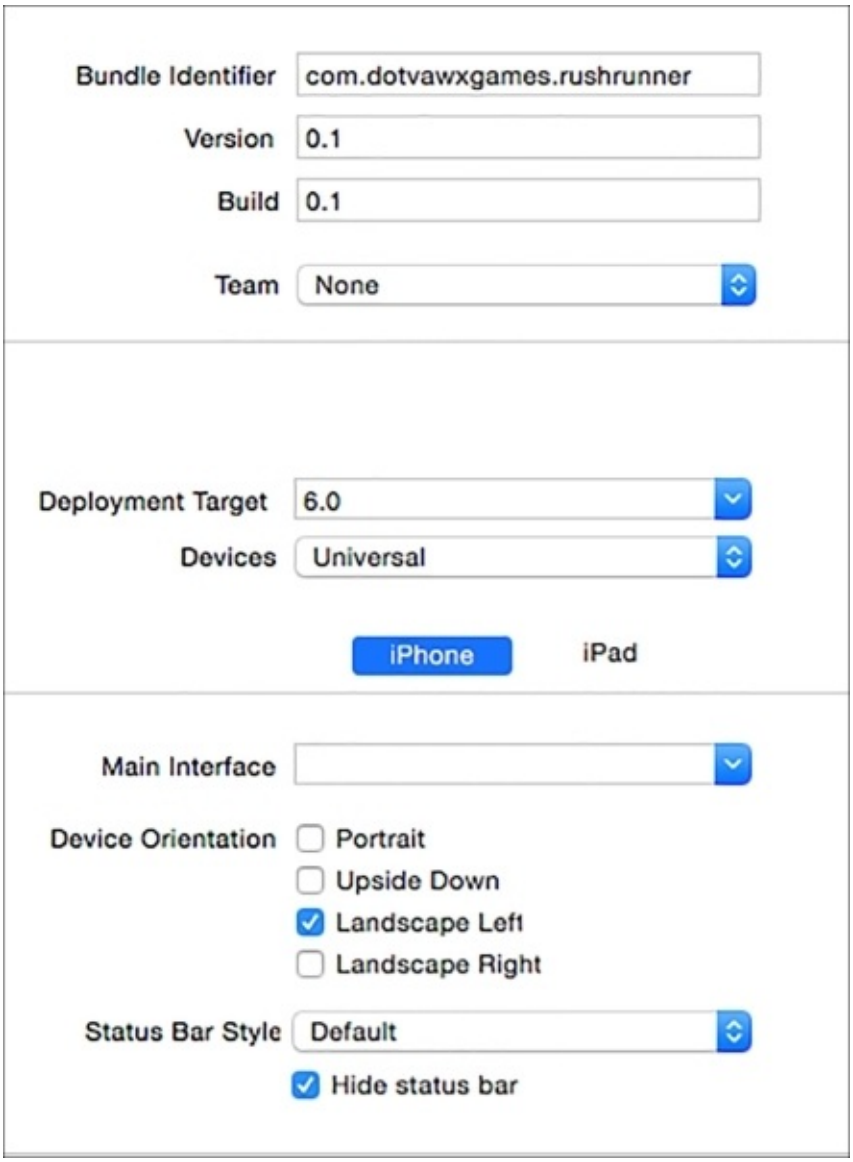
When everything is set, click on **Build**. This will prompt you to save the build at a location on your computer. It doesn't matter where.

When the build is done, navigate to where you saved it. If you are *not* on a Mac, you need to copy this whole folder onto your Macintosh machine. You can do this with version control, a local network, or you can zip it up and send it to yourself and then download it on your Mac.

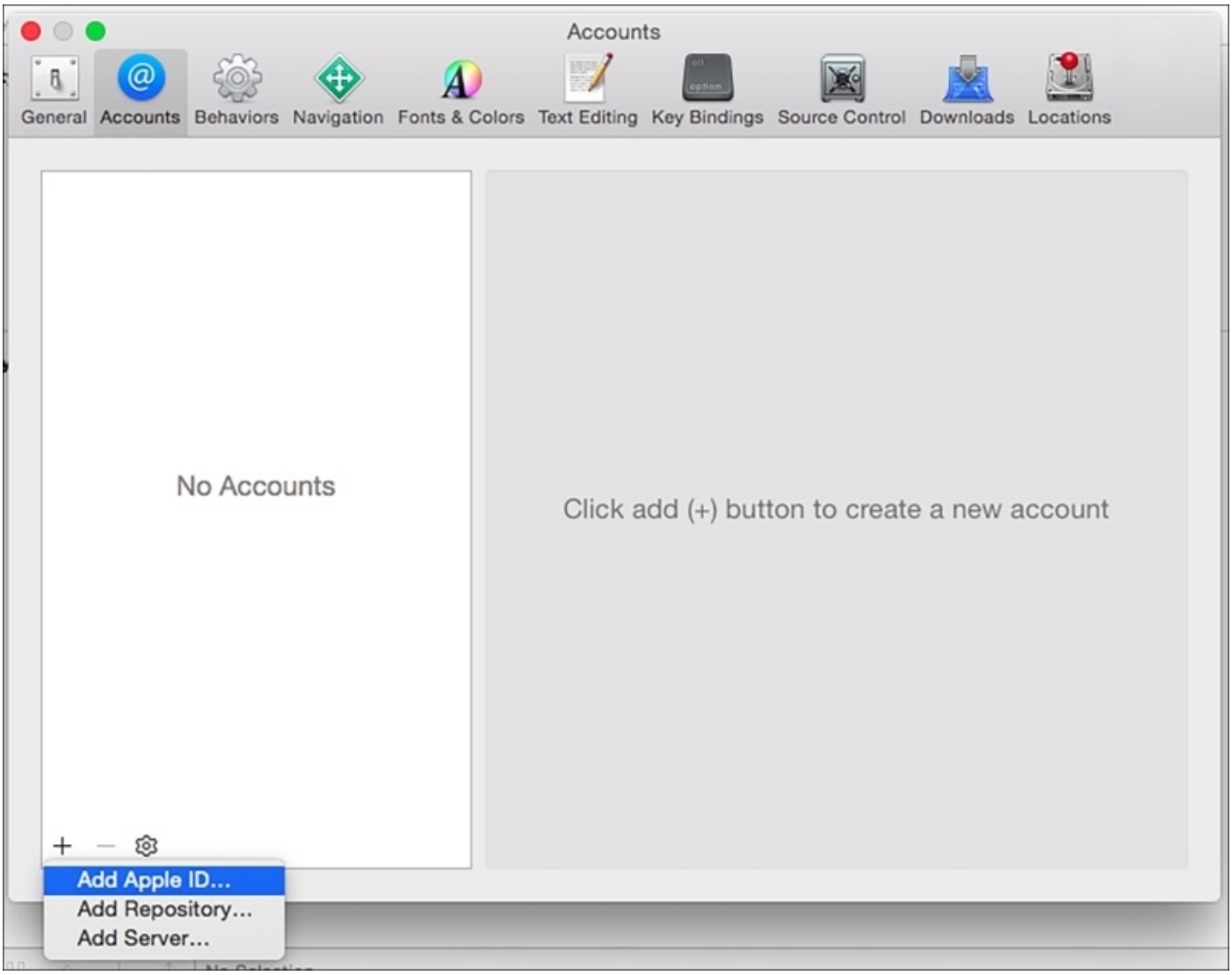
In order to open the project in Xcode, open the folder and double-click on the file that ends with **Xcodeproj**. This will open Xcode and your project.



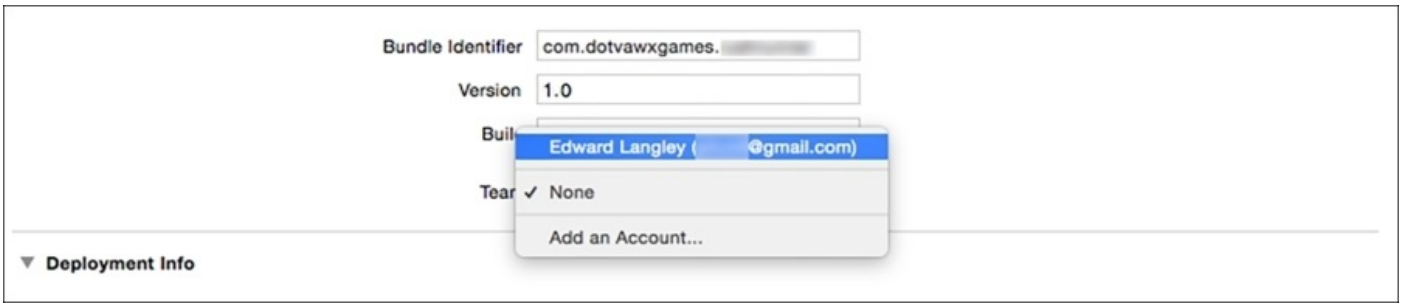

In the **General** tab in Xcode, you will see the project general settings. This includes things, such as **Bundle Identifier**, **Version**, **Deployment Target**, and the type of devices the game will run on.




Under the **Identity** portion of the **General** settings tab, there is a drop-down menu for **Team**. Click on it and then select **Add an account**. This will open the **Accounts** window. In the bottom-left corner of this window, click on the small + sign and select **Add Apple ID…**. This will prompt you to log in with your developer credentials. Use these credentials to log in.




After logging in, you can close the **Accounts** window. Now, from the **Team** drop-down list, you should be able to change it from **None** to your team name, which is probably the name and e-mail you used to create your account:



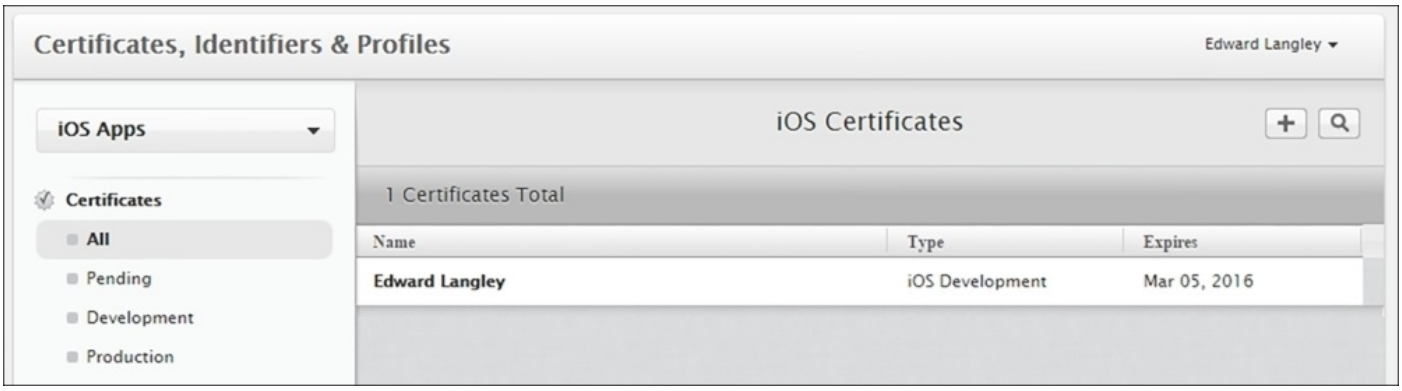

To make sure the certificate went through, visit [https://developer.apple.com](https://developer.apple.com).

At the bottom of the page, click on **iOS Apps**.

On the following page, if not logged in to your developer account, log in. After this, in the top-right corner, click on the link called **Certificates**, **Identifiers & Profiles**.

Under **iOS Apps**, click on **Certificates**.

When the page loads, you should have **All** selected in the **Certificates** section. On the right-hand side of your screen, you should see your name, the type of certificate, and the expiry date (1 year after the creation of it):




Through the course of this book, we will use two different types of certificates: iOS Development and iOS Distribution.

**iOS Development** refers to a certificate you can use during development, and **iOS Distribution** is the certificate that can be used when you are ready to submit your game for distribution.

# Why iTunes Connect is important

The process of setting up iTunes Connect and Developer Account is important because these are the settings we will use in Unity in order to unlock achievements, leaderboards, in-app purchases, and display iAds.

If we didn’t go through the process of creating an app in iTunes Connect, it would be impossible to release a game on the App Store, and if we didn’t set up the Game Center features, we wouldn’t be able to have achievements or leaderboards. The same is true for in-app purchases and iAds.

# Summary

In this chapter, we looked at basics of how to set up Unity and the iTunes network in order to create the iOS game. Apple has come a long way since the inception of iOS development, but the process can still be quite tedious.

In the next chapter, we will go through the game and begin to develop the input controller, which will be used to manipulate our game by the end user.

# Chapter 2. Player Input for Mobile Devices

One of the more unique aspects of mobile gaming is the input system. Unlike most other devices for gaming, mobile devices are not designed to take input (such as traditional gaming systems) and are instead in the spectrum of an entertainment device, meaning they are not solely used to play games. This means that we have to observe how the player is going to interact with the game differently as compared to devices specific to gaming—which have either dedicated or what you'd call standard input schemes that allow the player to have a pleasing experience.

The most successful games on the mobile platform do one thing well: they give the player a simple and responsive input. For example, games focused on puzzles such as connect three items of one type in a row by swiping types between each other, a slot machine, and casino games, which require the touch of a single button, or even the more complex games (such as the endless runner) where the player can control the game with taps and swipes that have a much better chance of being successful than a game that uses arrows or joysticks to move and multiple buttons to cause the character to do something. It's not to say that you can't have complex movement of the character in a game; it is more to say that if you do, solve the issue with the most basic input scheme possible.

For the game in this book, the only input will be of a swipe. The player will swipe to have the character on screen to jump. If on a wall, jump to another wall, continue forward, or swipe down. Swiping down while already jumping will also cause the character to fall to the ground faster.

For the sake of teaching, the code represented in this chapter will also detect when the player taps on the screen. This is included here in the hope of teaching you how to include different types of input if you were to make another game after following along with the instructions here.

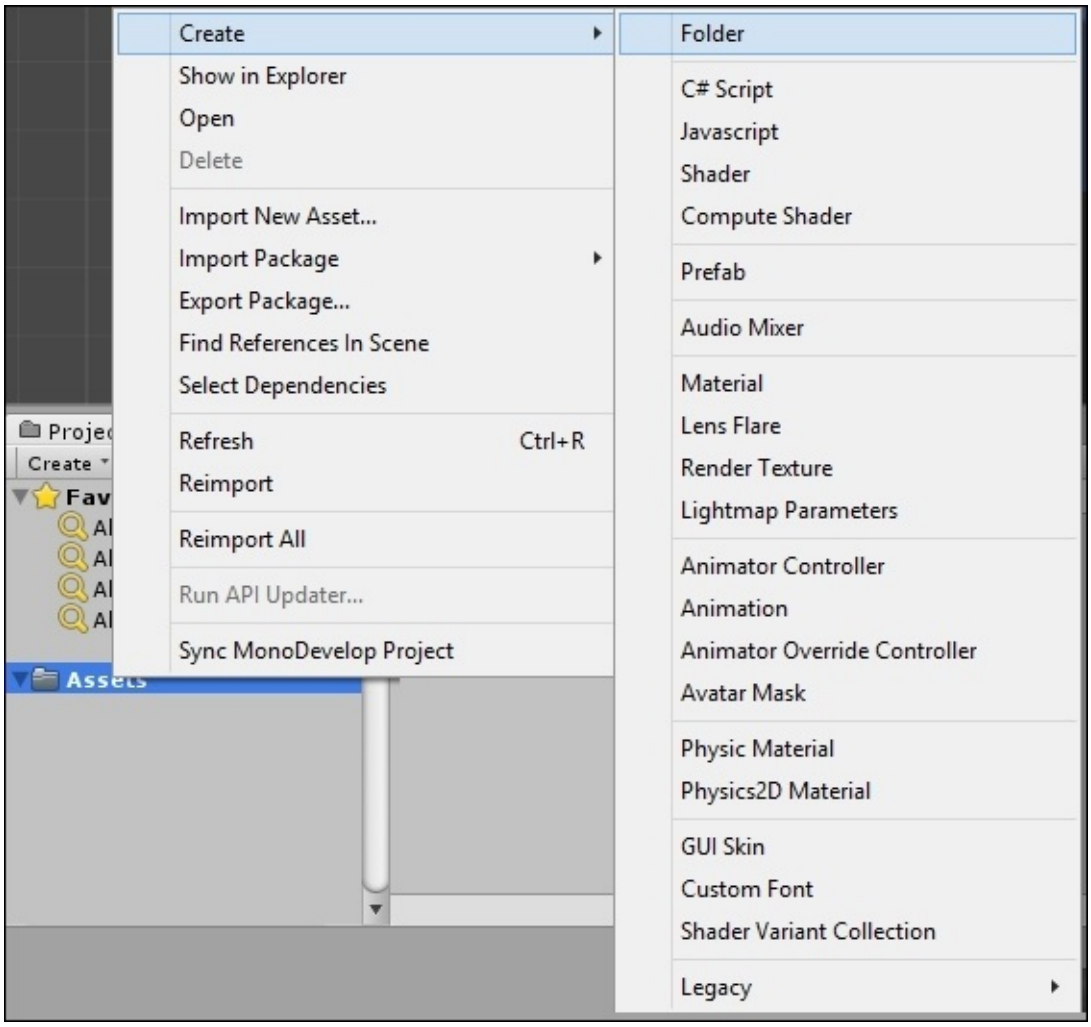
To sum up, we will cover the following topics:

- How Unity uses their `Touch` struct, and how we will use it to create our own struct called `SimpleTouch`
- Using the same technique to take in the mouse input so that we can test the game in the editor
- How the character on screen will know what the input is doing?

# Creating the C# input class

The very first step is to create the C# class that will calculate the input from the player. To start, open Unity with the following steps:

1. Right-click on the **Assets** folder and select **Create** and then **Folder**:




2. I named my folder `Scripts`; you can give it any name you like, although I recommend something that keeps you organized.
3. When this folder is created, right-click on it. Then, navigate to **Create** and select **C# Class**.
4. You can again name this what you like; I chose `PlayerInput`.

Unity ships with its own **Integrated Development Environment** (**IDE**) called **MonoDevelop**. When you double-click on `PlayerInput`, this is what will open.

## Tip

If you want to use a different IDE, you can do so by navigating to **Edit** | **Preferences** | **External Tools** and changing **External Script Editor** to something of your choice.

Now, double-click on **PlayerInput**.

When **PlayerInput** opens, it will have a default structure that Unity uses for classes created in the engine, as shown in the following code:

```
using UnityEngine;
using System.Collections;
public class PlayerInput : MonoBehaviour
{
  /* Use this for initialization */
  void Start ()
  {
  }
  /* Update is called once per frame */
  void Update ()
  {
  }
}
```

This code is the very basis of what Unity gives you.

- `Start()`: This is called when an instance of this class is created. It is generally used to set up references or other variables when the class first starts running in the game.
- `Update()`: This is called for every **tick** of the game. A tick is the time it takes for a device to render a frame. This means that if the game is running at 30 frames-per-second, `Update()` will be called 30 times in that second. `Update()` is what we will use to calculate the types of input the player will use.

# Including system

Before we start coding, we must include something in the class for a feature that is not defaulted. Enter `using System;` at the top of the input class under the following code:

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

Make sure to have all three of these included in the class.

Before we begin, we must first understand what Unity will give us when we look for the **Touch** input from the player.

# What is Touch?

Touch is a **struct**. To put it differently, it is a group of different variables that hold information about what the input can do. It is possible to monitor multiple `Touch` structs at once if the user is trying to input more than one, although we are only looking for one.

We do not need all the information Unity's touch gives us. We only need the location on the screen where the touch started, how long the touch lasted in seconds, where the touch ended on the screen, and the current state of the touch. From this, we will be able to detect the type of input and the direction of the input, if the player tapped, or if the player swiped in a direction.

# Our own Touch

The first step to coding our input class is to create our own struct that will hold the values we need. The benefit to creating our own Touch is that we can then use it for only what we need. It also gives us the means to use the mouse input to accomplish the same input logic as the device, allowing editor testing that will match the device.

At the top of the `PlayerInput` class, enter the following code:

```
public struct SimpleTouch
{
  public Vector2 StartTouchLocation;
  public Vector2 CurrentTouchLocation;
  public DateTime StartTime;
  public TouchPhase Phase;
}
```

As stated previously, we need to know the location where the touch started, ended, how long it lasted, and the current state (or phase) of the touch.

- `Vector2`: This is a struct that holds `X` and `Y` float variables and is generally used to store a 2D location.
- `DateTime`: This class is used to manage the date and time in your code. We will use this so that we know at what time the touch input started. Then, when you subtract the time the touch ended, we will get the length of the input in time.
- `TouchPhase`: This enum is used to categorize the state of the input. We will use this to compare the touch input from Unity with our `SimpleTouch`.

# Global variables

We will also need to include a few settings that we can use to manage how we detect a swipe. Under our `SimpleTouch` struct, enter the following code:

```
public float SwipeTime;
public float SwipeDistance;
```

These two values, which we can set from within the editor because they are **public**, allow you to fine-tune what we call a Swipe.

We will also need two variables to manage the touch input. The first one will be the one Unity gives us, and the second variable will be the `SimpleTouch` struct we made. To add these, enter the following code under the two previous variables:

```
private SimpleTouch ActiveTouch;
private Touch DeviceTouch;
```

- `ActiveTouch`: This specifies the simple information that we need to detect a swipe
- `DeviceTouch`: This denotes the raw touch information Unity gives us when the player touches the device screen

### Tip

These variables are set to **private** to prevent both the editor from showing them and prevent any other class from accessing them. To put it differently, we don’t want any other source to manipulate these values, so we label them as private.

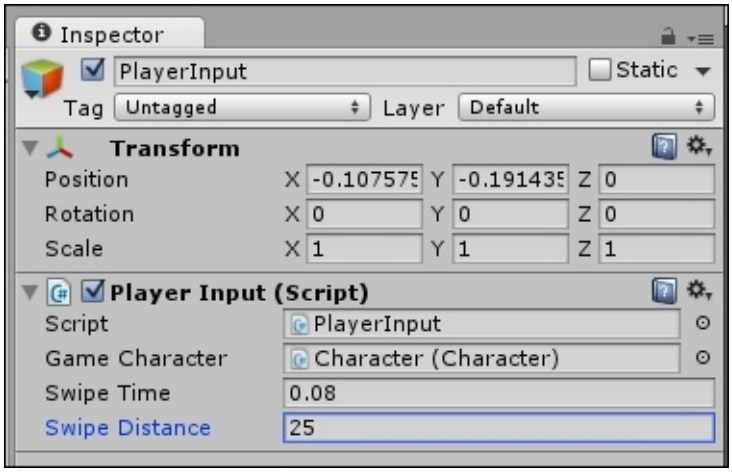
# Creating the Input GameObject

Save the document and then maximize or select Unity. This will force the editor to compile the code that we have just written.

1. On the left-hand side of the editor in the **Hierarchy** tab, right-click and select **Create Empty**. This will add a blank GameObject to the scene.
2. Right-click on the GameObject that we just created and select **Rename**. Name it `Input`.
3. While still having the `Input` GameObject selected, look to the right-hand side of the screen in **Inspector** and click on **Add Component**.
4. When the search window pops up, enter the first half of what you called your input class. For example, when I enter `player`, my **Player Input** class will appear in the list:

5. Click on your input class to add it to the `Input` GameObject. If everything goes according to plan, you will see the component added and **SwipeTime** and

**SwipeDistance** both set to `0`. Alternatively, you can add the `Input` component to GameObject by left-clicking, holding, and dragging it onto GameObject in the **Hierarchy** tab from the **Assets** | **Scripts** folder.




6. For now, set **Swipe Time** to `0.08` and **Swipe Distance** to `25`.

# Calculating the TouchInput function

We can now create the function (also called a method) to take in the touch information so that we can check it. To start, enter the following code under the last two variables:

```
private void CalculateTouchInput(SimpleTouch CurrentTouch)
{
}
```

- `void`: This a keyword used to identify the function as something that will not return a value. As this function is designed to take in data and manage it within its own **scope**, we don't need it to return any data back.

  ### Tip

  Scope is the range of code that the brackets will manage. In this case, the brackets of `CalculateTouchInput` will manage detecting specific types of touch input. Variables such as `ActiveTouch` and `DeviceTouch` are what is called Global Scope. This means that the entire class can read and manipulate them, whereas something such as `SimpleTouch CurrentTouch` are only seen and manipulated in `CalculateTouchInput`.

- `CalculateTouchInput`: This is the name of our function. This can be anything you want, although the naming function should be done in a way that gives some clue as to what it will do. In our case, it will check the input data and calculate what to do with it.
- `(SimpleTouch CurrentTouch)`: This portion of the function is what is called an argument or parameter. It is a way to send data to this function so that it can be used or altered.

Next, we will use `CalculateTouchInput` to detect what type of touch is being sent to the function based on the data of `CurrentTouch`.

Add the following code in the `CalculateTouchInput` function:

```
private void CalculateTouchInput(SimpleTouch CurrentTouch)
{
  Vector2 touchDirection  =
  (CurrentTouch.CurrentTouchLocation -
  CurrentTouch.StartTouchLocation).normalized;

  float touchDistance =
  (CurrentTouch.StartTouchLocation -
  CurrentTouch.CurrentTouchLocation).magnitude;

  TimeSpan timeGap = System.DateTime.Now - CurrentTouch.StartTime;

  double touchTimeSpan = timeGap.TotalSeconds;

  string touchType =
  ( touchDistance > SwipeDistance && touchTimeSpan > SwipeTime )
  ? "Swipe" : "Tap";
```

```
  print( touchType );
}
```

As mentioned before, we will need a few things to tell what type of input we are calculating.

- `touchDirection`: This is the value of the current location of the touch *minus* the start location of the touch.
- `normalized`: This converts the vector, which can have very large numbers, to a vector that has a magnitude of one—or all the values in the vector are no more than 1.0 or less than -1.0. For example, for `touchDirection`, we need the direction in which the player swiped. To do this, we subtract the current location from the start location of the touch. This vector value still holds the same direction between both the points; however, now the values of X and Y are normalized to `0.0` and `1.0`, respectively.

  Let's look at it like this. If you were to have a vector with the Y value of `1.0` and the X value of `0.0`, you know that the direction of the vector is vertical and going upwards in the direction of Y. Now, if you have another vector with the Y value of `0.8` and the X value of `0.2`, you can still see that the direction is moving upwards in the direction of Y, but it is also slightly moving towards positive X. As we have the start and current location (or when this is calculated, the end location as `CalculateTouchInput` is being called when the user lets go of a tap), we can subtract them from each other and get the direction between them. When we normalize this value, we will still have the same direction; only the vector member values are within -1 and 1 instead of something much larger, which is what we'd have if we didn't normalize the vector.

  ### Tip

  This example is set to `Vector2`, although the same is represented in Vector3, but with the added value of z.

- `touchDistance`: This is the total **magnitude** of the start and current touch locations. Unlike what was mentioned before in `touchDirection`, this magnitude represents the entire length value between the two vector points. For example, say you have a screen resolution of *400*400*. If the user taps at *0,0* and drags their finger to *0,200*, the magnitude of this swipe will be `200` because the total value of this swipe is *0 + 0* for X and *0 + 200* for Y.
- `timeGap`: One of the factors that decide, the difference between a swipe and a tap is the length in time the users finger was on the screen. We can have this value by first storing when the tap started and then subtracting it from the current time on the device.
- `touchTimeSpan`: As the `TimeSpan` struct is compatible with `System.DateTime.now`, we will be able to subtract one from the other. `TimeSpan` then allows you to convert this time difference directly to seconds, which is what we do to get `touchTimeSpan`. Once we have this, we know exactly how long the user had their finger on the screen. Coupled with `touchDistance`, we can decipher the type of touch.
- `touchType`: We will be able to tell you what type of touch is calculated by making

sure that `touchTimeSpan` is greater than **SwipeTime** and **touchDistance** is greater than **SwipeDistance**. If this comparison is `true`, we will assign **touchType** as **Swipe**. If it is set to `false`, we will set it to **Tap**.

- `print(touchType)`: This function allows you to print text to the console window in the Unity editor. Print is a handy way to display some debug information in the editor so that you can see the value of something directly.

Alternatively, you can also use `Debug.Log( )` to display the values in the console log.

# The Start function

Before we get to the core of the input logic, let's first set a default value to our `ActiveTouch` variable. As we don't want the game to start running with `ActiveTouch.Phase` to be set to anything but canceled, we need to set it.

Add the following code to the `Start` function:

```
/* Use this for initialization */
void Start ()
{
  ActiveTouch.Phase = TouchPhase.Canceled;
}
```

## Tip

By default, this would be set to `TouchPhase.Began`, which would cause an unwanted touch to be detected when the game loads.

# Starting to work with Update

Next, we will set up the `Update` function to look after any touch input from the user. In order to do this so that it works in the editor and on the device, we will need a bit of logic to repeat itself. This is because the mouse input isn't related to the touch input.

To begin, add the following code to the `Update` function:

```
  void Update ()
  {
    if (Application.isEditor)
    {
      if (Input.GetMouseButton(0))
      {
      }
    }
    else
    {
      if (Input.touches.Length > 0)
      {
      }
    }
  }
```

In this comparison, we will check whether the game can run in the editor. If it is not, we run what is in `else`.

If `isEditor` is true, we then check whether `MouseButton(0)` is pressed. Otherwise, as most people know, left-click on it.

If `isEditor` is false, we can check whether `touches.Length` is more than zero. If it is more than zero, we know that there is a touch on the screen.

Furthermore, you can also use platform-dependent conditions to check for specific devices. For example, you could use the following code:

```
#if UNITY_IOS
  // Code goes here
#endif
```

## Tip

There are many platform-specific conditions you can use in this way, which can be found at http://docs.unity3d.com/Manual/PlatformDependentCompilation.html.

Given the context of the game we are making and the level of knowledge that is required to follow along, `Application.isEditor` is suitable.

# The in-editor input logic

Under `Application.isEditor` and inside `Input.GetMouseButton(0)`, add the following code:

```
if (Input.GetMouseButton(0))
{
  if (ActiveTouch.Phase == TouchPhase.Canceled)
  {
    ActiveTouch.CurrentTouchLocation = Input.mousePosition;
    ActiveTouch.StartTouchLocation = Input.mousePosition;
    ActiveTouch.StartTime = System.DateTime.Now;
    ActiveTouch.Phase = TouchPhase.Began;
  }
  else
  {
    ActiveTouch.CurrentTouchLocation = Input.mousePosition;
  }
}
```

As mentioned before, when you go through `CalculateTouchInput`, you will need a few values to compare them with each other to know the type of input we will calculate. This is how they get their value, which will represent a touch.

To begin this block of code, we will first compare `ActiveTouch.Phase` to check whether it is equal to `TouchPhase.Canceled`. If this is true and we know that the left mouse button is down, we know that it is a new touch. As this is a new touch, we will assign the values we need based on the location of the mouse and the time of *now* and set `ActiveTouch.Phase` to `TouchPhase.Began`.

Now, if the mouse button is still down, but `ActiveTouch.Phase` does not equal `TouchPhase.Canceled`, we know that the user sill has an input pressed, so we will update `CurrentTouchLocation` of `ActiveTouch` to represent where their mouse, or finger, is on the screen.

As we have all the information we need for when the user had the finger on the screen, we can now check and handle what happens if they stop the input.

Under the `if (Input.GetMouseButton(0))` code block, add the `else` statement and the following code:

```
else
{
  if (ActiveTouch.Phase == TouchPhase.Began)
  {
    CalculateTouchInput( ActiveTouch );
    ActiveTouch.Phase = TouchPhase.Canceled;
  }
}
```

To keep up, the entire `Update` function should look similar to the following code:

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

```
{
  if (Application.isEditor)
  {
    if (Input.GetMouseButton(0))
    {
      if (ActiveTouch.Phase == TouchPhase.Canceled)
      {
        ActiveTouch.CurrentTouchLocation = Input.mousePosition;
        ActiveTouch.StartTouchLocation = Input.mousePosition;
        ActiveTouch.StartTime = System.DateTime.Now;
        ActiveTouch.Phase = TouchPhase.Began;
      }
      else
      {
        ActiveTouch.CurrentTouchLocation = Input.mousePosition;
      }
    }
    else
    {
      if (ActiveTouch.Phase == TouchPhase.Began)
      {
        CalculateTouchInput( ActiveTouch );
        ActiveTouch.Phase = TouchPhase.Canceled;
      }
    }
  }
  else
  {
    if (Input.touches.Length > 0)
    {
    }
  }
}
```

As `Update` is called for every frame rendered, we want to make sure that code is being run when it is time. In order to make sure that we handled `ActiveTouch` when we need to, we first check to make sure that the mouse button is not down. Once we know that the mouse button is *not* down, we then check whether `ActiveTouch.Phase` is equal to `TouchPhase.Began`.

If this is `true`, we know that the player has just let go of the mouse as we assign `TouchPhase.Began` in the block that checks whether `GetMouseButton(0)` is `true`.

As the player has let go of the input, we can send the `ActiveTouch` data to `CalculateTouchInput` to see what type of touch it is. Once this is done, we can also make sure to assign `ActiveTouch.Phase` back to `TouchPhase.Canceled` because the touch is no longer active.

# Testing in editor

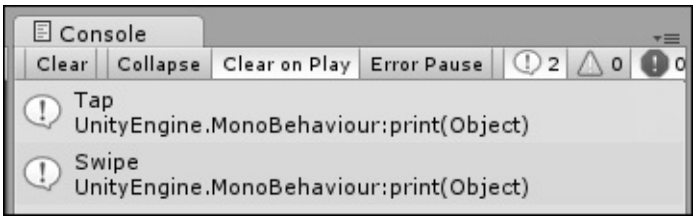
At this point, we can test our code to verify that it works.

## Tip

You should still have the `PlayerInput` GameObject that we created before, but if you don't, here are the steps to create another:

- In the **Hierarchy** window, right-click and select **Create Empty**. Select the new GameObject in **Hierarchy**. Then in **Inspector**, click on **Add Component**.
- Search for `Player`, and **Player Input** should appear in the list. Click on **Player Input** to add it to the components of the **GameObject**.

To test, click on the **Play** button in the editor. It is located at the top center of the editor. Now, if you scroll down and don't move the mouse when the button is down, the console should read **Tap**. If you scroll down and move the mouse when the button is down, the console should read **Swipe**.



# Input for the Touch device

To calculate the type of touch from the touch input, we will follow the same logic. We will only use the touch location to calculate `StartTouchLocation` and `CurrentTouchLocation`.

Add the following code in the `else` block of the `Update` function, after `Application.isEditor` and in `if Input.touches.Length > 0`:

```
if (Input.touches.Length > 0)
{
  DeviceTouch = Input.GetTouch(0);
  if (ActiveTouch.Phase == TouchPhase.Canceled)
  {
    ActiveTouch.Phase = DeviceTouch.phase;
    ActiveTouch.StartTime = System.DateTime.Now;
    ActiveTouch.StartTouchLocation = DeviceTouch.position;
    ActiveTouch.CurrentTouchLocation = DeviceTouch.position;
  }
  else
  {
    ActiveTouch.CurrentTouchLocation = DeviceTouch.position;
  }
}
```

As you can see, this follows the same concept as what we do for when the game is running in the editor. The big difference being that we will now assign `DeviceTouch` to `Input.GetTouch(0)` when there is more than *one* touch active. From this, we can view the position of the touch on screen and update this position for however long the touch exists.

Just as with the mouse input, we need to manage the touch when the user stops clicking on the screen.

Add an `else` block after `if Input.touches.Length > 0`:

```
else
{
  if (ActiveTouch.Phase != TouchPhase.Canceled)
  {
    CalculateTouchInput( ActiveTouch );
    ActiveTouch.Phase = TouchPhase.Canceled;
  }
}
```

# Wrapping up

The entire `PlayerInput` class should look similar to the following code:

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

public class PlayerInput : MonoBehaviour
{
  public struct SimpleTouch
  {
    public Vector2 StartTouchLocation;
    public Vector2 CurrentTouchLocation;
    public DateTime StartTime;
    public TouchPhase Phase;
  }

  public float SwipeTime;
  public float SwipeDistance;

  private SimpleTouch ActiveTouch;
  private Touch DeviceTouch;

  private void CalculateTouchInput(SimpleTouch CurrentTouch)
  {
    Vector2 touchDirection  =
    (CurrentTouch.CurrentTouchLocation -
    CurrentTouch.StartTouchLocation).normalized;

    float touchDistance =
    (CurrentTouch.StartTouchLocation -
    CurrentTouch.CurrentTouchLocation).magnitude;

    TimeSpan timeGap =
    System.DateTime.Now - CurrentTouch.StartTime;

    double touchTimeSpan =
    timeGap.TotalSeconds;

    string touchType =
    ( touchDistance > SwipeDistance
    && touchTimeSpan > SwipeTime ) ? "Swipe" : "Tap";

    print(touchType);
  }

  /* Use this for initialization */
  void Start ()
  {
    ActiveTouch.Phase = TouchPhase.Canceled;
  }

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

```
{
  if (Application.isEditor)
  {
    if (Input.GetMouseButton(0))
    {
      if (ActiveTouch.Phase == TouchPhase.Canceled)
      {
        ActiveTouch.CurrentTouchLocation = Input.mousePosition;
        ActiveTouch.StartTouchLocation = Input.mousePosition;
        ActiveTouch.StartTime = System.DateTime.Now;
        ActiveTouch.Phase = TouchPhase.Began;
      }
      else
      {
        ActiveTouch.CurrentTouchLocation = Input.mousePosition;
      }
    }
    else
    {
      if (ActiveTouch.Phase == TouchPhase.Began)
      {
        CalculateTouchInput( ActiveTouch );
        ActiveTouch.Phase = TouchPhase.Canceled;
      }
    }
  }
  else
  {
    if (Input.touches.Length > 0)
    {
      DeviceTouch = Input.GetTouch(0);
      if (ActiveTouch.Phase == TouchPhase.Canceled)
      {
        ActiveTouch.Phase = DeviceTouch.phase;
        ActiveTouch.StartTime = System.DateTime.Now;

        ActiveTouch.StartTouchLocation =
        DeviceTouch.position;
        ActiveTouch.CurrentTouchLocation =
        DeviceTouch.position;
      }
      else
      {
        ActiveTouch.CurrentTouchLocation =
        DeviceTouch.position;
      }
    }
    else
    {
      if (ActiveTouch.Phase != TouchPhase.Canceled)
      {
        CalculateTouchInput( ActiveTouch );
        ActiveTouch.Phase = TouchPhase.Canceled;
      }
    }
  }
```

```
  }
}
```

This is the input class. This class is made to be simple because it will cater to our game specifically. Some games will require managing multiple inputs, input recognition for different types of swipe patters, and input that would register to in the world (such as tapping a box in the game world to have it explode).

# Summary

As every game has its own form of rules, it often results in a unique type of input interface for the player to send input to. In our case, you learned that for a mobile, simple is better, which is why we will only use a system that gets swipes and taps.

In more complex games, such as those made for PC or consoles, the input can become a much bigger project because there are more options given to the player via a keyboard or controller. The one thing that will stay the same regardless of the game you are creating is that the input system has to cater to the game being made. If you try to make an overly complex input system for a simple game or a simple input system for a complex game, the player will often find trouble when trying to play.

Unity also offers another type of input called Gyroscope. It is a method of taking the tilt of the device and translating it into movement. Popular top-down games, such as those who have a ball, often use this type of input to simulate the surface of a tabletop, so as you rotate the device around, the orientation of the table will shift, allowing the ball to roll in the direction of the slope a tilted table would represent.

Another example of Gyroscope is used in racing games. You can tilt the device (such as a steering wheel) and the vehicle will turn in the direction you will turn the device.

In the next chapter, we will create the character class. This will receive the input from the player input class to move on the screen. You will be able to collide and touch Game Objects in the world.

# Chapter 3. Player Character, Obstacles, and Pickups

The player character is the visual representation of what the user will input to the device. In the game we're creating, it will be quite simple. Our character will be designed to perform only three things: change its direction based on the swipe input of the user, collide with pickups in the world, and collide with obstacles in the world.

The swipe direction will determine the direction the character will move and with how much force it will move. The collision with pickups will add to a collection of points for the user. Also, the collision with obstacles will "kill" the character and end the game.

# What is a character?

A character is a GameObject that has `Rigidbody` attached to it. `Rigidbody` is a component in Unity that will give the character simple physics and things such as gravity and collision. This means that `Rigidbody` also allows the character to collide with the world and with other GameObjects collision “touch” boxes.

# What is a pickup?

A pickup is a GameObject that has a box collider attached to it. When the box collider is touched by `Rigidbody`, we will run code to check what touched it. If what touched it is the player character, we will give the player character a point of what type of pickup it is and then destroy the pickup so that it cannot be touched again. Also, if the pickup is a coin, we will tell the player character that it collected a coin by taking the value of coins the player currently has and add *one* to it.

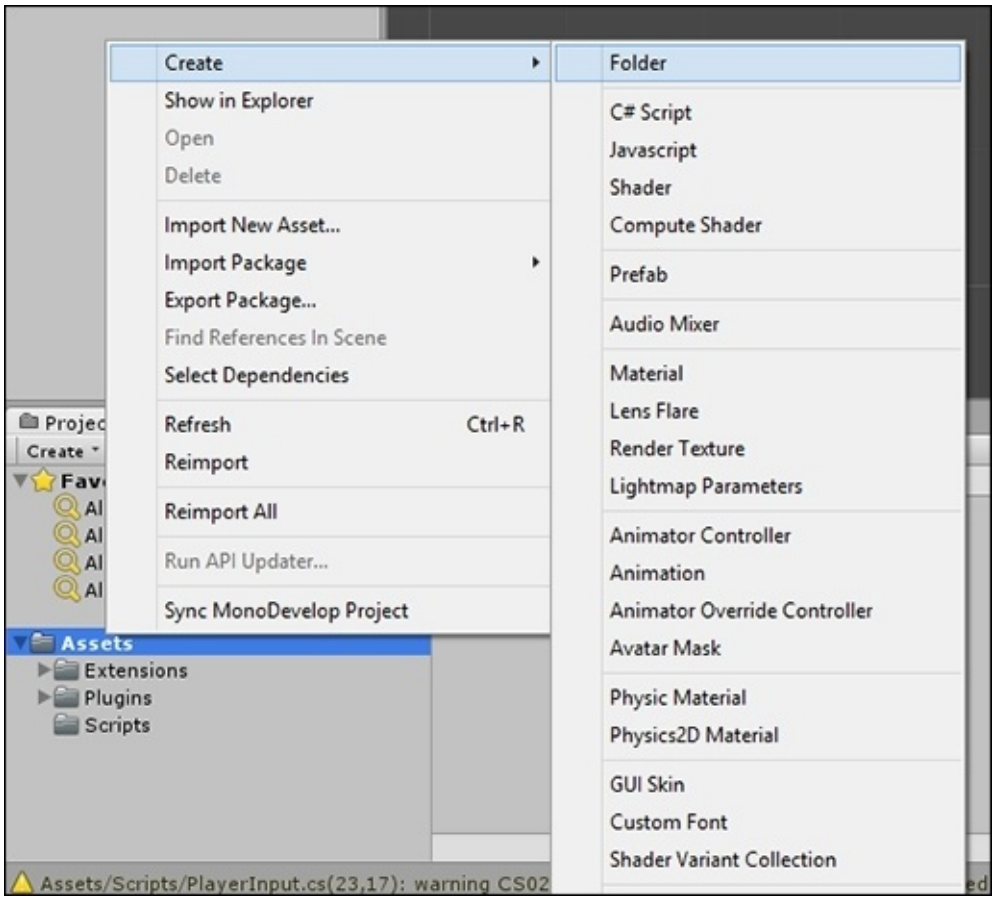
# What is an obstacle?

Similar to the pickup, an obstacle is a GameObject that has a box collider attached to it. The thing that makes an obstacle different from a pickup is that the obstacle is designed to destroy the player character and end the game if it’s collided with. This means that the obstacle is more crucial to the overall game loop than the pickup is.

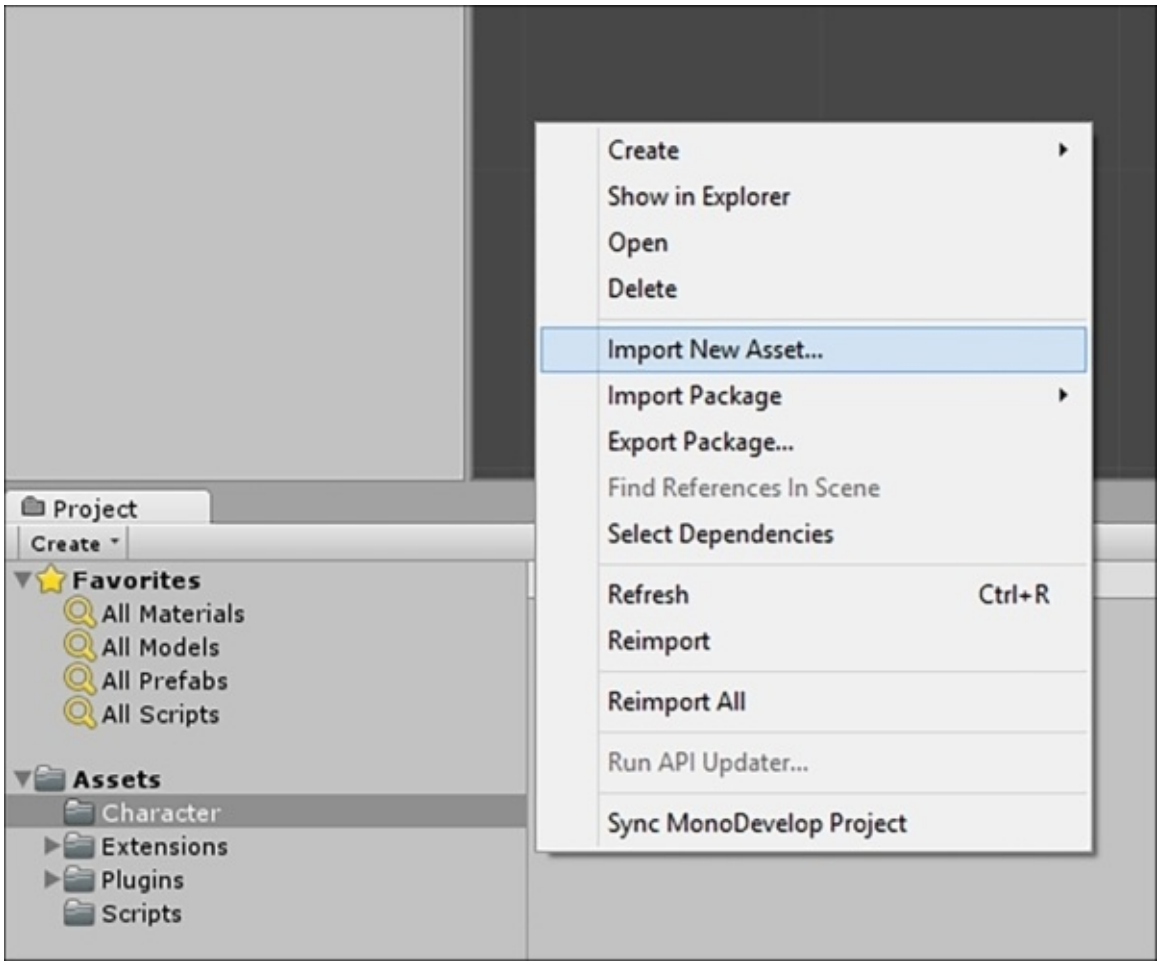
# Creating the character

The character will be our representation of what the player is controlling. You can think of it as the “hero” or positive visual character, something the player can identify with their input actions.

1. To begin with, open Unity and create a new folder in the **Project** window in the **Assets** folder**:**




2. Name it `Character`.
3. In this folder, right-click and select **Import New Asset…**:



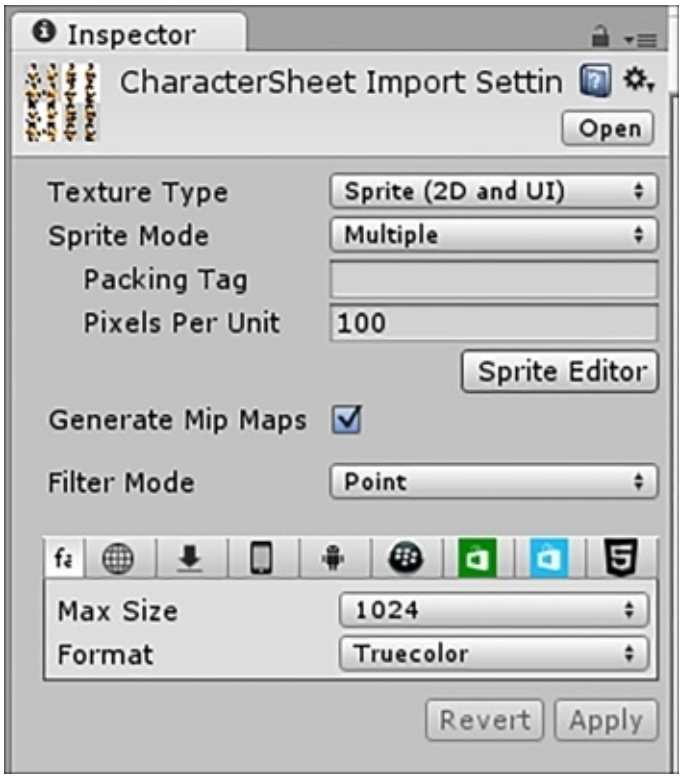

4. Navigate to the `Art` folder for this book. You will find a subfolder called `CharacterThree_Character`.
5. Open it. In this folder, there is a file named `Character_Sheet.png`.
6. Select this file and then click on `Open` to import the asset into Unity.

This will import the default running sprite sheet that will be used for our character.

## Tip

[http://opengameart.org/](http://opengameart.org/) is a website designed to offer 2D artwork under the creative commons license. If you want to find the art that you can use, manipulate, and distribute freely, this site can be very useful.

1. Once you have `Character_Sheet.png` in the `Assets` folder, select it. Then, in the **Inspector** tab, you will see the settings for the PNG file.
2. Select **Sprite Mode** as **Multiple**.
3. Select **Filter Mode** as **Point**.
4. Select **Max Size** as **1024**.
5. Select **Format** as **Truecolor**:




6. Then, click on **Apply**.

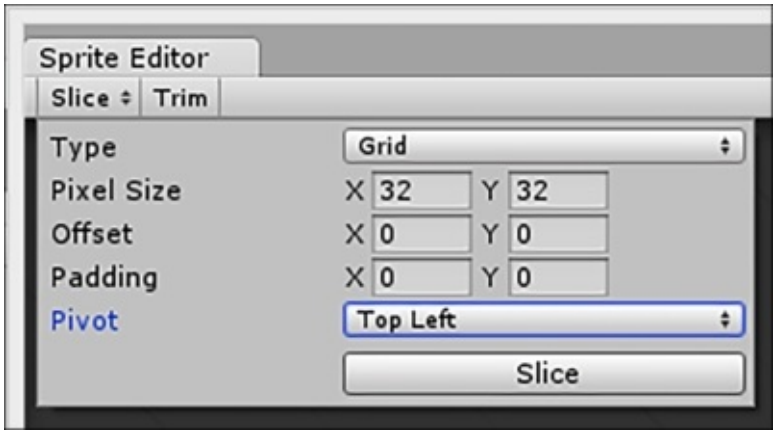
# Creating the character controller

These settings allow the sprite sheet to be animated, have the same detail as the normal image, and keep the max file size down by limiting the max texture sheet size to 1024 pixels.

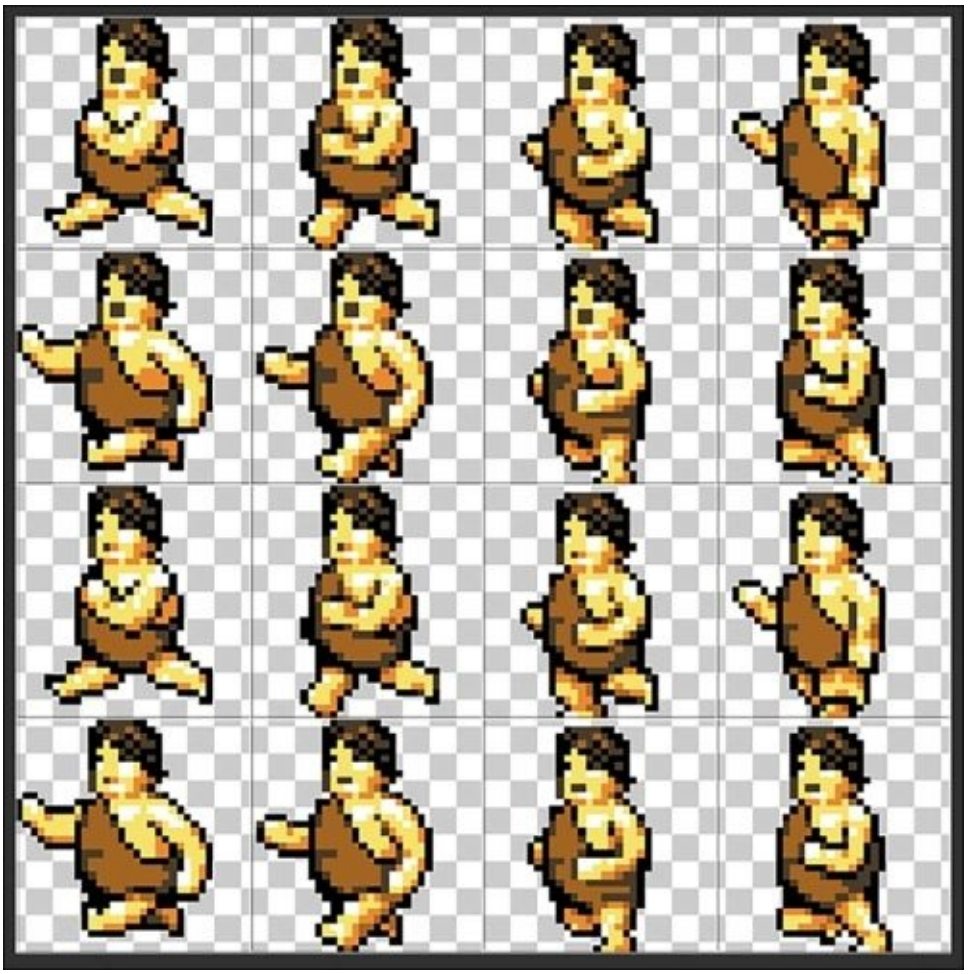
Next, we need to update the sprite sheet so that it knows the boundaries for each of the animation frames in the sheet. To perform this, in **Inspector**, click on **Sprite Editor**.

This is designed to help us break the image based on the **X** and **Y** values we want to. For example, if each frame of the animation is `32` pixels wide on **X** and `32` pixels wide on **Y**, we can use the **Sprite Editor** to cut the image into smaller images based on that size.

1. In **Sprite Editor**, click on **Slice** in the top-left corner of the window.
2. Select **Type** as **Grid**.
3. Select **Pixel Size** on **X** as `32` and **Y** as `32`.
4. Leave **Offset** and **Padding** as `0`.
5. Change the **Pivot** to **Top Left**.
6. Then, click on **Slice**:




This should result in the image being sliced into 32 x 32 boxes across the whole image. Take a look at the following image:

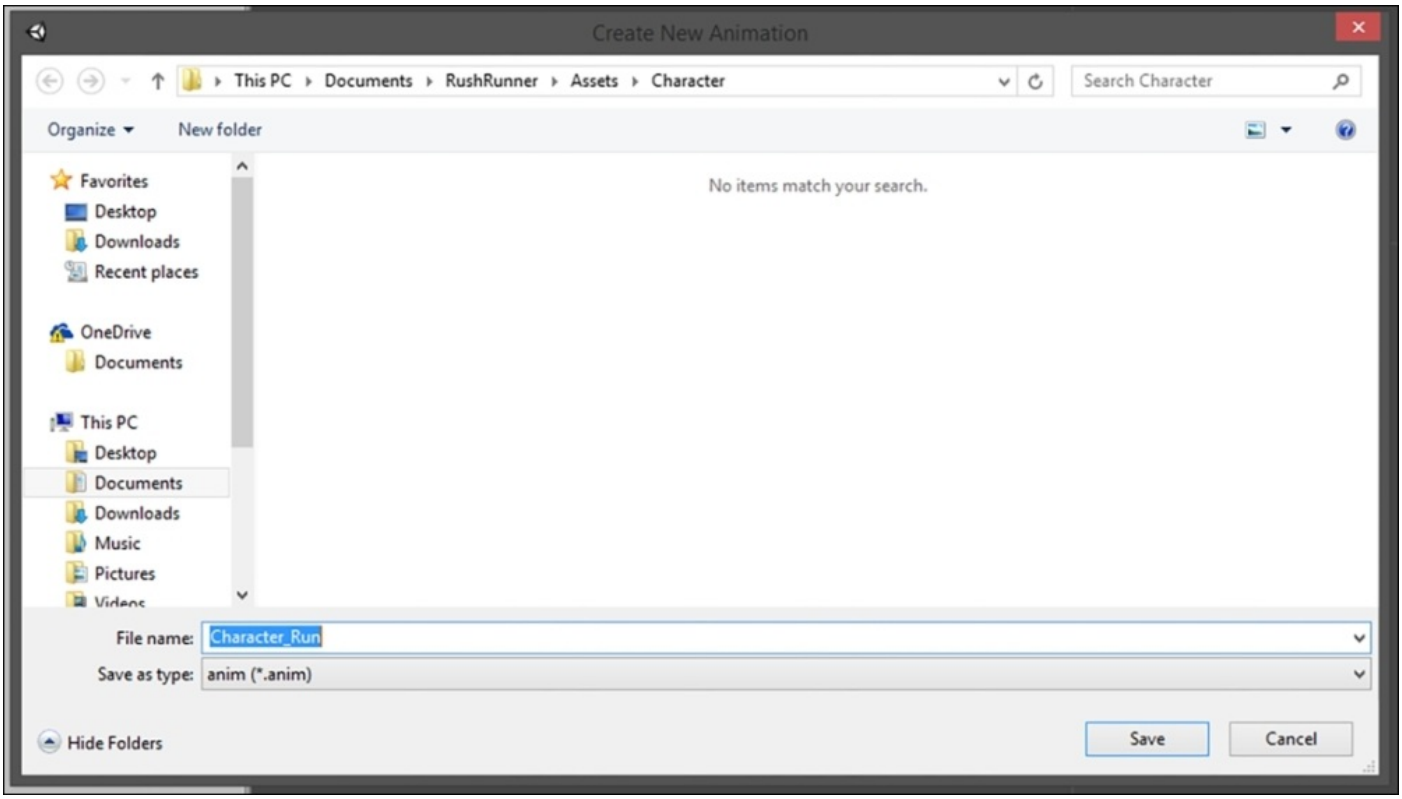


The major issue with this image is that it has the character facing in the wrong direction because the character will run from left to right in the game and not from right to left, as shown in the preceding image. We will fix this later by flipping the **X** axis of the character to -1, which will have the character face the correct orientation.

In the `Assets` folder, you can see that `CharacterSheet` went from a single image to now a parent object with 16 children, ranging from `CharacterSheet_0` to `CharacterSheet_15`.

In order to make this `CharacterSheet` work in game, we need to create an animation for it. To perform this, select the `CharacterSheet` parent object in the `Assets` folder and left-click and drag it onto the **Scene** window.

You should see a popup asking you to save the anim file in the `Assets/Character` folder. Name it `Character_Run` and click on **Save**. Remember to drag it onto Scene and *not* on Hierarchy. If you drag it onto Hierarchy, the animation window will *not* appear.




After you click on **Save**, a couple of things will happen. The first and most noticeable change is that the GameObject named `CharacterSheet_0` will be available in your game world. The second and third things are additions to files in your `Assets/Character` folder.

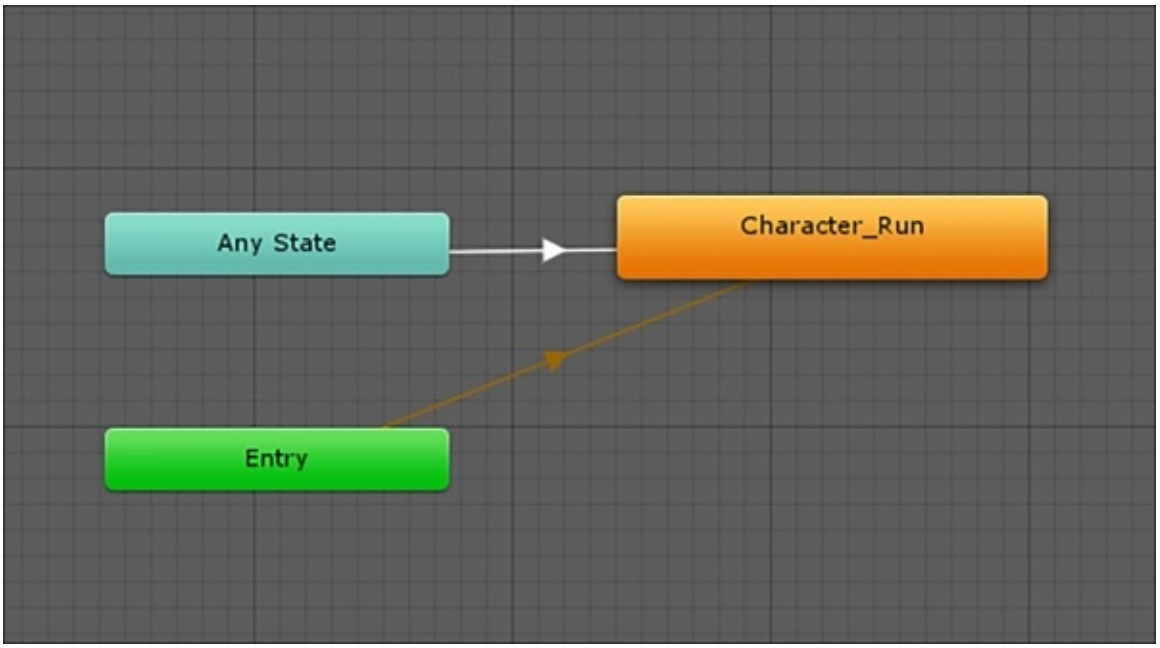
The first file added will be the `.anim` file called `Character_Run`. This is the **animation** file that is in charge to cycle the animation of the character. The second is a **controller** file.

The **controller** file is used to set how the character will blend its animations. Double-click on the `ControllerSheet_0` controller file to open it. You will see three nodes: the **Any State** node, the **Entry** node, and the **Character_Run** node. The **Any State** node is the node that essentially starts the controller, meaning that no matter what goes on in the controller file, this is what will run as long as it exists.

In order to start your controller, right-click on the **Any State** node and select **Make Transition**. You'll see that your mouse will not be connected to a white line with an arrow in the middle.

Move the end of the white line and click on the **Character_Run** node. This connects the **Any State** node to the **Character_Run** node, making it the default node that will play if this controller is being used. This will be the case for our character.

To be sure, select **File** and then click on **Save Project**.




Next, we need to complete the controller by adding a jumping portion to it. To perform this, we need to take the same steps as the **Character_Run** node:

In the `Assets/Character` folder, right-click on and select **Import New Asset…**.

In the same `ChapterThree_Character` folder, select the `CharacterJumpSheet.png` file and import it.

1. With the new import file selected, go to Inspector and set **Sprite Mode** as **Multiple**.
2. Select **Filter Mode** as **Point**.
3. Select **Max Size** as **512**.
4. Select **Format** as **Truecolor**.
5. Then, click on **Apply**.
6. We now need to slice **CharacterJumpSheet.png** into the same 32 x 32 subimages.
7. Open the **Sprite Editor** by having **CharacterJumpSheet.png** selected and click on **Sprite Editor**.
8. Select **Slice** in the top-right corner of the **Sprite Editor** window.
9. Select **Type** as **Grid**.
10. Select **Pixel Size** for **X** as `32`.
11. Select **Pixel Size** for **Y** as `32`.
12. Select **Pivot** as **Top Left**.
13. Then, click on **Slice** and then on **Apply**.

You should now have a similar file as what `CharacterSheet` shows and a parent object name (`CharacterJumpSheet`) with *two* children named `CharacterJumpSheet_0` and `CharacterJumpSheet_1`.

In order to have an animation that we can use for this file and `CharacterSheet`, perform the same as what we did earlier for `CharacterSheet`: click on `CharacterJumpSheet` and drag it onto the **Scene** window.

Name the anim file `Character_Jump`.

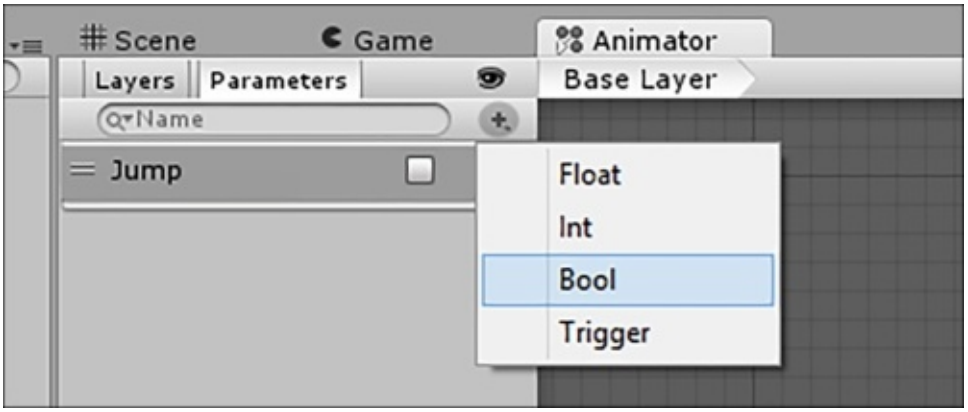
One thing that needs mentioning is that by default, Unity will create another controller file for this animation. We do not need this.

Click on `CharacterJumpSheet_0.controller` and then *Delete* on your keyboard. Alternatively, you can right-click on `CharacterJumpSheet_0` and select **Delete** from the options.

We now need to connect the new `Character_Jump.anim` file to the `ControllerSheet_0.controller` file. To start with, open the `ControllerSheet_0.controller` file. Now, you can again see the animation state controller in the **Animator** window.

Before adding the `Controller_Jump.anim` file, we first need to create a **parameter** for the controller file to use in order to blend between the two animations we have. In the top-left corner of the **Animator** window, there are two tabs: **Layers** and **Parameters**. Click on **Parameters**.

Under the selection for **Parameters**, there is a small + icon. Click on it and select **Bool**. Name this bool `Jump`:




The parameter system in the **Animator** window is a utility that allows you to blend between specific animations during the period for which the game runs. In our case, we need to let our controller know when the player jumps. When this happens, we can change the bool value in the controller to true and the animations will blend from running to jumping.

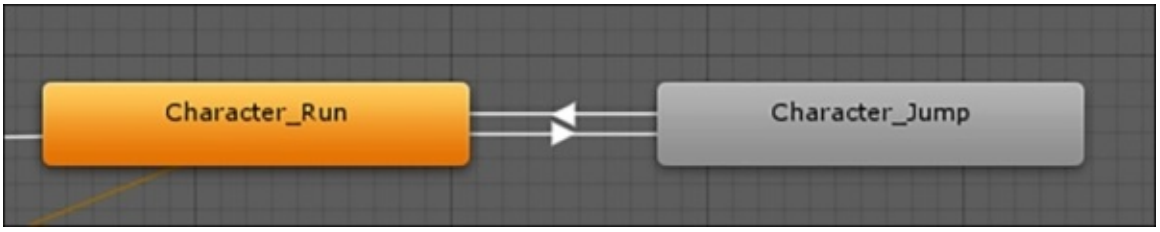
Next, let's add the `Character_Jump.anim` file to the controller. To perform this, select the anim file and then click and drag it onto the **Animator** window. You'll see that a new animation state box called **Character_Jump** will appear in the **Animator** window.

We now need to connect the **Character_Run** state to the **Character_Jump** state. As before, when we connected states, right-click on the **Character_Run** state box and select **Make Transition**. Then, with the white line attached to the mouse cursor, click on **Character_Jump**.

We also need to connect **Character_Jump** to the **Character_Run** state box. To perform

this, follow the same steps of selecting the animation state box—this time **Character_Jump**—and select **Make Transition**. With the white line attached to the mouse cursor, click on **Character_Run**.

This will now make it such that both the animation states are connected to each other and allows you to blend between each of them:




For the controller to know when to switch between these two animation states, we have to tell the transition lines what parameter to look out for. Perform the following steps:

1. Click on the transition line from **Character_Run** to **Character_Jump**.
2. In **Inspector**, at the bottom, look for the **Conditions** section. In this section, there is a gray + button.
3. Click on the gray + button.

As we only have *one* parameter, this will automatically fill with **Jump** and set it to **true**.

Next, find the section named **Has Exit Time** and uncheck this selection box. If checked, this setting allows the animation to complete before it transitions into the new state. As we want the **Jump** animation to play as soon as the input has been received from the player, we will uncheck it.

We now need to do the same thing in reverse to connect **Character_Jump** to the **Character_Run** transition.

1. Select the transition line that faces from **Character_Jump** to **Character_Run**.
2. Now, click on the gray + button in **Conditions**.
3. Change **true** to **false**.
4. Then, uncheck the **Has Exit Time** box.
5. To save, click on **File** and select **Save Project**.

# Creating the Character GameObject

Now that we have the major components of the appearance of our character, we need to create it in the game world.

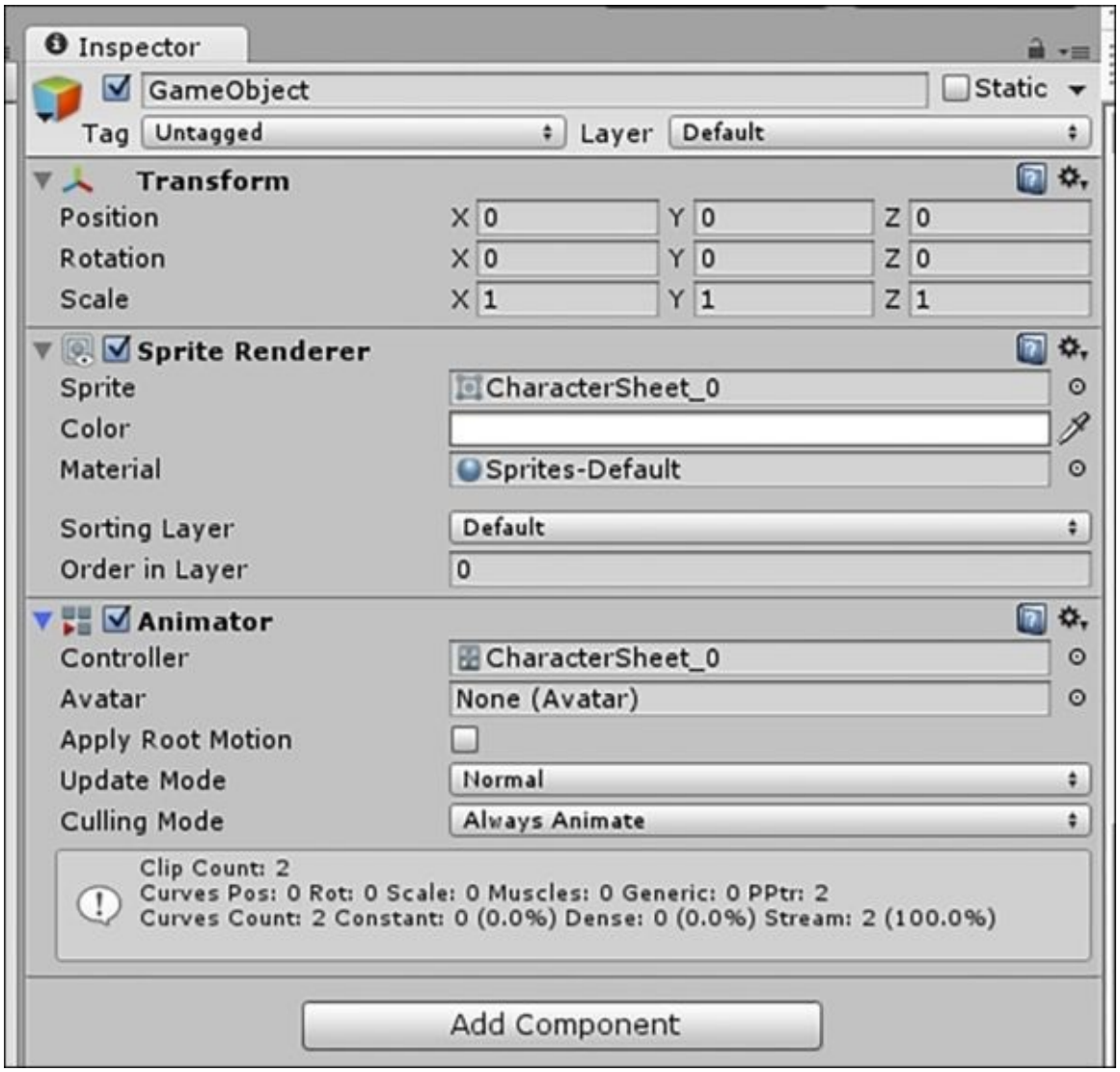
In the **Hierarchy** tab, on the left-hand side of the **Editor** window, right-click and select **Create New**.

This will add an empty GameObject to your scene.

## Tip

If you still have the **Animator** window open, click on the **Scene** tab at the top of the **Editor** window.

1. Make sure that the new empty GameObject is selected. Click on **Add Component** at the right-hand side of the **Editor** window in the **Inspector** tab.
2. Search for **Sprite Renderer** and click on **Sprite Renderer** from the list.
3. In the `Assets/Character` folder, select `CharacterSheet` and click and drag it onto the **Sprite** section of **Sprite Renderer**.
4. You'll see that the sprite gets named as the first child element of the `CharacterSheet` object: `CharacterSheet_0`. This is correct and fine.
5. Next, we need to add an **Animator** component to GameObject.
6. In the **Inspector** window, click on **Add Component**.
7. Search for **Animator** and then click on **Animator** to add it:




8. In the `Assets/Character` folder, click on the `CharacterSheet_0.controller` file and drag it into the **Controller** section of **Animator**.
9. Next, let's name GameObject to something we can relate to. I will use `Character`. You can name it anything you'd like, although organization is important in games, so try to choose something that is easy to recognize.
10. You'll see that the character is facing the wrong direction.
11. In the **Inspector** tab under **Transform**, change **Scale** on the **X** axis from `1` to `-1`. This will flip the character to face the correct way for the game.

## Tip

If you play the game now, you'll see the character running.

# Adding Rigidbody 2D

The **Rigidbody 2D** component is a component that simulates the basic physics for the GameObject it is attached to. In our case, it will simulate the gravity for our character, allowing it to appear as if it is running into other GameObjects, jumping and falling.

With the `Character` GameObject selected, click on **Add Component** in the **Inspector** tab.

Search for **Rigidbody 2D** and click on **Rigidbody 2D** from the list to add it.

In the **Inspector** tab, with the **Character** GameObject selected, click on the **Fixed Angle** checkbox inside **Rigidbody 2D**. This will force the rigid body component to stay fixed to an angle and avoid spinning, resulting in the character doing flips.

This is it for Rigidbody 2D. It is a cool component that does a huge amount of work for us with little effort.

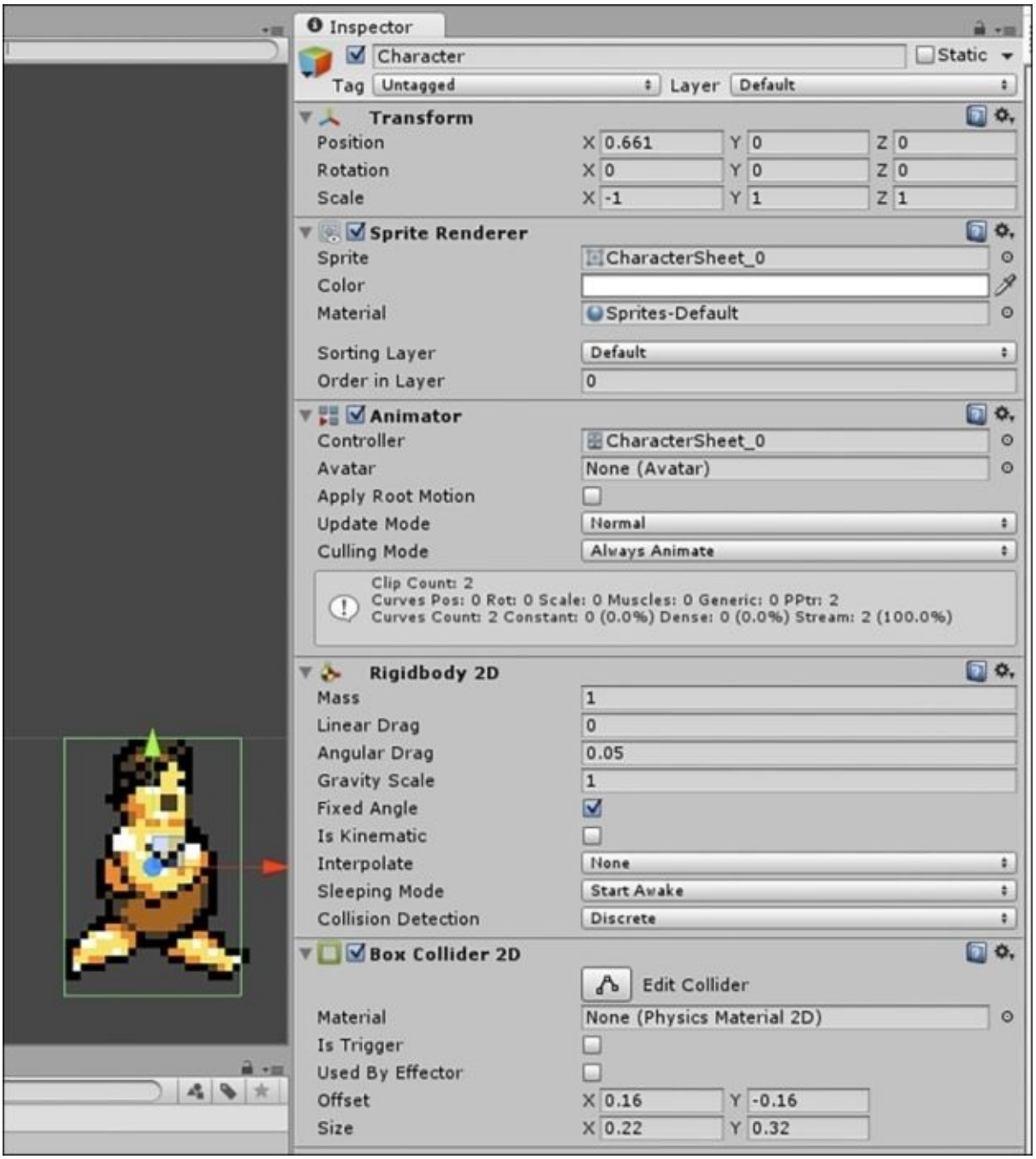
# Adding Box Collider 2D

The **Box Collider 2D** component will give our character a bounding box that will interact with the world around it. This means that if the character hits another GameObject that also has **Collider**, we will be able to use code to know about it. This is imperative to the interaction of the character with the game world since without it, we wouldn't have the information needed to do things, such as collect pick-ups, or know when the character hit something dangerous, such as the obstacles.

Just as Rigidbody 2D, the Box Collider 2D component is easy to set up.

1. With the **Character** GameObject selected, navigate to the **Inspector** tab and click on **Add Component**.
2. Search for **Box Collider 2D** and select **Box Collider 2D**.

By default, **Box Collider 2D** will fit itself around the image. As our image is 32 x 32 pixels wide and high by default, the collider will match it as it is also 32 x 32 pixels wide and high. To make it fit our character better, change the **X** value of **Size** down to `0.22`:




This area is the basis of our character. It will simulate gravity, collide with anything else that is representative of a collider, and take in any sort of force we want to give it, such as changing its velocity when the player swipes on the mobile device.

In order to test our changes, we will need to create a temporary floor that will prevent the character from falling forever.

1. In **Hierarchy**, right-click and select **Create Empty**.
2. Select the new empty GameObject and in **Inspector**, click on **Add Component**.
3. Search for **Box Collider 2D** and select **Box Collider 2D**.
4. With GameObject still selected, change the **X** value of **Size** as `50`.

We will also need to position the character in the world correctly. To perform this, start by

selecting the **Character** GameObject.

1. In **Inspector**, change its **Position** in the **Transform** component to `0` on **X**, `0` on **Y**, and `0` on **Z**. This will put the character in the very center of our game world.
2. Next, select the empty GameObject that is representative of the temporary floor.
3. In its **Transform** component, select the **Position** as `0` on **X**, `-1` on **Y**, and `0` on **Z**. This will put a collider just after **Character**, preventing it from falling.
4. To have a better view of **Character**, select the **Main Camera** GameObject in the **Hierarchy** tab. In **Inspector**, select the **Size** as `1`. If **Camera Projectile** isn't set to **Orthographic**, make sure it is set to this.

If you play the game now, you will see the character fall a little and then stop and run in place. If you don't see this, make sure to revisit the preceding steps.

- If the character isn't falling, make sure that you have Rigidbody 2D on it.
- If the character isn't playing our running animation or is playing the wrong animation, make sure to check whether the controller is correctly assigned and set up.
- If the character doesn't have a sprite, view **Sprite Renderer** to make sure that a sprite is assigned.
- If the character is falling forever, make sure to navigate to the temporary floor we made and Box Collider 2D to verify that it is present and sized correctly. If the floor is set correctly, make sure **Character** has a Box Collider 2D component on it.

# Writing the character code and connecting the PlayerInput class to it

The last stage of the character is its code class. The code class will receive input from the PlayerInput class we wrote earlier, so the character can be controlled through the input and filter out the type of GameObjects that collided with from the parent of that collider type.

Perform the following steps:

To start with, navigate to the `Assets/Scripts` folder and right-click on it. Select **Create** and then **C# Script**. Name it `Character`.

Next, we need to give our `PlayerInput` class a reference to our character. To perform this, navigate to the `Assets/Scripts` folder and double-click on the `PlayerInput.cs` file to open it.

At the top of the class, under the `SimpleTouch` struct, add the following code:

```
public Character GameCharacter;
```

We also need to tie the function that will call `GameCharacter` when `PlayerInput` gets a new one. At the bottom of the `CalculateTouchInput` function, add the following code:

```
if (GameCharacter != null)
{
  if (!GameCharacter.isDead)
  {
    GameCharacter.RecieveInput(touchDistance, touchDirection);
  }
}
```

# Character attributes

At the top of the `Character` class, we need to add the bool value that will be used to know whether a character is jumping or not and if a character is dead or alive; we need an int value to know how many coins a character has and if a character should be fading out. The `isJumping` bool will be used in the code that we will write to know when the character jumps from the `SimpleTouch` type of `PlayerInput`. Finally, we will store references to `GameCharacterSprite` and `CharacterRigidBody` so that we can use them through the use of this class. Add the following code before the `Start` function:

```
[System.NonSerialized]
public bool isJumping;

[System.NonSerialized]
public bool isDead;

[System.NonSerialized]
public int CoinCount;

[System.NonSerialized]
public Vector3 RestartLocation;

private bool isFadeOut;

SpriteRenderer GameCharacterSprite;

Rigidbody2D CharacterRigidBody;

Animator CharacterAnimator;
```

## Tip

`[Sytem.NonSerialized]` is a way to declare a public variable and have Unity ignore it when it populates the exposed component variables. It means that `isJumping` won't show up in the component list of variables in the editor. It will also prevent this variable from being serialized. Serialization is the process of converting an object to a stream of bytes so that it can be transmitted into memory.

To explain more, let's understand the following terms:

- `isJumping`: This is the bool value to let both the class and the animator know if the character is jumping. *True* is jumping, whereas *false* is not jumping.
- `isDead`: This is a bool value that does exactly what the `isJumping` bool does only to check whether the player is dead or not. Again, *true* is dead, whereas *false* is not dead.
- `CoinCount`: This specifies how many coins the player has collected. Later on, we will store this value so it doesn't reset every time the player restarts the game.
- `RestartLocation`: This denotes the `Vector3` location that we can use to reset the character to after they have died.
- `IsFadeOut`: This is a bool that transitions the character from an opaque state to a translucent state. *True* is fading to translucent, whereas *false* is fading to opaque.

- `GameCharacterSprite`, `CharacterRigidBody`, and `CharacterAnimator`: These are references that we can keep, so we don't have to keep getting them. `GameCharacterSprite` is the sprite sheet, `CharacterRigidBody` is Rigidbody 2D for the character, and `CharacterAnimator` is the animation tree for the character.

To set some of these values, we can use the `Start` function. Add the following code in the `Start` function:

```
// Use this for initialization
void Start ()
{
  RestartLocation = gameObject.transform.position;
  GameCharacterSprite = gameObject.GetComponent<SpriteRenderer>( );
  CharacterRigidBody = gameObject.GetComponent<Rigidbody2D>( );
  CharacterAnimator = gameObject.GetComponent<Animator>( );
}
```

This allows you to use all of these references in the other code for this class without having to use the `GetComponent` function again and again.

# PlayerInput

Now, we know that we have to have a couple things for character to be functional. Character needs to take the `SimpleInput` struct data from `PlayerInput` so that it knows how to move, and the character needs a function to know when it has collided with another GameObject that has a collider.

Under the `Update` function in the `Character` class, add the following functions:

```
    // Get input data from PlayerInput
    public void RecieveInput(float Distance, Vector2 Direction)
    {
      if (CharacterRigidBody != null && CharacterAnimator != null )
      {
        Direction.x = 0.0f;
        if (Direction.y > 0.0f && !isJumping)
        {
          CharacterRigidBody.AddForce(Direction * Mathf.Clamp(Distance, 0,
200));

          CharacterAnimator.SetBool("Jump", true);
          isJumping = true;
        }
        else if ( Direction.y < 0.0f )
        {
          CharacterRigidBody.AddForce(Direction * Mathf.Clamp(Distance, 0,
250));
        }
      }
    }

    // When the character collides with another GameObject
    void OnCollisionEnter2D(Collision2D Col)
    {
      if (CharacterAnimator != null)
      {
        CharacterAnimator.SetBool("Jump", false);
        isJumping = false;
      }
    }
```

The `RecieveInput` function takes the parameters, the distance of the player's swipe, and the direction of the player's swipe. We need the distance in order to know how "hard" the player swiped on the screen, and we need the direction to know whether the player swiped up or down.

When we receive this data from `PlayerInput`, we then need to get ready to use it. To begin, we need a reference to the **Character** GameObject's Rigidbody 2D component. To have this, we will use `gameObject.GetComponent<Rigidbody2D>( )` and store it in a local variable named `characterRigidBody`.

We will also need a reference to the animator on character that holds the controller. The controller holds the `Jump` variable, which we will set when the character jumps.

We then check to make sure that `characterRigidBody` is something by checking that it is *not* null; null meaning empty. We also perform the same for `characterAnimator`.

If we have a reference in `characterRigidBody` to GameObject's Rigidbody 2D, we start by making sure that the **X** axis in **Swipe** is set to `0`. This means that the direction will only be up and down, so the player can't jump forward or backward.

We then check whether the **Y** axis on the **Direction** parameter is more than `0.0`, meaning the player swiped up if the character is not already jumping. If this is `true`, we will add a force to the `Rigidbody2D` character.

`AddForce` takes two arguments: the direction the force needs to be and how much force to apply to Rigidbody 2D. For us, the direction has already been given and altered. The force amount is calculated using `Mathf.Clamp( )`.

`Mathf.Clamp( )` is a way to make sure that the force value is within bounds. If we let the distance decide how much force is added to the character always, the character would go shooting off into space if the player swiped the entire screen of their device. So, we will use the clamp function to check whether the swipe distance is less than 200 and use this value. If the swipe distance is more than 200, clamp it down to 200.

After `Rigidbody2D` character has a force, we want to update the controller so that it knows to switch to the jumping animation. We do this by setting the `Jump` bool value to true with the stored reference set earlier in the function and check to make sure that it exists.

As the character needs to know whether it is jumping, we will set `isJumping` to `true` here as well. This way if the player tries to swipe while character is already jumping, the condition we have will fail. Also, there will be no force added in the positive **Y** axis. If we didn't have this check, the player would be able to swipe continuously. Also, the character would continue to jump higher and higher.

If the first condition that checks whether the direction of the swipe is in positive Y and if character is *not* jumping fails, we check whether the direction of the swipe is in negative Y. The reason we don't check `isJumping` here is because it doesn't matter if we swipe down if the player is or isn't currently jumping. If Character is already on the ground, it can't go farther down, and if the character is jumping, a force downwards will make it fall to the ground faster, which is intended in design.

The `OnCollisionEnter2D` function is a Unity function designed to manage what happens when colliders collide.

`OnCollisionEnter2D` takes a single parameter: the collider that was collided with. For the moment, the only thing we know is that the character can collide with the temporary floor we have on the scene. As a result, we can assume that the only thing to do here is to stop the player from jumping. We start by having the reference back to the character's **Animator** component and check whether it exists. If it does, we will set the `Jump` bool value to `false`; to set that the player is no longer jumping, we will also set the `isJumping` class variable to `false`, so if the player was to swipe upwards again, the `AddForce` function gets a call. Also, the character will jump.

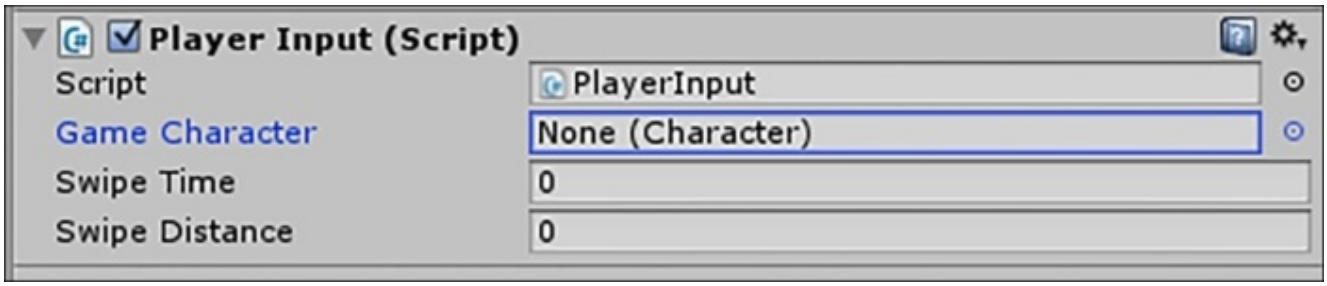
We will compare the type of GameObject with the one that is being collided with so that we can decide what to do with the colliding GameObject as well as what to do with the **Character** GameObject.

If you play the game now, you should see the character transition from jumping and not jumping; this is based on the player input of swiping upwards to jump and the character stopping the jump by hitting the temporary floor

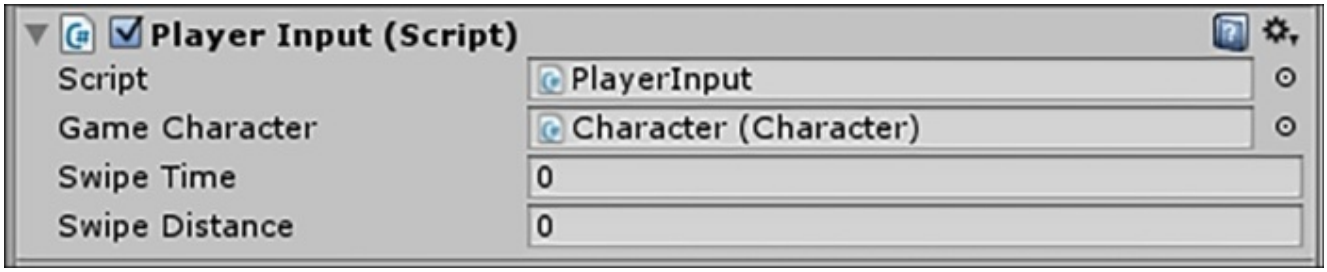
In addition, we will check to make sure that `GameCharacter` isn't null (it exists); we also make sure that it is not dead and then we will call `GameCharacter.RecieveInput( )`, which will send `touchDistance` and `touchDirection` to the `Character` class, which allows you to control the character based on them.

Perform the following steps:

1. Save this `PlayerInput` file and navigate back to Unity, which should compile the code now.
2. After the code is compiled, right-click on **Hierarchy** and select **Create Empty**.
3. Name this new GameObject as `Player Input`.
4. With the **PlayerInput** GameObject chosen, select **Add Componen**t from the **Inspector** window.
5. Search for **PlayerInput** and select **PlayerInput**:




6. Next, select the **Character** GameObject.
7. In the **Inspector** window, select **Add Component.** Then, search for **Character** and select **Character**.
8. Now, select the **PlayerInput** GameObject and then left-click on the **Character** GameObject and drag the object onto the **Character** field of the **PlayerInput** component:



## Tip

If you are like me and have **Swipe Time** and **Swipe Distance** set to `0`, the settings from the `PlayerInput` chapter are **Swipe Time** = `0.08` and **SwipeDistance** = `25`.

This will give the **PlayerInput** component of the **PlayerInput** GameObject a reference to the **Character** GameObject. This also allows you to use **PlayerInput** to “communicate” with **Character** or to be more direct, send the `SimpleTouch` struct data of the current touch input from the user to a function of **Character** so that **Character** knows what it should do.

Next double-click on the `Character.cs` class to open it. Like the `PlayerInput` class, you will see the generated code from Unity; this includes the `Start` and `Update` functions and the two `Using` declarations as well.

# The KillCharacter function

Now, we need to write a function to kill the character. This will be used when the character collides with an obstacle.

Under the `OnCollisionEnter2D` function, add the following code:

```
    // Kills the character
    public void KillCharacter()
    {
      if (!isDead)
      {
        if (CharacterRigidBody != null)
        {
          CharacterRigidBody.AddForce(new Vector2(Random.Range(-1, 1), 1) *
512);
          isDead = true;
          isFadeOut = true;
        }
      }
    }
```

This function is designed to make sure that the character's Rigidbody 2D component exists—by making sure it is *not* null—add a force to it in a random direction between the negative *one* and the positive *one* on the **X** axis and the positive *one* on the **Y** axis. This means that a force will be added, which will throw the character upwards with the force of the random direct multiplied by 512. We can also set the character as dead, so it won't be able to pick up anything.

To avoid having the character go floating off into space, let's use the `isFadeOut` variable to fade the character if it is true. This way the player will see the character fade to transparent before it escapes the screen.

In the `Update` function of the character C# class, add the following code:

```
  // Update is called once per frame
  void Update ()
  {
    if (isFadeOut)
    {
      if (GameCharacterSprite != null)
      {
        float currentAlpha = GameCharacterSprite.color.a;
        currentAlpha -= 0.0085f;

        if (currentAlpha < 0.01f)
        {
          currentAlpha = 0.0f;
        }

        Color newColorAlpha = new Color(1.0f, 1.0f, 1.0f, currentAlpha);
        GameCharacterSprite.color = newColorAlpha;
      }
    }
```

```
}
```

Let's start by checking whether the `isFadeOut` bool value is `true`. If so, we can continue to set the alpha of `SpriteRenderer`. To set the alpha or fade out the **Character** sprite, we will need a reference to its `SpriteRenderer` component. We can have this by using the `gameObject.GetComponent` function. When we have this, we can make sure it exists by verifying that it is *not* null. We will then set a float variable to the current value of the `SpriteRenderer` reference and then subtract a small amount from this.

We will then check `currentAlpha` of the sprite to check whether it is less than `0.01`. If it is, we set `currentAlpha` to `0`. Also, we will create a new `Color` variable and set the alpha channel to the `currentAlpha` value.

Finally, we will set the reference to the `SpriteRenderer.color` value to the updated alpha value.

Then, we will need a function that will reset the character back to running. This will be used later on when we create a reset button in the UI. Write the following function under `KillCharacter`:

```
// Revives the character
public void ReviveCharacter()
{
  Color resetColorAlpha = new Color( 1.0f, 1.0f, 1.0f, 1.0f );

  gameObject.transform.position = RestartLocation;
  isDead = false;
  isFadeOut = false;

  if (GameCharacterSprite != null)
  {
    GameCharacterSprite.color = resetColorAlpha;
  }
}
```

To revive the character is quite easy. We can create a new world location at the `0X`, `0Y`, and `0Z` location (the center of the world) and set the **Character** GameObject to that location.

We also need to reset the `SpriteRenderer.color.a` (alpha) value back to default. To do this, we need to get a reference of `SpriteRenderer` and make sure that it exists by making sure that it is *not* null.

If it exists, we set the color value of `SpriteRenderer` to the `resetColorAlpha` value, which is `1.0f` red, `1.0f` green, `1.0f` blue, and `1.0f` alpha by default.

We will then set `isDead` to `false`, meaning that the character is alive, and set `isFadeOut` to `false` so that the character does not fade out anymore.

# Creating the coin pickup

The coin pickup is a representation of the good type of collider the character can interact with. It will be what the player collects in the game that they can use for currency. Perform the following steps:

1. To start with, navigate to the `Assets` folder, right-click on it, and select **Create Folder**. Name this folder `Pickup`.
2. Open the `Pickup` folder, right-click on it, and select **Import New Asset…**. Navigate to the `Art` files for this book, and open the `ChapterThree_Pickup` folder, and select the `Pickup_Coin.png` file to import.
3. With the `Pickup_Coin.png` file imported, select it. Then, in the **Inspector** window, select **Sprite Mode** as **Multiple**.
4. Then, select **Filter Mode** as **Point**.
5. Now, select **Max Size** as `256`.
6. Then, select **Format** as **Truecolor**.
7. Finally, click on **Apply**.
8. Now, click on the **Sprite Editor** button to open **Sprite Editor**.
9. Then, click on the **Slice** button in **Sprite Editor** and select **Type** as **Grid**.
10. Now, select **Pixel Size** for both **X** and **Y** as `32`.
11. Then, select **Pivot** as **Top Left**.
12. Now, click on **Slice** and then on **Apply**.
13. Select the `Pickup_Coin.png` file from the `Assets/Pickup` folder and drag it onto the **Scene** window.
14. A new window will open to create an animation file for it; this is similar to what we did for the character running and jumping.
15. Name this file `Coin_Spin`.
16. Double-click on the `Pickup_Coin_0.controller` file to open **Animator**.
17. Then, right-click on the **Any State** node in the **Animator** window and select **Make Transition**.
18. Now, left-click on the **Coin_Spin** node to connect the nodes.
19. Select the **Pickup_Coin_0** GameObject in the **Hierarchy** tab.
20. In the **Inspector** tab, select **Add Component**.
21. Search for **Box Collider 2D** and select **Box Collider 2D**.
22. In the **Box Collider 2D** component settings, check **IsTrigger**.
23. Change the size of **Box Collider 2D** from **X** as `0.32` and **Y** as `0.32` to **X** as `0.2` and **Y** as `0.2`.

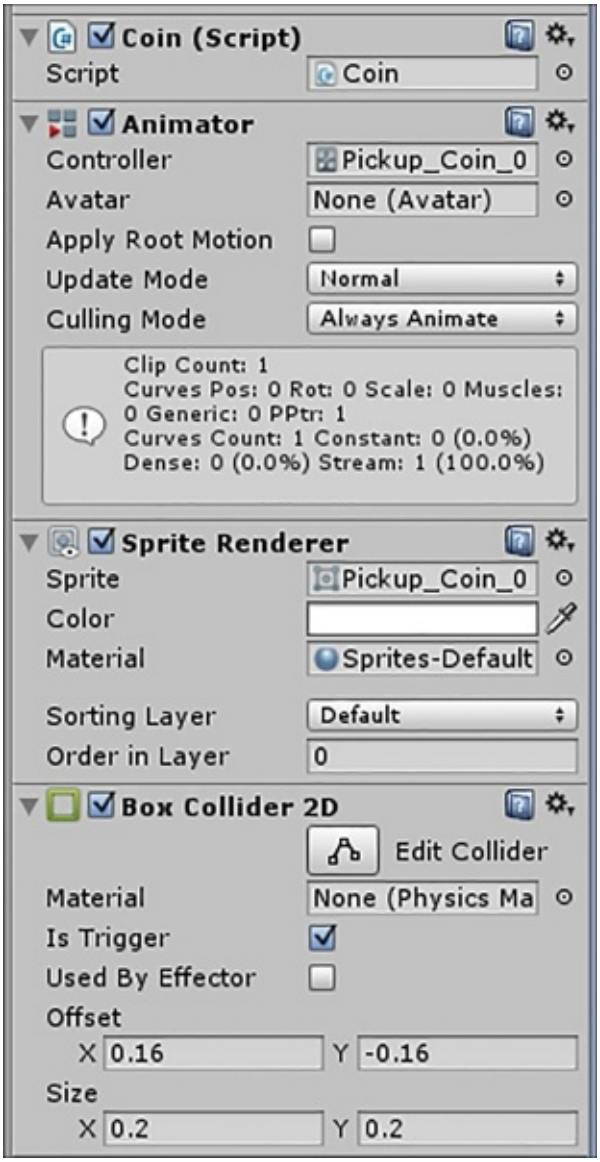
    You will see that these steps are similar to the steps we took for the character and just like the character, this pickup will need a code class for itself, although it will be much smaller than the character.

24. Right-click on the `Assets/Scripts` folder and select **Create** and **C# Class**.
25. Name this C# Class as `Coin`.
26. Then, click on the **Pickup_Coint_0** GameObject in the **Hierarchy** tab and select

**Add Component**.

27. Search for **Coin** and select **Coin**.
28. Double-click on the `Coin.cs` file to open it.

Just as with all the other classes, you will find only the generated code from Unity, including the `Start` and **Update** functions and two `Includes`:




Under the `Update` function, add the following code:

```
// When trigger collider is collided with
void OnTriggerEnter2D(Collider2D Col)
{
  print( Col.gameObject );
}
```

If you position `Pickup_Coin_0` before the character and play the game and jump on the coin you should see the following console message:

```
Character (UnityEngine.GameObject)
UnityEngine.MonoBehaviour:print(Object)
```

Unlike the collider on the character, the trigger collider will not block the colliding object from moving; it will only say that something has collided with it. This is handy for us because we don't want the coins to block the character if it hits the coin. We only want to know that the character has collided with it. `OnTriggerEnter2D` is a Unity function that is similar to the `OnCollisionEnter2D` function we used in character. It will give us the `Collider2D` object that it has collided with so that we can use it to do something, that is, give the character *one* more coin in our case.

Update the `OntriggerEnter2D` function as follows:

```
// When trigger collider is collided with
void OnTriggerEnter2D(Collider2D Col)
{
  Character gameChar;
  if (Col.gameObject.name == "Character")
  {
    gameChar = Col.gameObject.GetComponent<Character>( );
    if (gameChar != null)
    {
      if (!gameChar.isDead)
      {
        gameChar.CoinCount += 1;
        print(gameChar.CoinCount);
        Destroy(gameObject);
      }
    }
  }
}
```

This function will first check whether `gameObject` who owns `Collider2D`, which is being passed to the function, is named `Character`. As there will only be one GameObject named this, we will know that this is our character.

We will then assign a local reference to the `Character` component of the **Character** GameObject and check to make sure that it exists by verifying that it is *not* null.

If the character is not *null*, we will then make sure that it is not dead because we don't want the character to pick up anything when it is dead.

Once we have a valid reference to the `Character` component of the **Character** GameObject and know that the character is not dead, we can then add *one* to the **CoinCount** of the character and print out the current value of coins held by the character.

Once we have given character a coin, we will no longer want this coin to do anything because it's already been used, so we can call the `Destroy` function and give it the `gameObject` of the coin to destroy, removing it from the scene and preventing character from touching and collecting another coin from it.

## Tip

**gameObject** (lowercase **g** in game) is a reference to GameObject that the C# class belongs to.

This is our pickup class. This class is designed to only perform one thing—add *one* to a value of another class—so it is a simple C# class to write.

# Creating the obstacle

The obstacle will consist of two parts. There will be an image of an axe that we will rotate around and a base image that the axe will rotate around. The reason we will be controlling the rotation of the axe instead of using another animation is that it would be more complicated to scale Box Collider 2D on the object to always fit the rotating bounds of the axe. Perform the following steps:

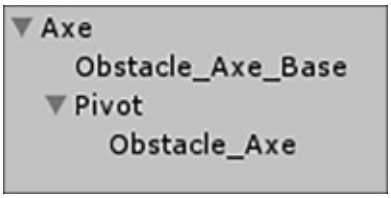
1. To begin with, navigate to the `Assets` folder, right-click on it, select **Create** and then **Folder**.
2. Name the folder `Obstacle` and open this folder.
3. Now, right-click on the `Obstacle` folder and click on **Import New Asset…**.
4. Navigate to the `Art` files for the book and open the `ChapterThree_Obstacle` folder.
5. Then, select `Obstacle_Axe.png` and click on **Import**.
6. Now, right-click on the `Obstacle` folder again and click on **Import New Asset…**.
7. Select `Obstacle_Axe_Base.png` and click on **Import**.
8. Select both the imported files in the `Obstacle` folder (click on one and *Ctrl* + left-click on the other).
9. Now, select **Filter Mode** as **Point**.
10. Then, select **Max Size** as `256`.
11. Now, select **Format** as **Truecolor**.
12. Then, click on **Apply**.
13. Let's click and drag each of these files onto the `Obstacle` folder in the scene. (If character and/or pickup are in the way, move them to the left or right direction so that you have room to work).
14. Now, select **Obstacle_Axe** and **Obstacle_Axe_Base** GameObjects in the scene.
15. In **Inspector**, select both of their positions for **X** as `0`, **Y** as `0`, and **Z** as `0`.
16. For **Obstacle_Axe_Base**, select its **Z** value as `0.1`.
17. Select **Obstacle_Axe** and move it up so that the base of the axe handle is in the center of **Obstacle_Axe_Base**, as shown in the following image:

With **Obstacle_Axe GameObject** selected, navigate to the **Inspector** tab and click on **Add Component.** Search for **Box Collider 2D** and select **Box Collider 2D**.

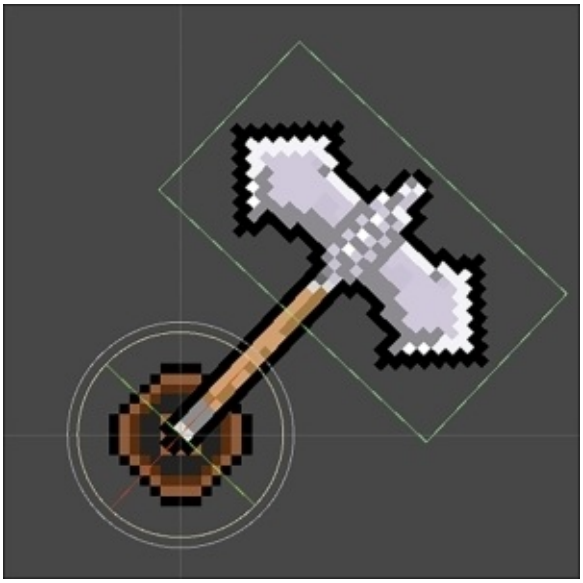
1. In the **Box Collider 2D** settings, check **IsTrigger**.
2. Also, in the **Box Collider 2D** settings, select the offset in the **Y** value as `0.12`.
3. Again, in the **Box Collider 2D** settings, select **Size** of **X** as `0.36` and **Y** as `0.2`.
4. Then, right-click on the **Hierarchy** tab and select **Create Empty**.
5. In the **Inspector** window, name the new GameObject as `Axe` and set its **Position** to **X0**, **Y0**, and **Z0**.
6. Select **Obstacle_Axe** and **Obstacle_Axe_Base** GameObjects and drag them onto the **Axe** GameObject.
7. Right-click on the **Hierarchy** tab and select **Create Empty** again.
8. Rename this new GameObject in the **Inspector** window as **Pivot** and set its **Position** to **X0**, **Y0**, and **Z0**.
9. Now, move the **Obstacle_Axe** GameObject to the **Pivot** GameObject.

If you are having an issue where the pivot isn't at the center of **Obstacle_Axe_Base**, change the pivot position from **Object** to **Pivot**, as shown in the following screenshot:

You should now have GameObject in the **Hierarchy** tab named as **Axe** with three children: **Obstacle_Axe_Base**, **Pivot**, and **Obstacle_Axe**:




You should now be able to select the **Pivot** GameObject and rotate it around so that it looks as if **Axe** is rotating from the center of the **Axe** GameObject:



We now need to create the code class that will handle this obstacle. In the `Assets/Scripts` folder, right-click and select **Create** and then **C# Script**. Name the class `Obstacle`.

Select **Axe** GameObject in the **Hierarchy** tab. Now, in the **Inspector** tab, click on **Add Component**. Search for **Obstacle** and select it.

In the `Assets/Scripts` folder, double-click on the `Obstacle` class to open it. As before, there is the Unity-generated code.

At the top of the class, we need a reference to the **Pivot** GameObject so that we know what GameObject to rotate. Add the following code:

```
// Pivot to rotate around
public Transform Pivot;
```

We also need the setting to set the rotation. We will use `Vector2` and use it as a range. Add the following code under the `Pivot` variable:

```
// Random range for rotation speed
public Vector2 RotationSpeedGap;
```

As `Vector2` is a struct with two float values, we can use both of its values for something other than a location. In this case, we will use them as the minimum and maximum rotation speed.

Now, we need a value that is chosen by `RotationSpeedGap`. Under `RotationSpeedGap`, add the following code:

```
// Actual rotation speed based on RotationSpeedGap
private float RotationSpeed;
```

The concept here is that we can use the Random tool of the Unity engine to pick a value for us to avoid having absolute consistency between all obstacles. This way, each obstacle used will have its own rotation speed, making the game a bit more random and also a bit more difficult.

In the `Start` function, add the following code:

```
// Use this for initialization
void Start ()
{
  RotationSpeed = Random.Range( RotationSpeedGap.x, RotationSpeedGap.y );
}
```

This sets `RotationSpeed` to a random value in the range that starts with `RotationSpeedGap.x` and ends with `RotationSpeedGap.y`.

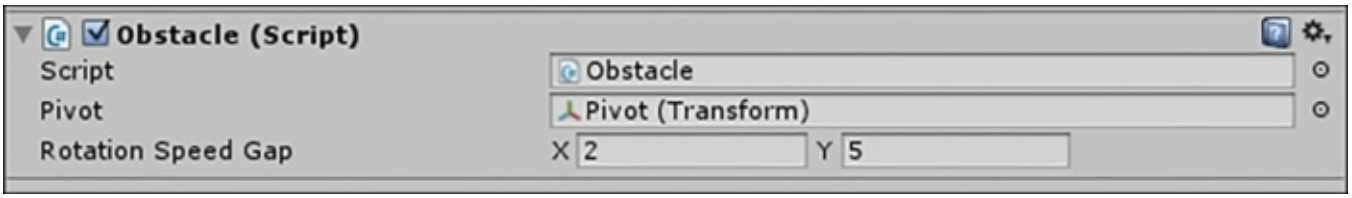
Now, we have to rotate the `Pivot` object based on the rotation speed in the `Update` function. Add the following code:

```
// Update is called once per frame
void Update ()
{
  Pivot.transform.Rotate( Vector3.forward, RotationSpeed );
}
```

The `Rotate` function is designed to take a direction, which is `Vector3.forward` and a speed to rotate around, which is our generated `RotationSpeed` from the Random value between `RotationSpeedGap.x` and `RotationSpeedGap.y`.

Select **Axe GameObject** in the **Hierarchy** tab, and in the **Obstacle** component settings, click on **Pivot** and drag it onto the **Pivot** box of the component in the **Inspector** tab.

For the **Rotation Speed Gap** values, I will use **X** as `2` and **Y** as `5`. You can use anything you'd like; the higher the values, the faster the axe spins.




If you play the game now, you will see the axe spinning towards the player.

The last step of **Obstacle** is to write the logic that will check for the character colliding with it and then kill the character. The downside to how we have the **Axe** obstacle set up is that as the parent object doesn't know of Box Collider 2D on **Obstacle_Axe**, we have to

create another C# class for **Obstacle_Axe** so that we can tell when the character collides with it.

In the `Assets/Scripts` folder, right-click and select **Create** and then **C# Script**.

Name this `Obstacle_Axe`.

Select the **Obstacle_Axe** GameObject in the **Hierarchy** tab, and in the **Inspector** tab, click on **Add Component**.

Search for **Obstacle_Axe** and select **Obstacle_Axe**.

Double-click on the `Obstacle_Axe` C# script to open it.

Select the `Start` and `Update` functions and delete them. These functions are not needed.

The `Obstacle_Axe` C# script should look similar to the following code:

```
using UnityEngine;
using System.Collections;

public class Obstacle_Axe : MonoBehaviour
{
  // When our collider collides with another collider
  void OnTriggerEnter2D(Collider2D Col)
  {
    Character gameCharacter;

    if (Col.gameObject.name == "Character")
    {
      gameCharacter = Col.gameObject.GetComponent<Character>( );
      if (gameCharacter != null)
      {
        gameCharacter.KillCharacter( );
      }
    }
  }
}
```

Similar to the pickup's `OnTriggerEnter2D` function, we will check whether the GameObject owns the **Col** collider named **Character**. If it does, we get the `Character` component on it and make sure it exists. If it does not, we will call the `KillCharacter` function to kill the character.

If you play the game now, you can collide and be killed with **Obstacle**. Remember that there is no reset to revive the character yet. We will do this in the future UI chapter, where we will have the button to restart the game.

# Summary

This chapter gives us an overview of our game. The player has control over the character, who can in turn respond to both negative and positive GameObjects it comes across in the game world.

You'll see a trend in the code. Unity gives us some basic tools to create the game logic with little effort. If the game had a bigger depth, the amount of code needed would have gone up, although the concept of getting references of other GameObjects and manipulating them would stay the same.

In the next chapter, we will begin to explore **prefabs**, which we will use to create the game world.

# Chapter 4. Editor Tool, Prefabs, and Game Level

In the last chapter, we set up what will become the core of our gameplay. The player has the ability to send input to the device, and we will handle this by manipulating the player character GameObject. We also set up some game logic so that the player character can interact with positive and negative world objects, such as `Coins` and `Obstacles`. To further develop the sense of a complete game, we need to create the pieces of the game world that represent a floor that the player will run on.

To create these pieces, we will create a Unity `EditorWindow` class that will help us create grids that will represent the ground the player runs on and the dirt below it. Traditionally, you would have to place each sprite one at a time. With this editor tool, we will be able to create bigger boxes in a grid based on our settings.

After we have our editor tool running, we will begin to create the prefabs that will hold multiple GameObjects and their components in a single file. Finally, we will write the code needed to move the floor and ground pieces below the player character, simulating the character as running forward.

To summarize, in this chapter, we will cover the following topics:

- Writing a Unity C# class that extends `EditorWindow`, which allows you to input settings and sprite files that will give you a box grid and simplify the level pieces creation.
- Creating the game-related prefabs so that you have grouped files in an easy-to-use file.

Use the prefabs we made in the C# script. This will move the level pieces of prefabs under the player character, simulating movement.

# Making the sprite tile editor tool

The Unity engine is incredibly flexible for all the aspects of game development, including creating custom editor tools to help fast track the more tedious aspects of development. In our case, it will be beneficial to have a tool that creates a root GameObject that will then create children GameObjects in a grid. The grid will be spaced out by the size of the sprite component that each of the children GameObject is using.

For example, if you were to place say *24* GameObjects one at a time, it could take some time to make sure that all are snapped correctly together. With our tool, we will be able to select the `X` value and the `Y` value for the grid, the sprite that represents the ground, and the sprite that represents the dirt below the ground. Perform the following steps:

1. To begin with, navigate to the `Assets` folder. Right-click on this folder and select **Create** and then **New Folder**. Name this folder `Level`.
2. Right-click on the new `Level` folder and select **Import New Asset…**.
3. In the files for this book, navigate to the folder named `Art` and then `ChapterFour_Floor`. Import all the four floor files: `Floor_00`, `Floor_01`, `Floor_02`, and `Floor_03`. We will use these files to create our game-level pieces.
4. Right-click on the `Script` folder, select **Create** and then **C# Script**. Name the script `SpriteTiler`.

# The SpriteTiler C# class

Double-click on the `SpriteTiler` C# file to open it. Change the file so that it looks similar to the following code:

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

public class SpriteTiler : EditorWindow
{
}
```

The big changes from the normally generated code file, is the addition to `using UnityEditor`, changing the inherited class to `EditorWindow`, and removing the `Start( )` and `Update( )` functions.

# Global variables

We now want to add the global variables for this class. Add the following code in the class block:

```
// Grid settings to make tiled by
public float GridXSlider = 1;
public float GridYSlider = 1;

// Sprites for both the ground and dirt
public Sprite TileGroundSprite;
public Sprite TileDirtSprite;

// Name of the GameObject that holds our tiled Objects
public string TileSpriteRootGameObjectName = "Tiled Object";
```

The `GridXSlider` and `GridYSlider` values will be used to generate our grid, `X` being left to right and `Y` being top down. For example, if you had `X` set to *five* and `Y` set to *three*, the grid would generate columns of *five* elements and rows of *three* elements or *five* sprites long and *three* sprites down.

The `TileGroundSprite` and `TileDirtSprite` sprite files will make up the ground and dirt levels.

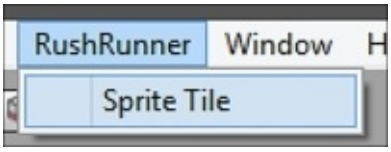
`TileSpriteRootGameObjectName` is the GameObject name that will hold the GameObjects children that have the Sprite components. This is editable by you so that you can choose the name of the GameObject that gets created to avoid having the default *new GameObject* for each one made.

# The MenuItem creation

Next, we need to create the `MenuItem` function. This will represent the **Editor** selection drop-down list so that we can use our tool. Add the following function to the `SpriteTiler` class under the global variables:

```
// Menu option to bring up Sprite Tiler window
[MenuItem("RushRunner/Sprite Tile")]
public static void OpenSpriteTileWindow()
{
  EditorWindow.GetWindow< SpriteTiler > ( true, "Sprite Tiler" );
}
```

As this class extends `EditorWindow`, and the preceding function is declared as `MenuItem`, it will create a dropdown in the **Editor** named **RushRunner**. This will hold a selection called **Sprite Tile**:




## Tip

You can name the dropdown and selection anything you like by changing the string that is passed into `MenuItem`, such as “MyEditorTool / Editor Tool Name”.

If you save the `SpRiteTiler.cs` file and go back to Unity and allow the engine to compile, you will be able to click on the **Sprite Tile** button under **RushRunner**. This will create an editor window named **Sprite Tiler**.

# The OnGUI function

Next, we need to add the function that will be used to draw all the window GUI elements or the fields that we will use to get the settings to make the grid. Under our `OpenSpriteTileWindow` function, add the following code:

```
    // Called to render GUI frames and elements
    void OnGUI()
    {
    }
```

`OnGUI` is the function that will draw our GUI elements to the window. This allows you to manipulate these GUI elements so that we have values to use when we create the GameObject grid and its children GameObjects with sprite components.

## The GUILayout and OnGUI setup

To begin with the `OnGUI` function, we want to add the GUI elements to the window. In the `OnGUI` function, add the following code:

```
        // Setting for GameObject name that holds our tiled Objects
        GUILayout.Label("Tile Level Object Name", EditorStyles.boldLabel);
        TileSpriteRootGameObjectName = GUILayout.TextField(
TileSpriteRootGameObjectName, 25 );

        // Slider for X grid value (left to right)
        GUILayout.Label("X: " + GridXSlider, EditorStyles.boldLabel);
        GridXSlider = GUILayout.HorizontalScrollbar( GridXSlider, 1.0f,
0.0f, 30.0f );
        GridXSlider = (int)GridXSlider;

        // Slider for Y grid value(up to down)
        GUILayout.Label("Y: " + GridYSlider, EditorStyles.boldLabel);
        GridYSlider = GUILayout.HorizontalScrollbar(GridYSlider, 1.0f,
0.0f, 30.0f);
        GridYSlider = (int)GridYSlider;

        // File chose to be our Ground Sprite
        GUILayout.Label("Sprite Ground File", EditorStyles.boldLabel);
        TileGroundSprite = EditorGUILayout.ObjectField(TileGroundSprite,
typeof(Sprite), true) as Sprite;

        // File chose to be our Dirt Sprite
        GUILayout.Label("Sprite Dirt File", EditorStyles.boldLabel);
        TileDirtSprite = EditorGUILayout.ObjectField(TileDirtSprite,
typeof(Sprite), true) as Sprite;
```

`GUILayout.Label` is a function that creates a text label in the window we are using. Its first use is to let the user know that the next setting is for **Tile Level Object Name**: the name of the root GameObject that will hold children GameObjects with sprite components. By default, this is set to **Tiled Object**, although we allow the user to change it.

In order to allow the user to change it, we need to give them a `TextField` parameter to

input a new string. We do this by saying that `TileSpriteRootGameObjectName` is equal to the `GUILayout.TextField` setting. As this is used in `OnGUI`, anything the user inputs will change the value of `TileSpriteRootGameObjectName`. We will use this later when the user wants to create the GameObject.

We then need to create two `HorizontalSlider` GUI elements so that we can get values from them that represent the `X` and `Y` values of the grid. Similar to `TextField`, we can start each of the `HorizontalSlider` elements with `GUILayout.Label`. This describes what the slider is for. We will then assign the `GridXSlider` and `GridYSlider` values to what the `HorizontalSlider` is set to, which is *one* by default.

As the user adjusts the sliders, the `GridXSlider` and `GridYSlider` values will change so that when the user clicks on a button to create the GameObject, we will have a reference to the values that they want to use for the grid.

After `HorizontalSliders`, we want to have `ObjectFields` so that the user can search for and assign sprite files that will represent the ground and dirt of the grid. The user then uses `EditorGUILayout.ObjectField` to select a sprite reference from the `Asset` folder. As we assigned `ObjectField` to only accept sprites, that is all that the user will see when they want to input one. As we want this `ObjectField` to be for sprites, we will set the type of object to `typeof( Sprite )` and then cast the result that is assigned to `TileGroundSprite` or `TileDirtSprite` to the sprite by using `as Sprite`.

# The OnGUI create tiled button

In order to know when the user wants to create the root GameObject and its grid of children GameObjects, we will need a button. Add the following code under the last GUI elements:

```
    // If butt "Create Tiled" is clicked
    if (GUILayout.Button("Create Tiled"))
    {
      // If the Grid settings are both zero,
      // send notification to user
      if (GridXSlider == 0 && GridYSlider == 0)
      {
        ShowNotification(new GUIContent("Must have either X or Y grid set
to a value greater than 0"));
        return;
      }

      // if Dirt and Ground Sprite exist
      if (TileDirtSprite != null && TileGroundSprite !=null)
      {
        // If the Sprites sizes don't match,
        // send notification to user
        if (TileDirtSprite.bounds.size.x != TileGroundSprite.bounds.size.x
|| TileDirtSprite.bounds.size.y != TileGroundSprite.bounds.size.y)
        {
          ShowNotification(new GUIContent("Both Sprites must be of matching
size."));
          return;
        }

        // Create GameObject and tiled
        // Objects with user settings
        CreateSpriteTiledGameObject(GridXSlider, GridYSlider,
TileGroundSprite, TileDirtSprite, TileSpriteRootGameObjectName);
      }
      else
      {
        // If either Dirt or Ground Sprite don't exist,
        // send notification to user
        ShowNotification( new GUIContent( "Must have Dirt and Ground Sprite
selected." ) );
        return;
      }
    }
```

The first condition we have set is the `GUILayout.Button( "Create Tiled" )` function. The `Button` function will return true as soon as it is clicked on, but it will still render to the window if `false`. This means that although the button is not active, it'll still be seen by the user.

As some settings will create a scenario that is not ideal for the concept of our `SpriteTiler` function, we first want to make sure that the settings are in line with what we have designed the tool to perform.

We will first check whether `GridXSlider` and `GridYSlider` are set to *zero*. If both of these values are set to *zero*, the grid won't create anything, and as the concept of the tool is to create a grid of children sprites, we will tell the user that they must have a selection above *zero* for either `GridXSlider` or `GridYSlider`.

We then check whether `TileDirtSprite` and `TileGroundSprite` have a value. If either of these values are *null*, the settings are not complete. This results in you telling the user that dirt and ground sprites need a selection.

If the user has set dirt and ground sprites to something, but their sizing is not the same, such as one being 32 x 32 and the other being 64 x 64, we will tell the user that both the sprites need to be of the same size. If we didn't check for this, the grid wouldn't align correctly, creating negative results and making the tool not function as we want it to.

If the user settings are in order, we will call the `CreateSpriteTiledGameObject` function and pass `GridXSlider`, `GridYSlider`, `TileGroundSprite`, `TileDirtSprite`, and `TileSpriteRootGameObjectName`.

## The CreateSpriteTiledGameObject function

This function is designed to take the user settings and create the grid from them. Add the following function under the `OnGUI` function:

```
// Create GameObject and tiled children based on user settings
public static void CreateSpriteTiledGameObject(float GridXSlider, float
GridYSlider, Sprite SpriteGroundFile, Sprite SpriteDirtFile, string
RootObjectName)
{
  // Store size of Sprite
  float spriteX = SpriteGroundFile.bounds.size.x;
  float spriteY = SpriteGroundFile.bounds.size.y;

  // Create the root GameObject which will hold children that tile
  GameObject rootObject = new GameObject( );

  // Set position in world to 0,0,0
  rootObject.transform.position = new Vector3( 0.0f, 0.0f, 0.0f );

  // Name it based on user settings
  rootObject.name = RootObjectName;

  // Create starting values for while loop
  int currentObjectCount = 0;
  int currentColumn = 0;
  int currentRow = 0;
  Vector3 currentLocation = new Vector3( 0.0f, 0.0f, 0.0f );

  // Continue loop until all rows
  // and columns have been filled
  while (currentRow < GridYSlider)
  {
    // Create a child GameObject, set its parent to root,
    // name it, and offset its location based on current location
    GameObject gridObject = new GameObject( );
```

```
    gridObject.transform.SetParent( rootObject.transform );
    gridObject.name = RootObjectName + "_" + currentObjectCount;
    gridObject.transform.position = currentLocation;

    // Give child gridObject a SpriteRenderer and set sprite on CurrentRow
    SpriteRenderer gridRenderer = gridObject.AddComponent<SpriteRenderer>(
);
    gridRenderer.sprite = ( currentRow == 0 ) ? SpriteGroundFile :
SpriteDirtFile;

    // Give the gridObject a BoxCollider
    gridObject.AddComponent<BoxCollider2D>();

    // Offset currentLocation for next gridObject to use
    currentLocation.x += spriteX;

    // Increment current column by one
    currentColumn++;

    // If the current column is greater than the X slider
    if (currentColumn >= GridXSlider)
    {
      // Reset column, increment row, reset x location
      // and offset y location downwards
      currentColumn = 0;
      currentRow++;
      currentLocation.x = 0;
      currentLocation.y -= spriteY;
    }

    // Add to currentObjectCount for naming of
    // gridObject children.
    currentObjectCount++;
  }
}
```

To start with, we must first have the `X` and `Y` sizes of the sprite we want to create so that we can offset the location of the children GameObjects that were created. As we originally checked to make sure that both sprites are of the same size, it doesn't matter which sprite object we get the size from. In our case, we will use `SpriteGroundFile`.

We will then move the `rootObject` position to `0X`, `0Y`, and `0Z` so that it is in the center of our scene. This can be set to anything you like, although when `rootObject` and its children get created, it is easier to find it at the center of the scene world.

After it has been moved, we can set its name to the setting that the user had entered or **Tiled Object** (the default).

Once we have `rootObject` set up, we can create its children GameObjects. To start this cycle, we will need a few variables to reference and change:

- `currentObjectCount`: This specifies the total number of children that will be created. This increments for each one created.
- `currentColumn`: This denotes the current column we are on in the row.

- `currentRow`: This specifies the current row we are on.
- `currentLocation`: This denotes the current location that the children GameObject will use and sets its position too. This is changed after each new child is created based on the `X` or `Y` setting of the sprite size.

Now that we have our `rootObject` and the variables we need to create the children, we can use a `while` loop. A `while` loop is a loop that will continue until its condition fails. In our case, we will check whether `currentRow` is less than the `GridYSlider` value. As soon as `currentRow` is equal to or greater than `GridYSlider`, the loop will stop because the condition failed.

The reason we will look at `currentRow` is that for each column created, we can reset its value to *zero* and increment `currentRow` by *one*. This means that each row will hold as many columns as were set by the `GridXSlider` value, and we know that the grid is complete when `currentRow` is equal to or greater than `GridYSlider`.

For example, if we had a grid setting of `3X` and `3Y`, the first row will hold *three* columns. When the first row is done, the row changes to *two* and adds three more columns. In the last row, it completes *three* more columns and then the `while` condition fails because the row value is equal to `GridYSlider`.

In each loop of the `while` loop, we start by creating `gridObject`. We set this grid object's parent to that of `rootObject`, set its name to `RootObjectName`, and concatenate an underscore, followed by `currentObjectCount` and then set the `gridObject` position to the `currentLocation` value, which will change based on the size of the sprite and the column/row.

We will then add a `SpriteRenderer` component to `gridObject` and assign a sprite to it. We will change the sprite based on whether `currentRow` is equal to *zero* or not. If it is, in the first row, we will set the sprite to `SpriteGroundFile`. If `currentRow` is not equal to *zero*, we will set the sprite to `SpriteDirtFile`.

The **ternary operator** is a sort of shorthand for `if else`. If the condition is `true`, we will set the value to what is behind the *question mark*. If the condition is `false`, we will set the value based on what's behind the *colon*. The *question mark* represents if, whereas the *colon* represents `else`.

## Tip

The ternary operator is as follows:

```
Value = ( condition == true ) ? ifTrue : elseNotTrue;
```

Once we have the sprite assigned to the `SpriteRenderer` component of `gridObject`, we can assign a **Box Collider 2D** component, and the **Box Collider 2D** component will make itself the same size as the sprite. If we were to add the **Box Collider 2D** component to `SpriteRenderer`, it would be the default size of 1,1,1, which would be too big.

We will then offset `currentLocation` by the `spriteX` size, so the next `gridObject` will offset the size of the `spriteX` size.

The `currentColumn` parameter is incremented by *one*, and we then check whether `currentColumn` is greater than or equal to the `GridXSlider` value. If it is, we know that we need to start the next row.

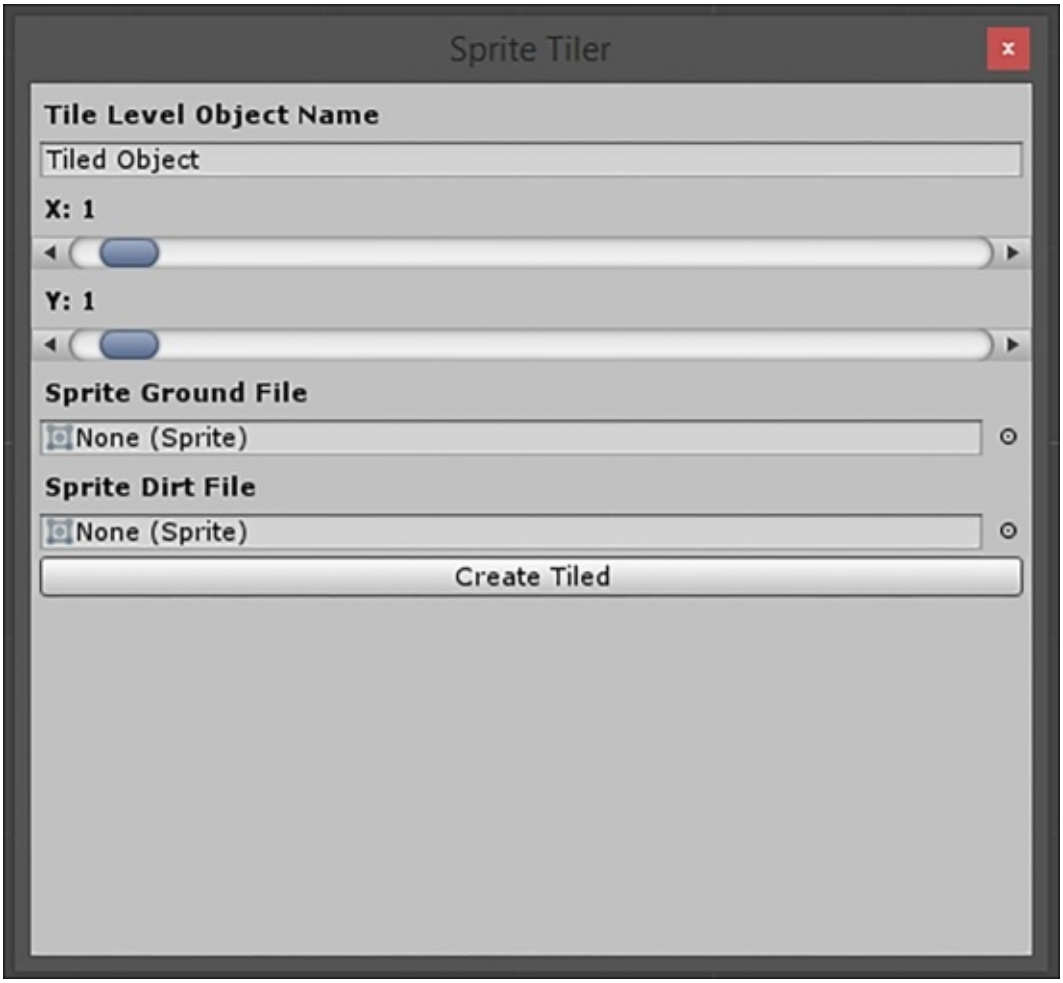
To do this, we reset `currentColumn` to *zero*, increment `currentRow` by *one*, set the `currentLocation.x` value to *zero*, and offset `currentLocation.y` by a negative `spriteY` size. This not only results in an offset location down, but also resets the `X` value to zero, making it possible for the columns to be created again; just down from the size of `spriteY`.

Finally, we increment `currentObjectCount` by *one*.

## Testing the SpriteTiler file

Make sure to save the `SpriteTiler.cs` file and then go back to Unity and allow it to compile.

When it does, you should continue to see the **RushRunner** drop-down selection on the top editor bar. Click on it and then on its child: **Sprite Tile**. This should open the **Sprite Tiler** window with all of our settings:

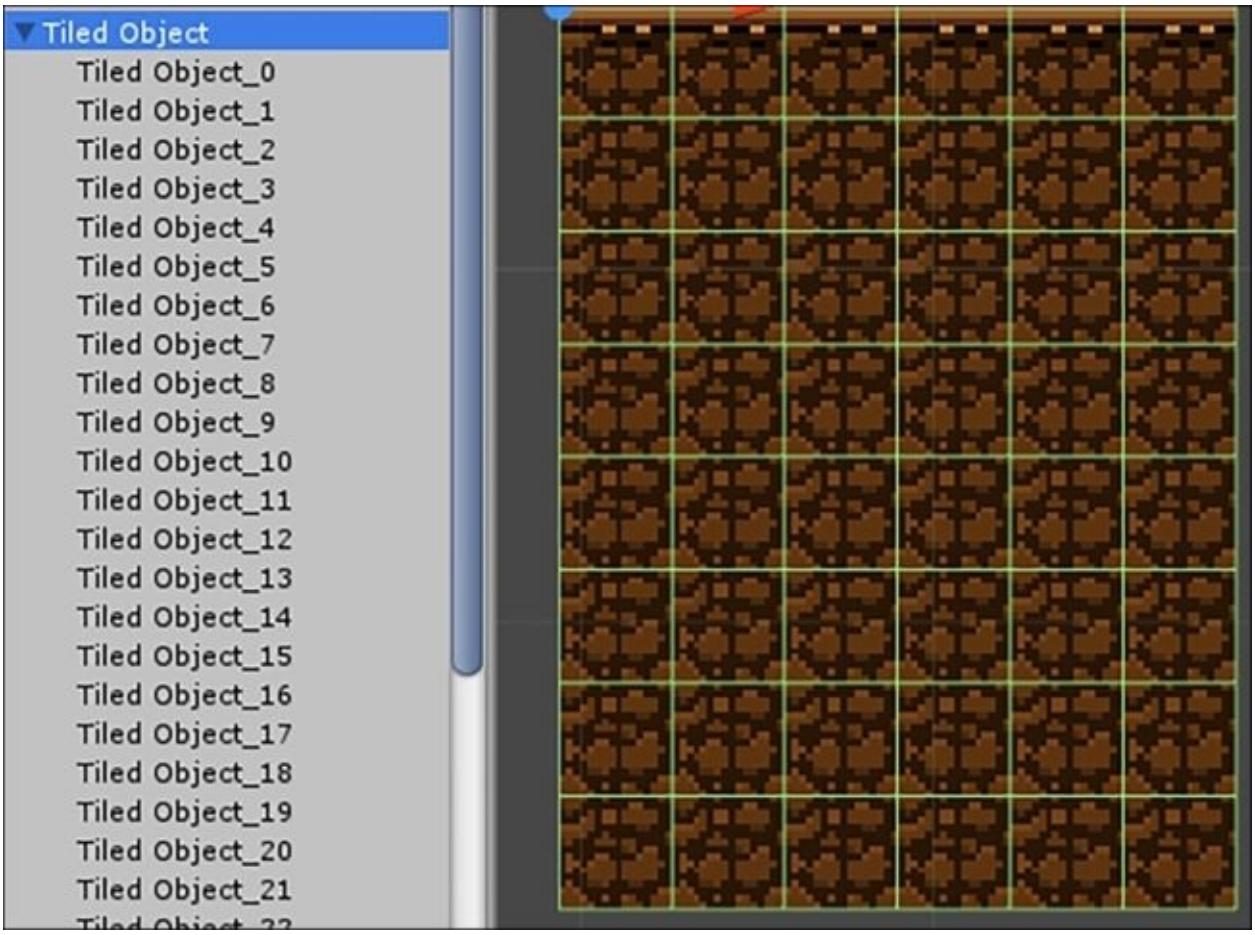



The **Sprite Tiler** window now allows you to input the settings so that you can begin to create GameObjects, which hold a grid of children GameObjects, which in turn hold a sprite component of what is set within the **Sprite Tiler** window.

To begin, follow these steps:

1. To test **Sprite Tiler**, you can keep the default **Tile Level Object Name**.
2. Now, change **X** to a bigger value.
3. Then, change **Y** to a bigger value.
4. Now, click on the small circle to the right of **Sprite Ground File** and add `Floor_00`.
5. Then, click on the small circle to the right of **Sprite Dirt File** and add `Floor_01`.
6. Finally, click on **Create Tiled**.

I chose **X** = `6`, **Y** = `8`, and the grid looks similar to the following screenshot:




## Tip

If you want to test a bigger grid, choose `Floor_03` as the ground sprite and `Floor_02` as the dirt sprite. These images are bigger; however, because they are of the same size, you can use them as `Floor_00` and `Floor_01`. They will make a bigger floor in the `X` axis because they are much wider.

## Sprite Tiler wrap-up

This editor tool allows you to create GameObjects that hold child GameObjects that have `SpriteRenderers` on them. This sort of method can be extended and modified to create more complex shapes and use different types of sprite images to create them. For now, this allows you to create the basic structure of our game-level pieces with much less effort than

if we didn’t have this tool to assist us.

Extending Unity’s `EditorWindow` allows you and your team to create an assortment of custom tools that can handle any sort of efficiency or workflow problem specific to the type of game you will create.

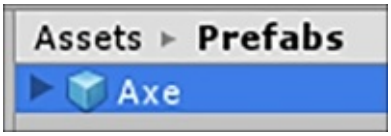
# The prefab creation

In order to create a prefab, you must first have a collection of GameObjects that you want to keep together as one prefab.

Perform the following steps:

1. Navigate to the `Assets` folder. Right-click on it and select **Create** and then **Folder**. Name this `Prefabs`.
2. Select the **Axe** GameObject in **Hierarchy**. This is the one we created in the last chapter. Click on it and drag it onto the `Prefabs` folder.

This will create an **Axe** prefab that we can use in the future that holds everything the current **Axe** GameObject does.




Anytime in the future, when we want to create an axe, we will have a complete file that holds everything we know as axe.

This is essentially all it takes to create prefabs. These are an incredibly simple and useful aspect of creating games with Unity.

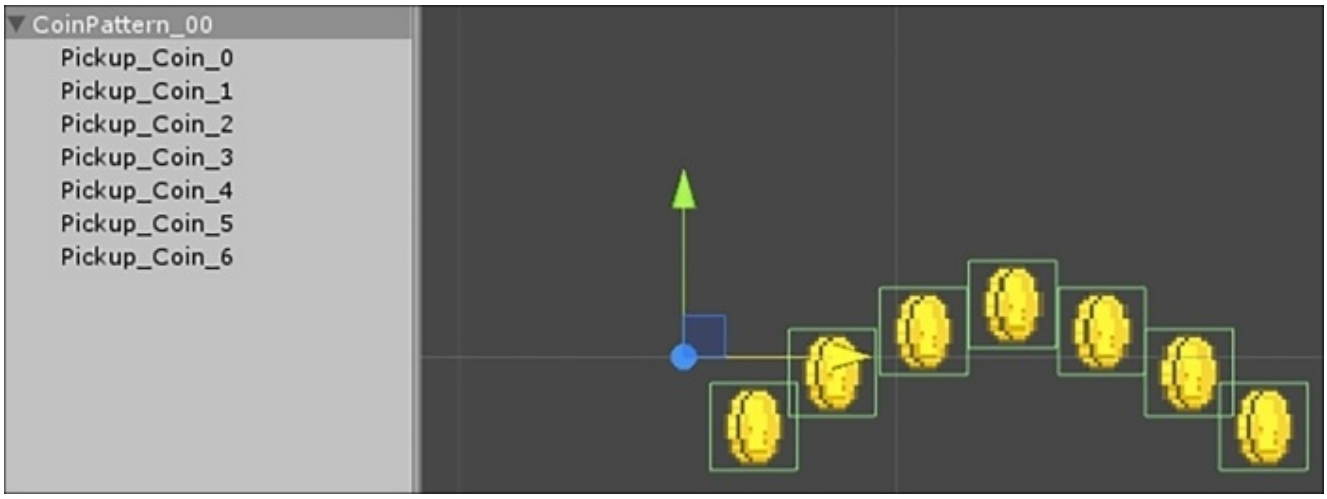
# Coin prefabs

Let's create a couple of different varieties of coins, so when we use them in our level, we will have a more efficient system. Perform the following steps:

1. Start by creating a new empty GameObject in the **Hierarchy** tab and name it `CoinPattern_00`.
2. Set the location of this **CoinPattern_00** GameObject to `0X`, `0Y`, and `0Z` in the world.

   Now, move the existing coin in the scene. Name it `Pickup_Coin_0` so that it becomes a child of the **CoinPattern_00** GameObject. Center this coin to `0X`, `0Y`, and `0Z`. This location is referenced by its parent, so it is a local position and not a world one. The **CoinPattern_00** GameObject has the world location and the `Coin` child is relative to its position.

3. Select the `Coin` child and click on *Ctrl + D* to duplicate it. Do this a few times and lay out the coins in an arch, as shown in the following screenshot:




4. With the **CoinPattern_00** GameObject selected, click on it and drag it onto the `Assets/Prefabs` folder.

Once the prefab is created for **CoinPattern_00**, you can drag a prefab onto the scene and see the power of prefabs. As the prefab stores all the references you created, when they are put into the scene, the prefab is exactly the same as the **CoinPattern_00** GameObject, which we created earlier.

Follow the same steps to create a couple of different **CoinPattern** GameObjects and make sure to create prefabs for all of them. I made three **CoinPattern** GameObjects and turned them into a prefab.

## Tip

If you want to alter a prefab, enter it in the scene and make your changes. When you are done with your changes, click on the parent GameObject and then drag it onto the existing prefab you want to edit. There is also an option in the **Inspector** window that says **Apply**.

If you click on this, the prefab will be updated to the current settings in the **Scene** window.

# Level piece prefabs

The last step of our prefab setup is to create the level pieces. These will be the pieces we put together with some code to create a “random” level. As the level pieces will be moving under the player, simulating the player moving, we will have to move these level prefab pieces around in the world and introduce coin and obstacles around them, so the player has a somewhat different experience as the game is running.

To start with, use our **Sprite Tiler** tool to create some tiled GameObjects. I suggest adding some slopes by taking the floor sprites, duplicating them and then rotating them. Here is an example:

This is a grid GameObject created with the **Sprite Tiler** tool with an additional row copied to the top and a couple more single grid objects duplicated and rotated. Use your imagination and create at least *three* of these. When you create these, make sure to keep the `Y` grid amount the same for the beginning and ending of all pieces. I will use *five* for a `Y` setting, as seen in the preceding image.

When you are done, take each of the root GameObjects that were created and create prefabs for each of them. Remember to make them unique prefabs with single root GameObjects because we will use each grid individually.

In addition, we will need a starting level piece. This will be the level piece that always starts the game. It will give the player a longer amount of time to get situated with the game and will give us a starting point to begin moving and attaching the other level pieces together. Perform the following steps:

1. Open the **Sprite Tiler** tool and name the **Tile Level Object Name** as `StartingLevelPiece`.
2. Then, select **X** as **3**.
3. Now, select **Y** as **5**.
4. For the **Sprite Ground File**, use `Floor_03`.
5. Then, for the **Sprite Dirt File**, use `Floor_02`.

6. Click on **Create Tiled**.
7. Finally, left-click and drag the resulting **StartingLevelPiece** GameObject onto the `Assets/Prefabs` folder to create its prefab.

# Further setup of the level pieces

At this point, you should have a collection of **CoinPattern**, **LevelPieces**, and **Axe** GameObject(s) prefabs in the `Prefabs` folder. This is the list I have:

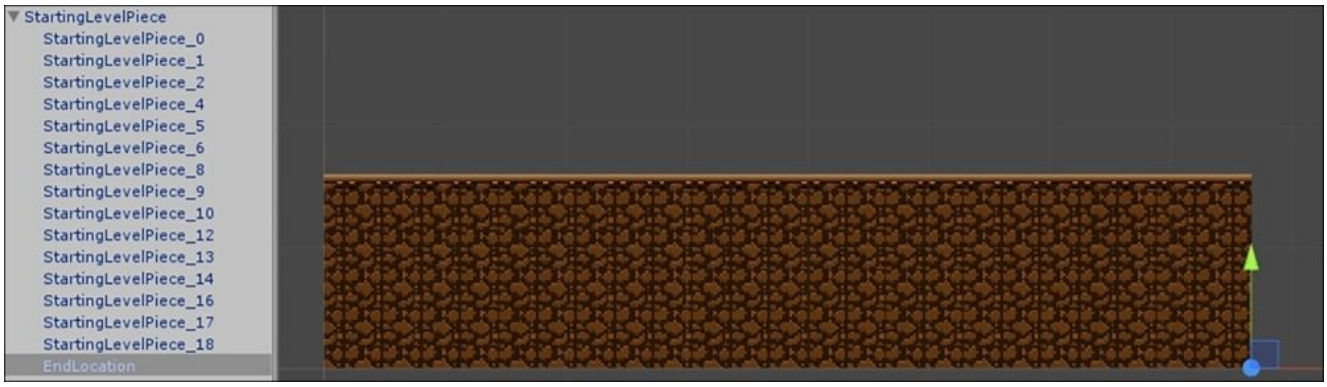
The level pieces will need two extra GameObjects so that we know where they all begin and end. This way, we will be able to place them together by putting the beginning of the next piece at the end of the previous one.

Make sure that any of the level pieces in the **Hierarchy** tab are saved as a prefab and delete them from the **Hierarchy** tab. When the scene is clear of level pieces, take **StartingLevelPiece** and put it in the scene with the following steps:

1. Select **StartingLevelPiece** and position its location at `0`, `0`, and `0` in the world with the **Inspector** window under **Transform**.
2. Open the **StartingLevelPiece** GameObject in **Hierarchy** by clicking on the small gray arrow on the left-hand side.
3. Select the children GameObjects and then position them so that the pivot of the **LevelPiece** parent GameObject is at the *bottom-left* of the selected children, as shown in the following image:

4. Right-click on **Hierarchy**, select **Create Empty**, and name it `EndLocation`.
5. Now, move the **EndLocation** GameObject in the *bottom-right* corner of **StartingLevelPiece**.
6. Take the **EndLocation** GameObjects and drag it onto the **StartingLevelPiece** GameObject so that it becomes its child.

7. Select the **StartingLevelPiece** GameObject and then click and drag it onto the existing prefab in the `Assets/Prefabs` folder. This will overwrite the current version of the prefab with the updated one.
8. If you click and drag the **StartingLevelPiece** prefab from the `Assets/Prefabs` folder and place it on the **Scene** window, you should see **EndLocation** as a child of it in **Hierarchy** and positioned in the *bottom-right* corner of the sprite children, as shown in the following image:




## Tip

When you place the pivot point and `EndLocation`, it is okay to have a few pixels of the sprites overlap; I even recommend it. If these locations are outside the pixels of the sprites, there will be small gaps when the pieces are put together, which isn't a huge deal, although noticeable.

This will give us a position we know as the root object position, which when we change it in code will move the rest of the children in a way we can rely on. The **EndLocation** GameObject also gives us a reliable position to connect another **LevelPiece** to it without needing to have any offsets.

Repeat these steps for the rest of the **LevelPiece** GameObjects. Make sure that the root GameObject is placed in the *bottom-left* corner of the GameObject, and the **EndLocation** is placed in the *bottom-right* corner of the GameObject.

# The level piece code

In order to keep track of all the **LevelPieces**, we need a small script that stores their initial location. Right-click on the `Assets/Scripts` folder and select **Create**. Then, click on **C# Script** and name it `LevelPiece`.

Double-click on the `LevelPiece` file to open it and make sure that it looks similar to the following code:

```
using UnityEngine;
using System.Collections;

public class LevelPiece : MonoBehaviour
{
  // The initial location of this level piece
  private Vector3 InitialLocation;

  // Use this for initialization
  void Awake ()
  {
    InitialLocation = transform.position;
  }

  // Get the initial location of this
  // LevelPiece
  public Vector3 GetInitialLocation()
  {
    return InitialLocation;
  }
}
```

This is a very simple class that is designed to keep track of where the **LevelPiece** starts. This will be used to reset the **LevelPiece** after it has been used so that it can be used again.

`InitialLocation` is a `Vector3` struct that gets assigned during the `Start` function. `Awake` is called as soon as the class is initialized or as soon as the game starts running because the class will exist on the **LevelPieces**, which will be in the world when the game starts.

We don't need the `Update` function, which is why it was removed from the class.

`GetInitialLocation` is a function that we can use from the next C# class that we will write so that it can manage the **LevelPiece** GameObject(s) by getting that location and moving the **LevelPiece** to it after we have used it.

## Tip

The difference between Awake and Start is that Awake gets called before the Update or Start function is called. Then, Start gets called following Awake, but still before the first Update call. This means the order of functions getting called when the class is created is Awake->Start->Update. As the next class we will use will immediately start its Update function, we need to make sure that InitialLocation of LevelPiece is assigned before its location gets moved.

# The level piece manager code

We now have everything we need to begin piecing the **LevelPieces** together with code.

Perform the following steps:

1. Right-click on the `Assets/Scripts` folder and select **Create**. Then, select **C# Script**. Name it `LevelPieceManager`.
2. In order to know when we need to attach another level piece, we must first have a `Vector3` position to check against. In **Hierarchy**, right-click and select **Create Empty**. Name this `LevelPieceManager`.
3. Now, select the **X** location as `LevelPieceManager` and the **Inspector | Transform** window as `-7.75`. This will offset `LevelPieceManager` so that when the game starts, the character will have a longer flat path to run along before reaching the first attached **LevelPiece**.
4. Then, select **LevelPieceManager**, and in **Inspector**, click on **Add Component**. Search for `LevelPieceManager` and select **LevelPieceManager**.
5. Finally, double-click on the `LevelPieceManager` script file in the `Assets/Scripts` folder, open it, and add the following code to it:

```
using UnityEngine;
using System.Collections;

public class LevelPieceManager : MonoBehaviour
{
  // Starting level piece
  public LevelPiece StartingLevelPiece;

  // Level pieces to rotate
  public LevelPiece[] LevelPieces;

  // How quickly the level pieces move
  public float LevelPiecesMoveRate;

  // The currently active Level Piece
  private LevelPiece[] ActiveLevelPieces;

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

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

`LevelPieceManager` is a class that will connect all the `LevelPieces` together and move them across the screen, simulating that the player is running forward. This is a design decision on my end. Although Unity does have what you can call a “scene size limit”, my

decision here is to try to keep the organization of the scene to a smaller area, hopefully giving you a more relaxed scene to work in.

The current code is explained here:

- `StartingLevelPiece`: This is the initial **LevelPiece** GameObject. It is a flat portion to get the game started without having to worry about placing anything as soon as the game starts.
- `LevelPieces`: This is an array that will hold the other `LevelPiece` prefabs we created. It is the list that will be used to assign a new piece to the actively moving pieces when another piece has reached a point where it is no longer needed, that is, when it is out of view of the player.
- `LevelPiecesMoveRate`: This is the speed at which the **LevelPiece** GameObjects will be moving across the screen.
- `ActiveLevelPieces`: This is a list of **LevelPiece** GameObjects that will move across the screen. It is set to private to prevent any other class from accessing and/or changing it.

Next, we want to add the code for the `Start` function. Add the following code to the `LevelPieceManager` class:

```
  // Use this for initialization
  void Start( )
  {
    ActiveLevelPieces = new LevelPiece[2];
    ActiveLevelPieces[0] = StartingLevelPiece;
    ActiveLevelPieces[1] = GetRandomLevelPiece();

    ActiveLevelPieces[1].transform.position =
StartingLevelPiece.gameObject.transform.FindChild("EndLocation").position;
  }
```

The `Start` function is called as soon as this class is active in the scene, just after `Awake` and just before `Update`, giving us room to assign some values before anything else happens in this class.

To begin with, we will create an array of *two* `LevelPiece` classes and assign it to `ActiveLevelPieces`.

Then, we will give the first element in the `ActiveLevelPiece` array the value of `StartingLevelPiece` because it is the first **LevelPiece** we want to use.

We will then get a “random” **LevelPiece** from the `GetRandomLevelPiece` function, which we will write soon; it is going to cycle through the array of `LevelPieces` from the scene. Make sure that the one we want to use isn’t currently active.

After we have the random **LevelPiece**, we have to assign its location to the **EndLocation** of the **StartingLevelPiece**. This is done by using the `transform.FindChild` function. This function will look for the name of the **Transform** child, which in our case is **EndLocation**. When we have this, we can use the transform position of **EndLocation** and assign the random `LevelPiece` position to it.

Earlier, I mentioned why the `LevelPiece` class uses `Awake` instead of `Start`. In the case of `LevelPieceManager`, we want to make sure that the `LevelPiece` class is done with all its assignments before `LevelPieceManager` begins to do anything, which will happen because the `Awake` function of **LevelPiece** will be called before the `Start` function of `LevelPieceManager`, giving us the order of `LevelPiece Awake`, `LevelPieceManager Start`, `LevelPieceManager Update`.

Next, we want to write the `Update` function for `LevelPieceManager`. Add the following code:

```
  // Update is called once per frame
  void Update ( )
  {
    for (int i = 0; i < ActiveLevelPieces.Length; i++)
    {
      Vector3 newLocation = ActiveLevelPieces[ i ].transform.position;
      newLocation.x -= LevelPiecesMoveRate * Time.deltaTime;

      ActiveLevelPieces[ i ].transform.position = newLocation;

      if (ActiveLevelPieces[ i ].transform.position.x <
transform.position.x)
      {
        if (ActiveLevelPieces[ i ] == StartingLevelPiece)
        {
          ActiveLevelPieces[ i ].gameObject.SetActive( false );
        }

        ActiveLevelPieces[ i ].transform.position = ActiveLevelPieces[ i
].GetInitialLocation( );
        ActiveLevelPieces[ i ] = GetRandomLevelPiece( );
        ActiveLevelPieces[ i ].transform.position = FindOtherLevelPiece(
ActiveLevelPieces[ i ] ).gameObject.transform.FindChild( "EndLocation"
).position;
      }
    }
  }
```

As mentioned before, the `Update` function is called for every frame that is rendered. This means that if your game is running at *thirty* frames per second, the Update function gets called *thirty* times a second as well.

As we will always have *two* `ActiveLevelPieces` and want them to move together, we will use a `for` loop that will loop through these `ActiveLevelPieces`, which if you recall is *two*.

For each one of these `ActiveLevelPieces`, we first want to get their location, which we named `newLocation`.

We then want to change the `location.x` value of this `newLocation` by `LevelPieceMoveRate` multiplied by the `Time.deltaTime` value, which is the time it took to render the last frame to this frame.

When we have `newLocation` assigned, we then want to update the location of the current `ActiveLevelPiece` element to the `newLocation Vector3` value, which will move the

`ActiveLevelPiece` element.

We then want to check whether the `ActiveLevelPiece` element has passed the bounds of the **LevelPieceManager** GameObject `position.x` value. If it has, we check whether the `ActiveLevelPiece` element is `StartingLevelPiece`. If it is, we know that we don't want to use it again, so we will use the `SetActive` function to hide it by saying that it is *not* active anymore.

As the `ActiveLevelPiece` element has reached the point where it no longer needs to be used, we will reset its location to its `InitialLocation` with the `GetInitialLocation` function written in the `LevelPiece` class.

After it has been reset, we will use the `GetRandomLevelPiece` function so that we can begin to use an available LevelPiece. As mentioned before, this function returns a **LevelPiece** that is currently not being used.

After we have our next **LevelPiece**, we then want to set its location to the `EndLocation` of the *other* `ActiveLevelPiece`, which is done with a function named `FindOtherLevelPiece`, in which we pass the current `ActiveLevelPiece` to compare against the rest of them, making sure that we don't accidentally return the `ActiveLevelPiece` that needs replacement.

Let's simplify what is going on here. We are taking a location of GameObject and offsetting it a small amount and then assigning the GameObject position to the new offset location. If this offset location is outside the bounds of what we have set, we can replace it with a new GameObject that begins the cycle all over again.

Next, we need the `FindOtherLevelPiece` function. Add the following function under `Update`:

```
// Get the other LevelPiece
// from the LevelPieces
// Array
private LevelPiece FindOtherLevelPiece(LevelPiece CurrentLevelPiece)
{
  for (int i = 0; i < ActiveLevelPieces.Length; i++)
  {
    if (ActiveLevelPieces[ i ] != CurrentLevelPiece)
    {
      return ActiveLevelPieces[ i ];
    }
  }
  return null;
}
```

This function is simple. We check the value of `CurrentLevelPiece` against the array elements of `ActiveLevelPieces`. As soon as we find that the `ActiveLevelPieces` array element is *not* equal to `CurrentLevelPiece`, we will return it. This means that we found the other `ActiveLevelPieces` array element (as there are only *two*). Also, it doesn't match the `CurrentLevelPiece` argument.

Next, we need to write the `GetRandomLevelPiece` function. Add the following function

under `FindOtherLevelPiece`:

```
// Get random level piece
// from LevelPieces Array
private LevelPiece GetRandomLevelPiece()
{
  LevelPiece returnPiece = null;
  while (returnPiece == null)
  {
    for (int i = 0; i < LevelPieces.Length; i++)
    {
      if ( !isActivePiece( LevelPieces[ i ] ) )
      {
        returnPiece = LevelPieces[ i ];
      }
    }
  }
  return returnPiece;
}
```

As mentioned before, the `while` loop will continue to loop until the condition is no longer `true`. In our case, we want to make sure that the loop stops when `returnPiece` has a value or is *not* equal to null.

For this, let's first create a `for` loop that loops through `LevelPieces`. These are the **LevelPiece** prefab GameObjects in our scene. We then check whether the `LevelPieces` array element is *not* active by passing it to a function called `isActivePiece` and then checking whether this function returns `false`.

When we have the `LevelPieces` array element that is *not* being used, we can assign its `returnPiece` to that `LevelPieces` array element. Once this is done, the `while` loop condition will become `false` and then the next line after the `while` loop returns the `returnPiece` found.

Simply put, we will look at all the `LevelPieces` and return the first one that is *not* being used.

The last function we need to write is the `isAlreadyActive` function. This will return *true* if the piece is active and *false* if it is not active. Under the `GetRandomLevelPiece` function, add the `isActivePiece` function:

```
// Check if LevelPiece
// is already used.
private bool isActivePiece(LevelPiece Piece)
{
  for (int i = 0; i < ActiveLevelPieces.Length; i++)
  {
    if (Piece == ActiveLevelPieces[ i ])
    {
      return true;
    }
  }
  return false;
}
```

Similar to the `FindOtherLevelPiece` function, we loop through the `ActiveLevelPieces` array and check whether the `Piece` argument is passed. If any of the `ActiveLevelPieces` array elements match the `Piece` argument being passed, the function will return *true* because the condition is *true*. If the function loops through the entire `ActiveLevelPieces` array and the `Piece` argument doesn't match any of them, the function will return *false* because the condition in the `for` loop was never *true*.

This is the `LevelPieceManager` class. If you save and open/tab back over to Unity and then compile, you can set up the **LevelPieceManager** settings.

# Setting up the LevelPieceManager GameObject

If not already created, right-click on the **Hierarchy** tab and select **Create Empty**. Rename this GameObject as `LevelPieceManager`.

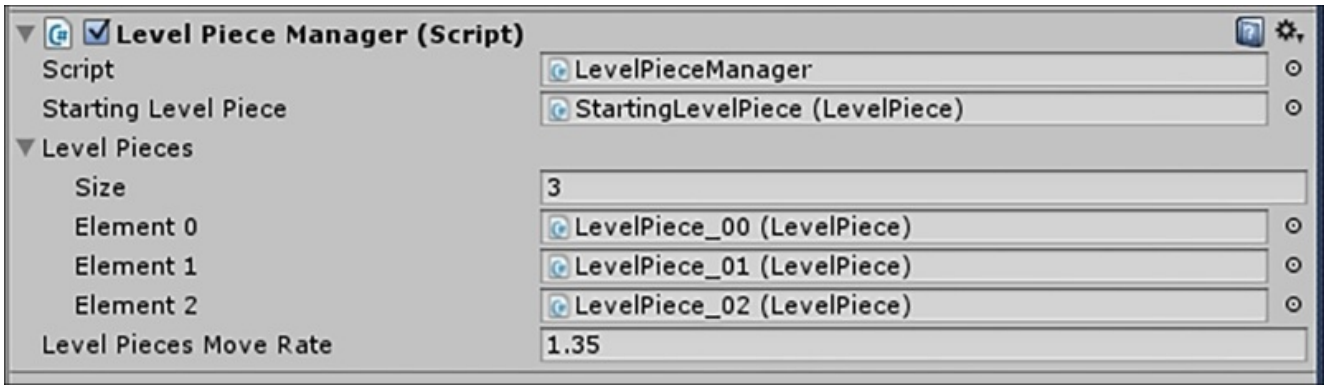
With the **LevelPieceManager** GameObject selected, click on **Add Component** in the **Inspector** window.

Search for **LevelPieceManager** and select **LevelPieceManager** to add the component.

With the **LevelPieceManager** GameObject still selected, change its **X** position to **-7.75** using the **Inspector | Transform** window. The **Y** and **Z** locations don't matter, although you can set them both to *zero* if you want to keep things organized.

Perform the following steps:

1. Take each of the **LevelPiece** prefabs and drag them onto the scene outside the view of the camera, except for **StartingLevelPiece**. Move the **StartingLevelPiece** position to `0X`, `0Y`, and `0Z` by selecting it from **Hierarchy** and using the **Inspector | Transform** settings.
2. Select the **LevelPieceManager** GameObject from **Hierarchy**.
3. Then, left-click and drag **StartingLevelPiece** from **Hierarchy** onto the **Starting Level Piece** slot in the **LevelPieceManager** GameObject.
4. Now, change the **Level Pieces** array amount from *zero* to *three*.
5. Then, take each of the **LevelPiece** prefab GameObjects, *not* including **StartingLevelPiece**, and put each of them in a slot of the `LevelPieces` of the **LevelPieceManager** GameObject.
6. Finally, change **Level Piece Move Rate** from `0` to `1.35`:




If you play the game now, you should see `LevelPieces` moving and connecting to each other, simulating that the player character is moving. You will also see that at times, the player character is getting pushed backwards by the slope of `LevelPieces`. To fix this, open the `Character` script from the `Assets/Scripts` folder.

In the `Update` function, before the condition that checks `isFadeOut`, add the following code:

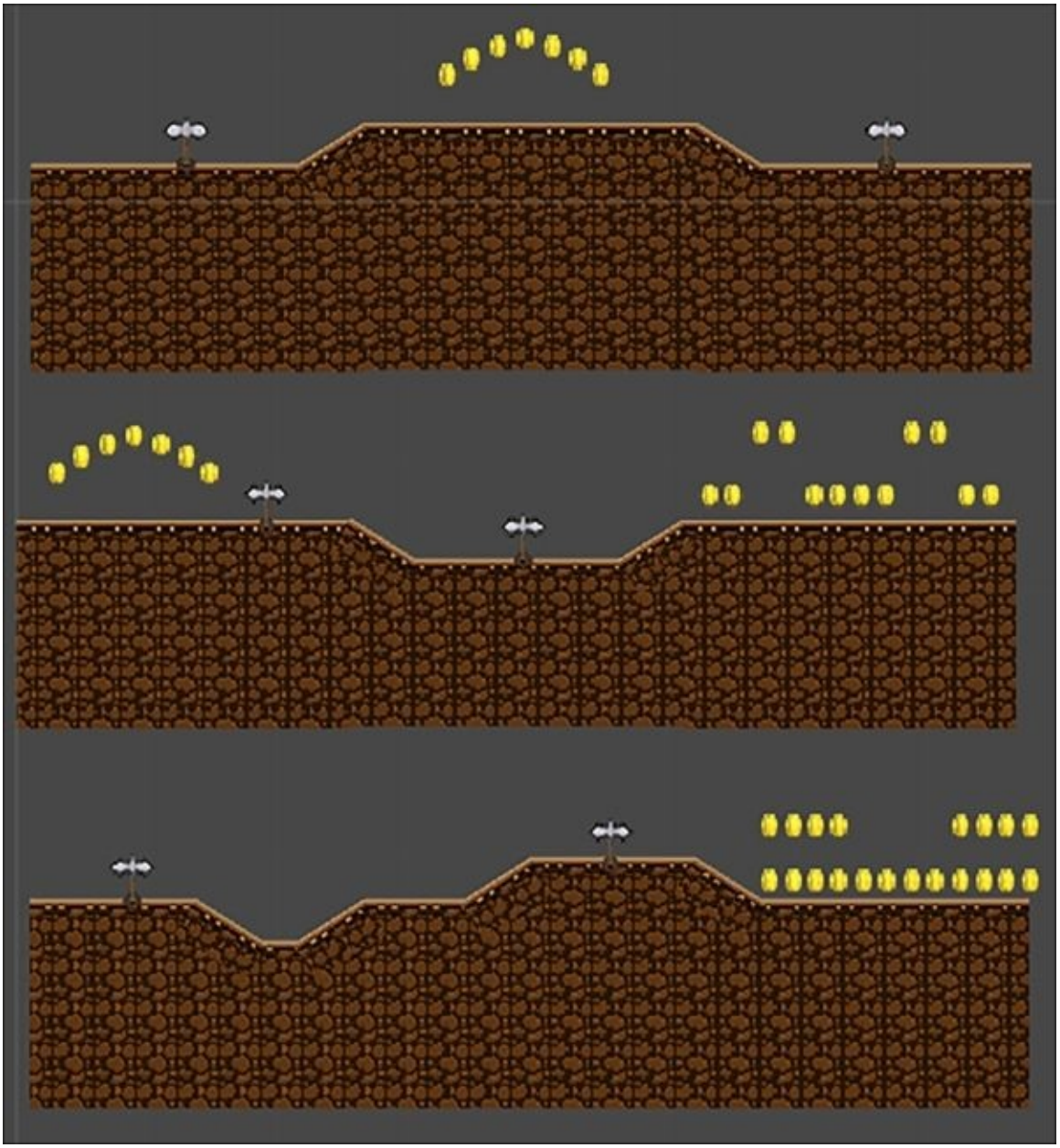
```
    // Update is called once per frame
    void Update ()
    {
      Vector3 lockXPosition = transform.position;
      lockXPosition.x = 2.25f;
      transform.position = lockXPosition;

      if (isFadeOut)
      {
```

If you play the game now, you will see that the character is locked in the **X** position of `2.5`, meaning that the slopes won't affect it anymore.

# Adding coins and obstacles to LevelPieces

The next step is to add coins and obstacles to our **LevelPiece** prefabs. As our `LevelPieceManager` can handle more than *three* **LevelPiece** prefabs, you can create duplicates to have different coin and obstacle layouts. Alternatively, you can keep it simple and only use the current *three* **LevelPiece** prefabs.

Drag and drop the **Coin** and **Obstacle** prefabs onto the scene and position them around the **LevelPiece** prefabs. For each of the **LevelPiece** prefabs, create as many obstacles or coins as you want, and after they are placed, make sure to drag them onto the **LevelPiece** GameObject, making the obstacles and coins children of **LevelPiece**. When you are done, you should have something similar to the following screenshot:

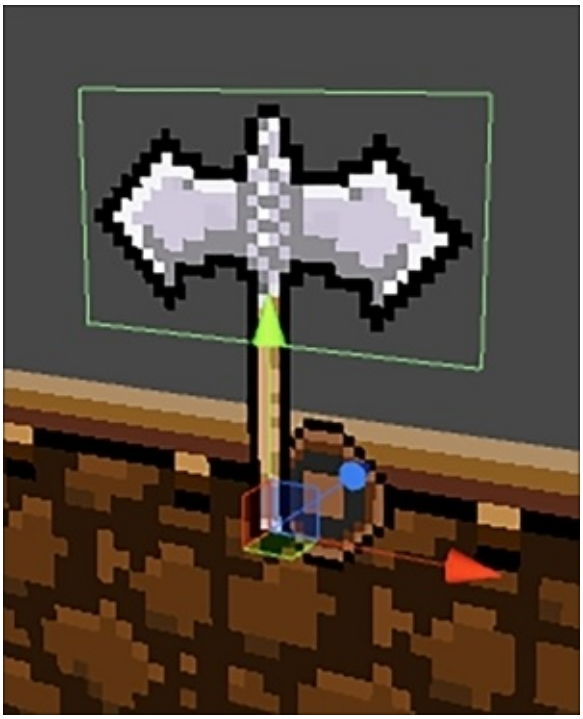


If your **Axe** obstacle is rendering behind **LevelPiece**, left-click on the 2D button at the top of the **Scene** window.

You can use the following controls to navigate the scene:

- **Alt + left click**: This control can be used to rotate around the center of the scene
- **Alt + middle mouse down**: This control can be used to move the camera around
- **Middle mouse wheel forward**: This control can be used to zoom in the camera
- **Middle mouse wheel backward**: This control can be used to zoom out the camera
- **F**: With the selected GameObject, the camera positions itself in front of the selected GameObject

Select the **Axe** GameObject that needs to be moved and use the move gizmo to move it in front of **LevelPiece**, as shown in the following screenshot:



## Tip

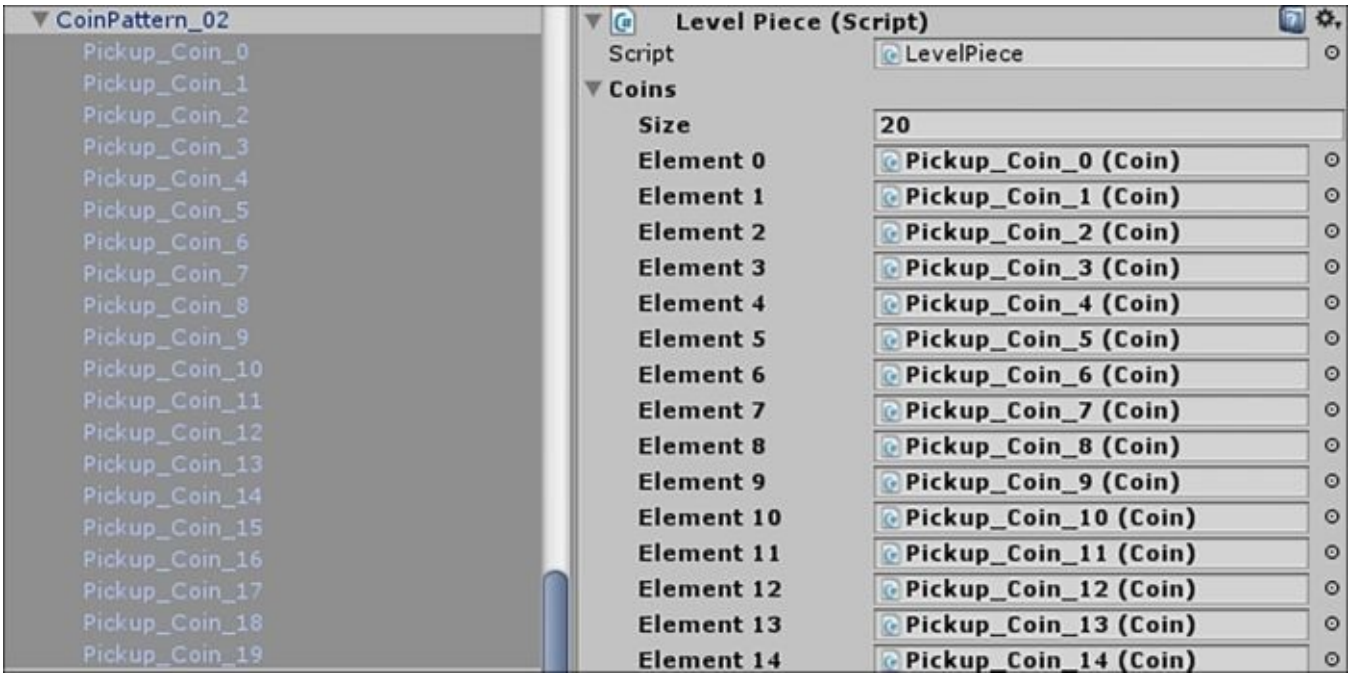
The **Z** position of GameObjects does not affect the game. You can position them in any order to make them render correctly in the scene. You can move anything forward or backward to make the sprites look correct.

# Managing coins and obstacles with the LevelPiece code

In the `Assets/Scripts` folder, double-click on the `LevelPiece.cs` file to open it. Add the following code at the top of the class:

```
// Coin children
public Coin[] Coins;
```

Save the `LevelPiece.cs` file and tab back or open Unity to let the code compile. Start by selecting one of the **LevelPiece** GameObjects in **Hierarchy**, and in the top-right corner of the **Inspector** window, click on the small lock; it should change from an open lock to a closed lock. Once it has, you can select all the coins from **Hierarchy** for the specific **LevelPiece** GameObject and drag them onto the **LevelPiece** component in the **Inspector** window; it will automatically fill and assign the **Coins** array, as shown in the following screenshot:




Do this for all the coins in all the **LevelPiece** GameObjects. Also, make sure to match them to the correct `Coins` array. Remember that the lock will keep the **LevelPiece** GameObject selected until you unlock it by clicking on the lock again. You will have to do this for each **LevelPiece**.

## Resetting coins

In the `Assets/Scripts` folder, open the `Coin.cs` class. We will change how the coin is being used here. Earlier, we were destroying it after the player collided with it. We do not want this now because this was only useful for testing purposes. We now want the coin to be in the scene always and managed in a way where it can be reactivated when **LevelPiece** becomes active.

In the `OnTriggerEnter2D` function, remove the following line:

```
Destroy( gameObject );
```

Then, replace it with the following code:

```
ActivateCoin( false );
```

Now, we need to write the `ActivateCoin` function. Add the following code under `OntriggerEnter2D`:

```
    // Activates or deactivates
    // coin
    public void ActivateCoin(bool bActivate)
    {
      SpriteRenderer renderer;
      BoxCollider2D collider;

      renderer = gameObject.GetComponent<SpriteRenderer>( );
      collider = gameObject.GetComponent<BoxCollider2D>( );

      collider.enabled = bActivate;
      renderer.enabled = bActivate;
    }
```

We first have the **SpriteRenderer** and **Box Collider 2D** components for the coin and then set them to the value of `bActivate`. This means that if the character collects the coin, the bActivate value will be false. This is because we don't want it activated after being collected, which will set enabled to false for both the collider and the renderer. Whereas, if we reset the game, we can call this same function but set bActivate to true. This will turn enabled to true for both the collider and renderer, making it so that the character can collect it again.

Furthermore, this function allows you to control the coin after it has been used so that it can be used again by setting the **Box Collider 2D** and **SpriteRenderer** components to enabled or not enabled. By setting these two components to not enabled, they won't react to the player character anymore, but if we set them to enabled, the player can again interact with them. This is good for us because we need to enable them every time **LevelPiece** gets used, although it has been used before.

Save this `Coin.cs` file.

## Resetting the coin in LevelPiece

In the `Assets/Scripts` folder, open the `LevelPiece` class. At the bottom of the class, add the following function:

```
    // Resets all children
    // coins
    public void ResetAllChildrenCoins()
    {
      for (int i = 0; i < Coins.Length; i++)
      {
        Coins[ i ].ActivateCoin( true );
```

```
      }
    }
```

This does exactly as the function describes. It loops through all the coins that we added from the **Hierarchy** tab to the `LevelPiece` script component and calls the `ActivateCoin` function we wrote early, setting the enabled bool value to `true`. This is what we will pass in `ActivateCoin`.

Then, we need to call the `ResetAllChildrenCoins` function from `LevelPieceManager`, when it decides that it needs a new **LevelPiece**. In the `Assets/Scripts` folder, open the `LevelPieceManager` code file.

Change the `Update` function as follows:

```
  // Update is called once per frame
  void Update ( )
  {
    for (int i = 0; i < ActiveLevelPieces.Length; i++)
    {
      Vector3 newLocation = ActiveLevelPieces[ i ].transform.position;
      newLocation.x -= LevelPiecesMoveRate * Time.deltaTime;

      ActiveLevelPieces[ i ].transform.position = newLocation;

      if (ActiveLevelPieces[ i ].transform.position.x <
transform.position.x)
      {
        if (ActiveLevelPieces[ i ] == StartingLevelPiece)
        {
          ActiveLevelPieces[ i ].gameObject.SetActive( false );
        }

        ActiveLevelPieces[ i ].transform.position = ActiveLevelPieces[ i
].GetInitialLocation( );
        ActiveLevelPieces[ i ] = GetRandomLevelPiece( );
        ActiveLevelPieces[ i ].transform.position = FindOtherLevelPiece(
ActiveLevelPieces[ i ] ).gameObject.transform.FindChild( "EndLocation"
).position;
        ActiveLevelPieces[ i ].ResetAllChildrenCoins( );
      }
    }
  }
```

The only change here is the last line. When we have a new `ActiveLevelPiece` and when we update its location, we call `ResetAllChildrenCoins` for the `ActiveLevelPiece` so that the `ActiveLevelPiece` is fully playable again as all the coins are back to being able to be collected.

Save the `LevelPieceManager.cs` file and go back to Unity. After the code has compiled, you should see your entire gameplay going, including all the `LevelPieces` moving, connecting, and resetting.

## Tip

Make sure to save all of your Unity files and the scene.

# Summary

In this chapter, we discussed how to create a Unity editor tool and a grid of GameObjects. These were laid out by the size of the sprites you chose and were flexible enough to use with your own settings. We also created prefabs for all of our bigger GameObjects, which could hold all of their components in a neat package. Finally, we wrapped up the gameplay by having the `LevelPiece` and `LevelPieceManager` code move, piece together, and reset all the active GameObjects, so the player can't even tell what is going on.

In the next chapter, we will explore the Unity UI system and start to piece together the UI so that the player can see the score of the game and also handle what happens when the player dies. We will also finish the final portions of the game level by having an animated background to give the game a more diverse feel.

# Chapter 5. Scene Background, the Game User Interface, and the Save Profile

In order to complete the experience of playing the game, we need to finish the background for the scene and use the Unity 5 UI system to show the players what is going on with the game, as well as allow them to restart the game when they die.

The background is going to be made up of four images: a static (nonmoving) background, clouds that will hover, and two sets of mountains that will move to make the player movement seem more real.

The Unity 5 UI system is made up of a panel object that will hold children, such as text and buttons, to display what is currently going on in the game to the user. For the game, we will display a unit number that represents how far the player has moved and keep a tally of how many coins the player has picked up. We will also display a restart button the user can tap when they die, restarting the game loop.

In this chapter, we will cover the following topics:

- Writing a C# script that will move the background image layers to make the character look as if it is moving more realistically through the game world
- Creating the game user interface with the Unity 5 UI system that will display the distance traveled by the character and how many coins they have collected
- Using the UI system, create a button that will display when the player dies, allowing them to restart playing

# Creating the scene background

To start with, we will need a C# script.

Perform the following steps:

1. First, search the `Assets/Scripts` folder and right-click on it.
2. Then, click on **Create** and select **C# Script**.
3. Now, name this `SceneBackground`.
4. Finally, double-click on the `SceneBackground` file to open it.

The `SceneBackground` file will hold *four* sprites. Each of these sprites will have their own movement speed and movement distance (speed being how fast they move and distance is how far forward and backward they can move).

Add the following code to the `SceneBackground` file before the `Start` function:

```
// Struct to hold data for Background Element
[System.Serializable]
public struct BackgroundElement
{
  public Sprite BackgroundSprite;
  public float MovementSpeed;
  public float MovementDistance;
  public Vector3 SpriteLocation;
  public Vector3 SpriteScale;

  [System.NonSerialized]
  public Vector3 ObjectStartLocation;
  [System.NonSerialized]
  public GameObject ChildObject;
  [System.NonSerialized]
  public bool bMoveBackwards;
};

// Array to store backgrounds for scene
public BackgroundElement[ ] SceneBackgroundElement;
```

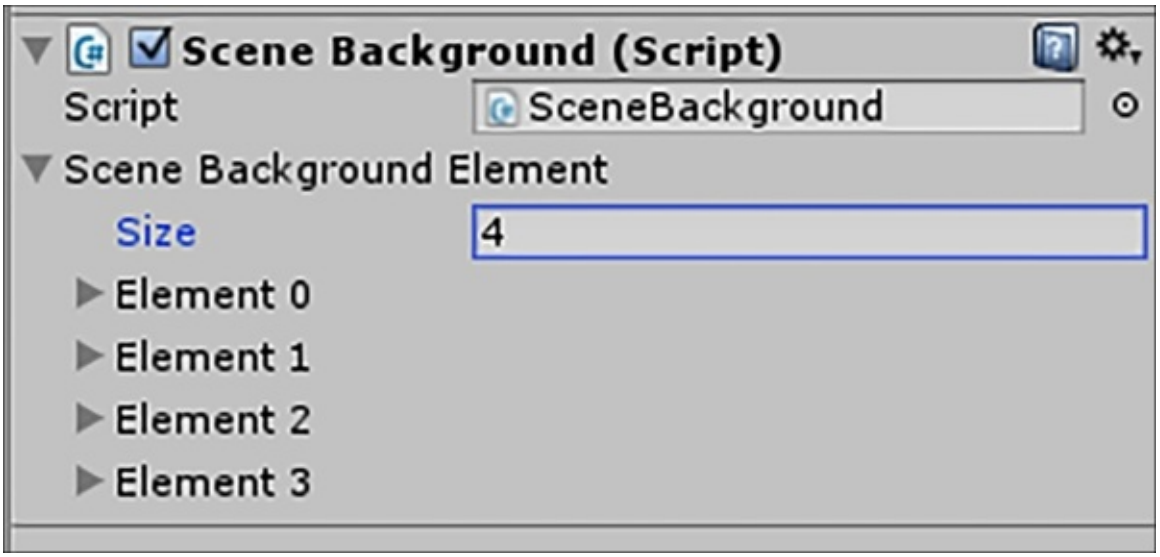
The `BackgroundElement` struct is a way for us to create a sprite and manage its `MovementSpeed` and `MovementDirection` without having to create individual variables for all of them. This way, we can compartmentalize the data into a single struct object and assign the values we need with less code clutter. Then, with the `SceneBackgroundElement` array, we can create and assign the background images with unique settings to handle them correctly.

`[System.Serializable]` allows you to edit these variables in the editor. Without this, the variables will not be shown in the **Inspector** window when you select the GameObject from the **Hierarchy** or **Project** folder(s).

Save the `SceneBackground` code file and go back to Unity. Right-click on **Hierarchy** and then select **Create Empty**. Name this `SceneBackground`.

In the **Inspector** window, with the new **SceneBackground** GameObject selected in **Hierarchy**, click on **Add Component**. Search for `SceneBackground` and select **SceneBackground**.

You should see a drop-down list for **Scene Background Element** with *zero* elements. Set the number of elements to *four*. You should now see the empty elements for **Scene Background Element**:




In order to set all of these, we need to import the art for them. In the `Assets` folder, right-click and select **Create** and then **Folder**. Name this `Background`.

Open the `Background` folder and right-click on it. Then, click on **Import New Asset…**. Navigate to the book art files and find the folder named `ChapterFive_Background`. Import all the *four* images.

This includes:

- `Background_00`
- `Background_01`
- `Background_02`
- `Background_03`

Select all the four images in the `Assets/Background` folder. Then, in the **Inspector** window, perform the following steps:

1. Set **FilterMode** to **Trilinear**.
2. Set **MaxSize** to `1024`.
3. Set **Format** to **Truecolor**.
4. Then, click on **Apply**.

With the **SceneBackground** GameObject selected in **Hierarchy**, click on the drop-down list for the **0** element of the **SceneBackground** script component.

In the **Sprite object** slot, click on the small circle with a dot. When the window opens,

select the sprite object, search for `Background_00` and then select `Background_00`. Change the Sprite Location to `0X`, `0Y`, and `0Z`; then, set the `Sprite Scale` to `1X`, `1Y`, and `1Z`. Leave **Movement Speed** and **Movement Distance** as *zero*.

Next, we want to set the second element: clouds. Again, with the **SceneBackground** GameObject selected, take at look at the **Inspector** tab and click on the **1**or **2** element drop-down list so that you change the settings as follows:

1. Set **Background Sprite** to `Background_01`.
2. Set **Movement Speed** to `0`.
3. Set **Movement Distance** as `0`.
4. Set **Sprite Location** for **X** to `5.65`, **Y** to `1.95`, and **Z** to `40`.
5. Set **Sprite Scale** for **X** to `0.2`, **Y** to `0.2`, and **Z** to `0.2`.

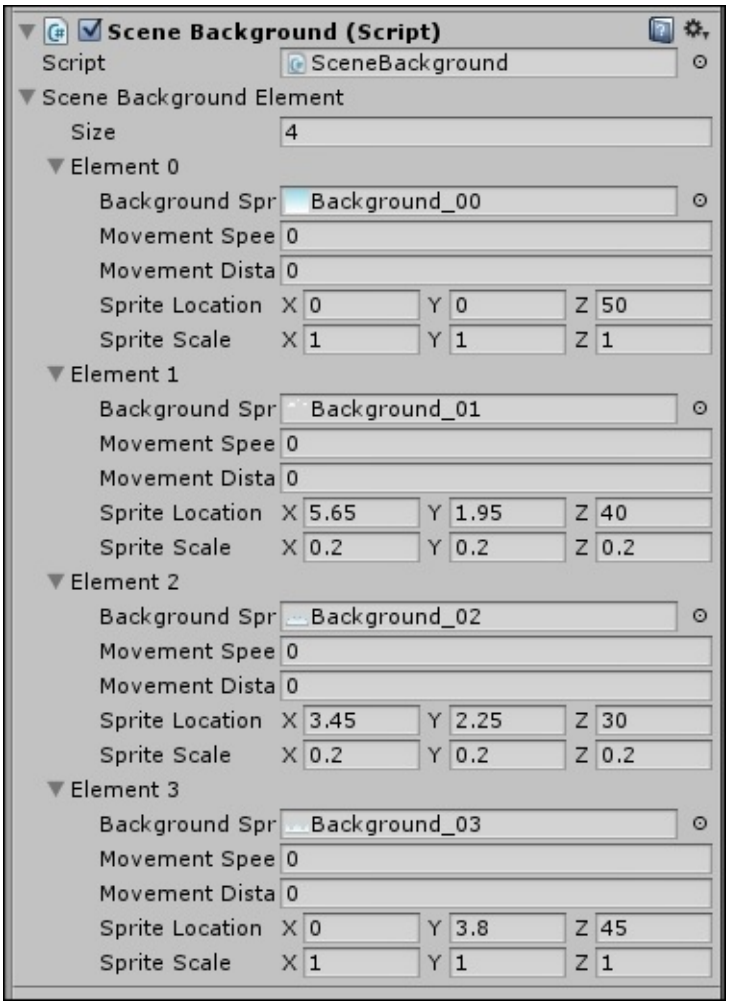
Next element is smaller mountains:

1. Set **Background Sprite** to `Background_02`.
2. Set **Movement Speed** to `0`.
3. Set **Movement Distance** to `0`.
4. Set **Sprite Location** for **X** to `3.45`, **Y** to `2.25`, and **Z** to `30`.
5. Set **Sprite Scale** for **X** to `0.2`, **Y** to `0.2`, and **Z** to `0.2`.

Finally, for the bigger mountains:

1. Set **Background Sprite** to `Background_03`.
2. Set **Movement Speed** to `0`.
3. Set **Movement Distance** to `0`.
4. Set **Sprite Location** for **X** to `0`, **Y** to `3.8`, and **Z** to `45`.
5. Set **Sprite Scale** for **X** to `1`, **Y** `1`, and **Z** to `1`:

The following is screenshot showing the settings for all the elements:




We will update the **Movement Speed** and **Movement Distance** values after we write the code that handles these values.

For now, let's write the code that will create and place the GameObjects and sprites based on the settings we just created. In the `Start` function of the `SceneBackground` class, write the following code:

```
  // Use this for initialization
  void Start ( )
  {
    SpriteRenderer backgroundRenderer;
    GameObject backgroundObj;

    for (int i = 0; i < SceneBackgroundElement.Length; i++)
    {
      // new GameObject
      backgroundObj = new GameObject( );

      // Parent it, set its position, and scale
      backgroundObj.transform.SetParent( transform );
      backgroundObj.transform.position =
SceneBackgroundElement[i].SpriteLocation;
      backgroundObj.transform.localScale =
SceneBackgroundElement[i].SpriteScale;
```

```
      // Keep reference of new object and its start location
      SceneBackgroundElement[ i ].ChildObject = backgroundObj;
      SceneBackgroundElement[ i ].ObjectStartLocation =
backgroundObj.transform.position;

      // Create Renderer and set Sprite
      backgroundRenderer = backgroundObj.AddComponent<SpriteRenderer>();
      backgroundRenderer.sprite = SceneBackgroundElement[ i
].BackgroundSprite;
    }
  }
```

As we only have values to use and are not using any direct reference to other GameObjects in the scene for these, we have to create them. To do this, we need a SpriteRenderer and a GameObject to hold it.

We know by the number of elements created by us (or if you were on a bigger team, the designer) in the GameObject `SceneBackground` script component that we have to create a certain number of GameObjects to hold the `BackgroundSprite` for each of the `SceneBackgroundElement` objects. In order to create all of them efficiently, we will use a `for` loop that will loop the number of created `SceneBackgroundElement` objects.

In each loop, we can create a new GameObject and set its parent using the `SetParent` function to transform the GameObject that holds this `SceneBackground` script component. We will then set its local position and scale, relative to the parent transform location, to the settings we entered. Then, we will store the new GameObject that was created to the `SceneBackgroundElement` of that loop and its location.

After this, add a `SpriteRenderer` component to the newly created GameObject and assign the sprite from the settings we entered to the sprite for `SpriteRenderer`.

If you run the game now, you should see a background with some mountains. The clouds will be out of the field of view of the camera. We will fix this by having them move across the screen.

# Adding movement to SceneBackgroundElement

As we want some movement to `SceneBackgroundObjects`, we need the code to manage it. In the `SceneBackground` class file, change the `Update` function, as shown in the following code:

```
    // Update is called once per frame
    void Update ( )
    {
      for (int i = 0; i < SceneBackgroundElement.Length; i++)
      {
        // Does the SceneBackgroundElement move
        if (SceneBackgroundElement[i].MovementSpeed > 0)
        {
          // Get current location and offset it by movement speed, based on
bMoveBackwards
          Vector3 newSceneBackgroundLoc = SceneBackgroundElement[ i
].ChildObject.transform.position;
          newSceneBackgroundLoc.x +=
(SceneBackgroundElement[i].bMoveBackwards)
          ? -SceneBackgroundElement[i].MovementSpeed * Time.deltaTime
          : SceneBackgroundElement[i].MovementSpeed * Time.deltaTime;

          // Set location based on movement speed offset
          SceneBackgroundElement[ i ].ChildObject.transform.position =
newSceneBackgroundLoc;

          // If ChildObject has moved farther than move distance, have it
move the other way.
          if
(Vector3.Distance(SceneBackgroundElement[i].ChildObject.transform.position,
SceneBackgroundElement[i].ObjectStartLocation) >=
          SceneBackgroundElement[i].MovementDistance)
          {
            // Change the direction of the SceneBackgroundElement
            SceneBackgroundElement[ i ].bMoveBackwards =
!SceneBackgroundElement[ i ].bMoveBackwards;
          }

        }
      }
    }
```

We begin the `Update` function with a `for` loop. The `for` loop will loop through the `SceneBackgroundElement` struct variables that we created earlier. For each one of the struct elements, we will start by checking whether the `MoveSpeed` elements is greater than *zero* because we do not want to try to move something if it isn't set to move.

Next, we will create a new `Vector3` local variable within the scope of the `Update` function named `newSceneBackgroundLoc` and assign it to the `ChildObject` variable of the element of `SceneBackgroundElement` represented by `i`. This is the value represented by the current loop.

Once we have the position value of the `ChildObject` variable of the current

`SceneBackgroundElement` variable, we can use the *ternary operator* to decide which direction the `ChildObject` variable needs to be moving in. In addition, it has a rule that if you add a negative value to a positive value, it is subtracted. Then, we can use the `+=` operator to assign a negative value to the offset of `newSceneBackgroundLoc.x` if the `bMoveBackward` bool value is `true`. If it is *not* `true`, we can then assign a positive value because positive plus positive is still positive. We multiply the negative or positive `MovementSpeed` value by `Time.deltaTime`, making the transition between movements a bit smoother.

We can then assign the `SceneBackgroundObject[ i ].ChildObject` position to the `newSceneBackgroundLoc` offset position, moving the `ChildObject` variable to whatever the `MovementSpeed` value is set to.

Once the `ChildObject` variable has been moved, we need to check whether the current distance from the start location is greater than the `MovementDistance` value of the current `SceneBackgroundObject`. If it is, we flip the `bMoveBackwards` value, making the previous ternary operator begin calculating the `newSceneBackgroundLoc` value in the opposite direction.

Let's go back to the Unity editor and change the movement values for the `SceneBackgroundElements` as follows:

In **Element0**:

- Set **Movement Speed** to `0.2`
- Set **Movement Distance** to `2`

In **Element1**:

- Set **Movement Speed** to `1`
- Set **Movement Distance** to `6`

In **Element2**:

- Set **Movement Speed** to `0.2`
- Set **Movement Distance** to `4`

In **Element3**:

- Set **Movement Speed** to `0.2`
- Set **Movement Distance** to `4`

If you play the game now, you will see a lot more movement with the `SceneBackgroundElements`. Feel free to adjust these values to anything you want. The values are literal, meaning the larger the number, the farther/faster the object will change.

# Creating the game user interface

To begin with the user interface for our game, we need a class that manages when and how to use UI elements, such as buttons and when to show the UI screen.

# The GameInfo class

In order for our other code classes to get a reference to the `GameInfo` class, we can create a `GameObject` in the scene that holds a GameInfo script component. To do this, perform the following steps:

1. In the `Assets/Scripts` folder, right-click and select **Create** and then click on **C# Script**. Name this `GameInfo`.
2. Double-click on the `GameInfo` script and select the `Start` function as `Awake`. Save this file and go back to Unity.
3. Right-click on **Hierarchy** and select **Create Empty**. Name this new GameObject `GameInfo`.
4. With this selected, search for **Inspector** and click on **Add Component**. Then, search for **GameInfo** and select **GameInfo**.

# Importing UI images

We now need to import some PNG files to represent elements of the UI. These PNG images will represent elements such as buttons. To do this, perform the following steps:

1. Let's go back to Unity, navigate to the `Assets` folder, right-click and select **Create** and then select **Folder**. Name this folder `UI`.
2. Right-click on the `UI` folder and select **Import New Asset…**. Navigate to the book `Art` files and find the folder named `ChapterFive_UI`. Import the PNG files to the `UI` folder.
3. Select the UI art files. Then, in **Inspector**, change the image settings as follows:
    1. Set **FilterMode** to **Trilinear**.
    2. Set **MaxSize** to `256`.
    3. Set **Format** to **Truecolor**.
    4. Then, click on **Apply**.

## Tip

You can select more than one file in the `Project` folder by left-clicking on one and holding down *Shift* to select more than one range. For example, select the top file in the folder and press *Shift* + left-click on the bottom file. This will select all of these files between the top and bottom.

## Creating the UI Canvas

Right-click on the **Hierarchy** tab. Then, click on **UI** and select **Canvas**. Name this new GameObject `GameUI`.

With the **GameUI** GameObject selected, take a look at **Inspector**, Then, in the **Canvas Scaler** component, select **UI Scale Mode** as **Scale With Screen Size**. Also, select **Reference Resolution** as **X1024 Y768**. This will keep the UI in the same ratio, irrespective of the screen size of the device used, such as iPhone versus iPad.

### The coin background

Right-click on the **GameUI** GameObject. Then, click on **UI** and select **Image**. Name this image `CoinBackground`.

In the image component of the **CoinBackground** GameObject, click on the small circle next to the object field named **Source Image**. Search for `ButtonBackground` and select, **ButtonBackground**.

With the **CoinBackground** GameObject selected, as **Inspector**, change the following settings:

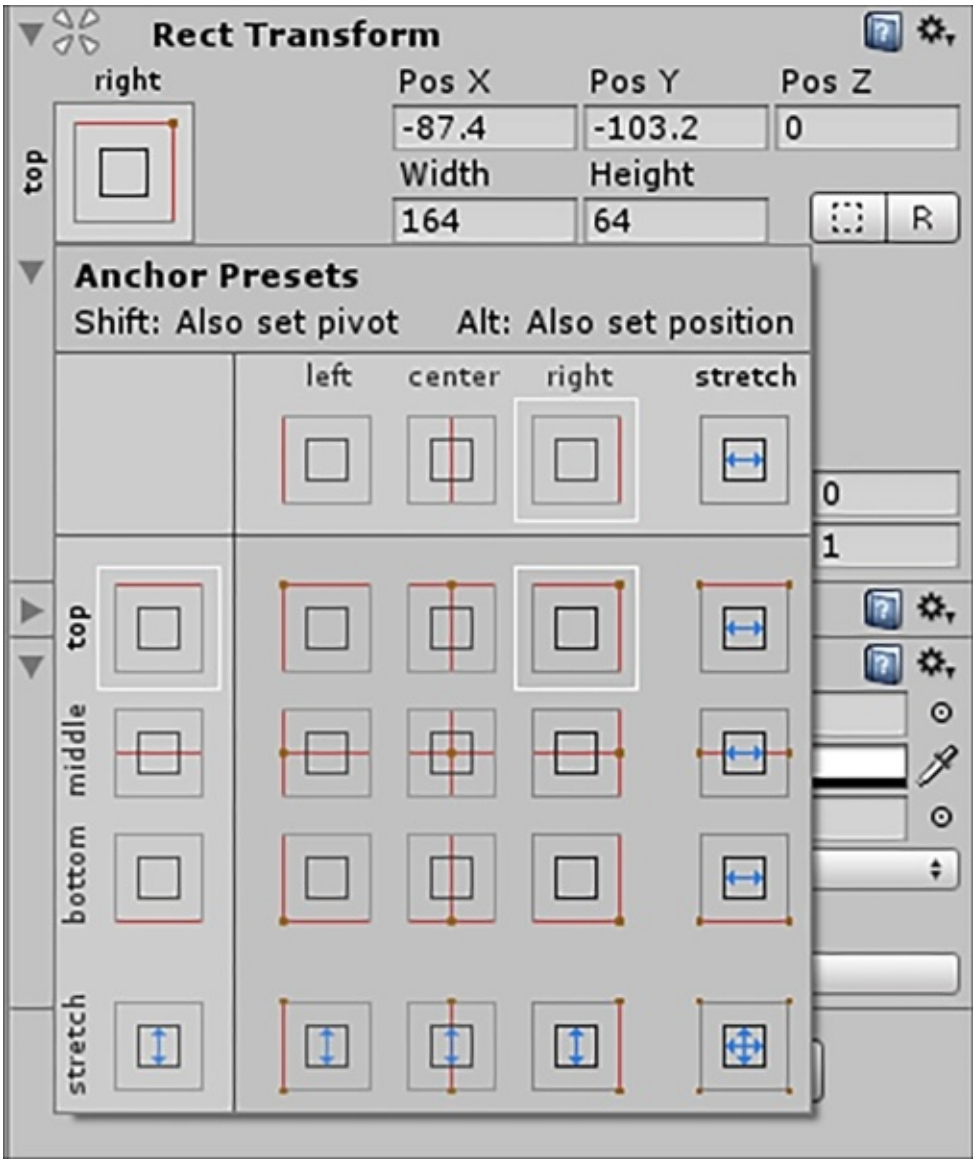
1. Set **Pos X** to `-87.4`.
2. Set **Pos Y** to `-103.2`.
3. Set **Pos Z** to `0`.

4. Set **Width** to `164`.
5. Set **Height** to `64`.

## Tip

If I do not list a change for any setting, leave it as default. This will be `true` for all the settings for the UI.

In the top-left corner of the **Inspector** tab, there is a box with red lines for two of its sides. Click on it and select the *top right* box selection:




This will **anchor** the image to that side of the UI, making a better reference for it to draw if the screen size is to change. To make it simpler, no matter the screen size, the image will be rendered to the same location of the screen space.

## Tip

**Anchor** is the position on the UI screen that will hold this element of the UI. The element that is anchored to this anchor will follow it, irrespective of the screen size.

With the **CoinBackground** GameObject selected, right-click on it and select **UI**. Then, select **Text** from the list and name it `CoinValue`.

Change the **CoinValue** text settings as follows:

1. Set **Pos X** to `19.6`.
2. Set **Pos Y** to `1.3`.
3. Set **Pos Z** to `0`.
4. Set **Width** to `103.6`.
5. Set **Height** to `41.1`.
6. Set **Text** to `0`.
7. Set **Font** to **Arial**.
8. Set **Font Style** to **Bold**.
9. Set **Font Size** to **36**.
10. Set **Alignment** to **Right**.

## Tip

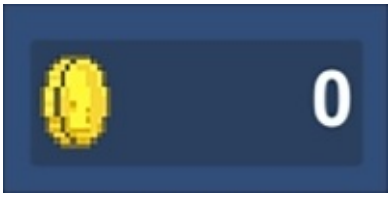
The alignment is set by the small buttons next to it. Select the third button from the left-hand side to set it to **Right**.

With the **CoinValue** GameObject selected, right-click on it and select **UI**. Select **Image** from the list. Name this image `CoinImage`.

Change the **CoinImage** settings as follows:

1. Set **Pos X** to `-73.5`.
2. Set **Pos Y** to `0`.
3. Set **Pos Z** to `0`.
4. Set **Width** to `100`.
5. Set **Height** to `100`.
6. Set **Source Image** to `Pickup_Coin_0`.

You should now see something similar to the following image in the top-right corner of the **Game** tab of the editor:




Before we set up the code to adjust the value in the **CoinValue** text component, we will set up the **DistanceBackground** field.

### The distance background

`DistanceBackground` will reveal the distance the player has run for each attempt and will

reset it to *zero* if they restart.

With the **GameUI** GameObject selected, right-click on it and select **UI**. In the list, select **Image**. Name this image `DistanceBackground`.

With the **DistanceBackground** GameObject selected, search for `Inventory` and change its settings as follows:

- **Pos X**: `-87.39999`
- **Pos Y**: `-37`
- **Pos Z**: `0`
- **Width**: `164`
- **Height**: `64`
- **Source Image**: `ButtonBackground`
- **Anchor**: **top right**

Still, with the **DistanceBackground** GameObject selected, right-click on it and select **UI**. Select **Text** from the list and name it `DistanceValue`.

Change the **DistanceValue** settings using **Inspector** as follows:

- **Pos X**: `0`
- **Pos Y**: `-0.1`
- **Pos Z**: `0`
- **Width**: `141`
- **Height**: `41.6`
- **Anchor**: **top right**
- **Text**: `0`
- **Font**: Arial
- **Font Style**: **Bold**
- **Font Size**: **36**
- **Alignment**: **Right**

You should now see both the distance and the coin UI elements in the top-right corner of the **Game** tab in the editor if you play the game.

### The pause button

Having a pause button is a nice luxury for any game. Unity makes this functionality fairly simple.

Select the **GameUI** GameObject and right-click on it. Navigate to **UI** and select **Button** from the list. Name this button `PauseButton`. Select the child GameObject named **Text** and delete it. We do not need text for `PauseButton`.

With the **PauseButton** GameObject selected in **Inspector**, change its settings as follows:

- **Pos X**: `34.5`
- **Pos Y**: `-34`
- **Pos Z**: `0`
- **Width**: `54`

- **Height**: `54`
- **Anchor**: **top left**
- **Source Image**: `PauseButton`

For **PauseButton** to work, we need to update our `GameInfo` C# class. In the `Assets/Scripts` folder, double-click on the `GameInfo` file and add the following function in it:

```
// Toggle pause, on or off
public void PauseGame( )
{
  Time.timeScale = ( Time.timeScale == 1 ) ? 0 : 1;
}
```

This is a very simple function. We can set `Time.timeScale` to *zero* or the *one* based on whether the current time scale is equal to *one*. If it is, we set it to *zero*. If it is *not*, set it to *one*; we will set it to *one*.

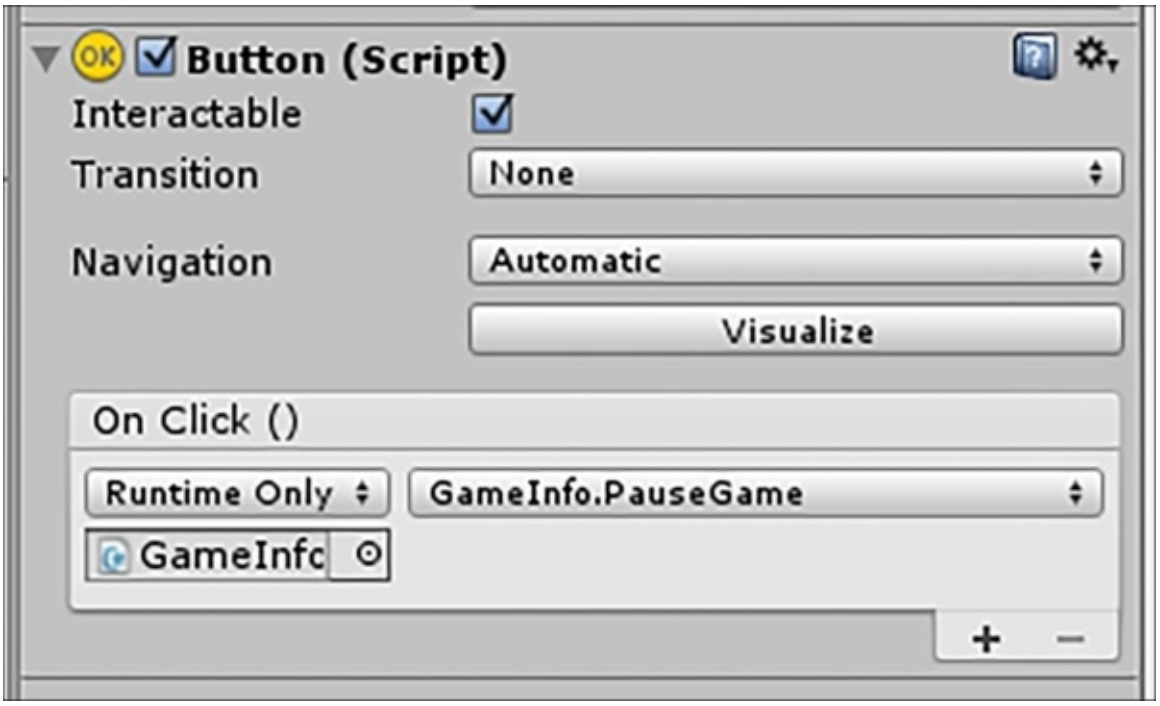
Save the `GameInfo` file and go back to Unity.

In order for the `PauseButton` to work, we need to create a button component and then set the `OnClick` function of the button component to call our `PauseGame` function.

Right-click on **Hierarchy** and select **Create Empty**. Name this new GameObject `GameInfo`. Now, in **Inspector**, click on **New Component**. Then, search for `GameInfo` and select **GameInfo**.

Select the **PauseButton** GameObject in **Hierarchy**. Then, in the **Button** script component in **Inspector**, click on the small plus symbol at the bottom of the **Button** script component. Then, click and drag the **GameInfo** GameObject onto the **On Click ()** object field.

Now, click on the drop-down menu that says **No Function** and navigate to the selection called **GameInfo**. In the list for **GameInfo**, select **PauseGame()**:




If you play the game now, when you click on **PauseButton**, the game will stop moving most things. The `Obstacle` GameObjects will not stop rotating; we can fix this by alternating the `Update` function in the `Obstacle` class.

Open the `Assets/Scripts` folder and double-click on the `Obstacle` class. Change the `Update` function as follows:

```
// Update is called once per frame
void Update ()
{
  if (Time.timeScale == 1)
  {
    Pivot.transform.Rotate(Vector3.forward, RotationSpeed);
  }
}
```

Now, **PauseButton** will freeze some aspects of the game, including the physics, the `FixedUpdate` function, and the `WaitForSecond` class. This means that as we are checking against `Time.timeScale`, before rotating the axe(s) around, it will only rotate if `Time.timeScale` is set to *one*. For example, if `timeScale` is set to *one*, which means that the game is running at normal speed, the axes(s) can rotate. Otherwise, if `timeScale` is *zero*, the axe(s) will stay stationary.

## Updating code to show coins and distance

For the UI to see these values, we need to have a couple of our code classes to tell it what to display.

In the `Assets/Scripts` file, open the `Character` file *under* the line that uses the following code:

```
[System.NonSerialized]
```

```
    public int CoinCount;
```

Add:

```
    public Canvas GameUI;

    [System.NonSerialized]
    public int DistanceCount;

    [System.NonSerialized]
    public float CurrentTime;
```

- `GameUI`: This will reference our **Game User Interface Canvas** GameObject.
- `DistanceCount`: This will represent the distance that the character has moved across the screen so that we can display it to the screen as well as submit any leaderboard records related to the distance traveled. It is not based on the actual movement, rather we will use time to add to it.
- `CurrentTime`: This will be the value we use to add to `DistanceCount`.

We also need to add another `using` tag for the UI to have a reference to the Text class at the top of the `Character` code file under the following code:

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

Now, add the following code:

```
using UnityEngine.UI;
```

In the same `Character` class, navigate to the `Update` function and *under* the lines of the following code:

```
  // Update is called once per frame
  void Update ()
  {
    Vector3 lockXPosition = transform.position;
    lockXPosition.x = 2.25f;
    transform.position = lockXPosition;
```

Add the following code:

```
    if (!isDead && Time.timeScale == 1)
    {
      CurrentTime += Time.deltaTime + 0.025f;
      if (CurrentTime >= 1.0f)
      {
        AddDistance(1);
        CurrentTime = 0.0f;
      }
    }
```

This will make sure that the player is alive by making sure `bDead` is *false* and also make sure that `timeScale` is set to *one*, meaning the game is *not* paused before we change the `CurrentTime` and `CurrentDistance` values.

If these values can be changed (the condition is `true`), we can add `time.DeltaTime +`

`0.025f` to the `CurrentTime` value. The `0.025f` parameter is a small addition to make the `CurrentDistance` count faster than every *one* second.

As soon as `CurrentTime` becomes greater than *one*, we can call the `AddDistance` function and give it a value of *one*. We can also reset `CurrentTime` back to *zero* so that it can begin to count up to *one* again.

Let's add the `AddDistance` function to the `Character` class. At the bottom of the class, under the `ReviveCharacter` function, add the following code:

```
    // Adds to the Distance count and updates the UI
    public void AddDistance(int Additional)
    {
      DistanceCount += Additional;
      if (GameUI != null)
      {
      Text distanceText = GameUI.transform.Find(
"DistanceBackground/DistanceValue" ).GetComponent<Text>( );
      if (distanceText != null)
      {
        distanceText.text = DistanceCount.ToString( );
      }
    }
  }
```

We start this function by adding the value that is passed in to `Additional`.

We then check to make sure that the reference to **GameUI** is not null, making sure **GameUI** has a reference before using it.

When we have a good reference to the **GameUI**, we can create a `Text` local variable and assign it to the **DistanceValue** GameObject text component. This is done by using the **GameUI** reference and using its `transform.Find` function to go through the hierarchy of the **GameUI** GameObject and look for the name of the GameObject passed into the `Find` function. In our case, `DistanceBackground` / `DistanceValue`, meaning it goes to `DistanceBackground` and then looks for `DistanceValue` in its children. When it gets there, we can use `GetComponent` to have the `Text` component from the **DistanceValue** GameObject.

We then check whether the `Text` reference we got from **DistanceValue** is valid. If it is valid, we can update the text of `distanceText` to the `DistanceCount` value. We can do this by using the `ToString` function to convert the `DistanceCount` integer value into a string so that we can assign it to the `distanceText.text` value.

To break this down, we will add the distance, check for a good reference to the **GameUI**, get the **Text** component from one of its children and then change it to the value of **DistanceCount**, showing the player how far they have moved.

We also need to write a function called `AddCoins` for the coins. In the `Character` class, write the following function *under* the `AddDistance` function:

```
    // Add to the coin count
    public void AddCoins(int Additional)
```

```
    {
      CoinCount += Additional;
      if (GameUI != null)
      {
        Text coinText = GameUI.transform.Find( "CoinBackground/CoinValue"
).GetComponent<Text>( );
        if (coinText != null)
        {
          coinText.text = CoinCount.ToString( );
        }
      }
    }
```

This takes the exact same steps as the `AddDistance` function. We can add `CoinCount`, check for the valid **GameUI** reference and then update its child `Text` component to change the text so that the player knows that the `CoinCount` has changed.

### Updating the coin class

The final step to having the `CoinCount` change when a coin is collected is to have the Coin class tell the character that a coin has been picked up.

In the `Assets/Scripts` folder, open the `Coin` script file by double-clicking on it.

Navigate to the `OnTriggerEnter2D` function and change the following lines:

```
gameChar.CoinCount += 1;
ActivateCoin( false );
```

To the following code:

```
gameChar.AddCoins(1);
ActivateCoin( false );
```

As we are now adjusting the **GameUI** to show the `CoinCount` on the screen, we made the `AddCoins` function in the `Character` class to do it, we no longer need to directly alter the `CoinCount` for the `Character` class in the `Coin`. We can instead select the value we want to change it using `AddCoins`.

This way, we can handle everything we need from the `Character` class instead of relying on `Coin` to directly alter the values in `Character`.

Save the `Coin` file and go back to Unity. If you play the game now, you should see the distance and coins adding up during the events that would add them up, running forward, or by collecting coins.

### The restart button

In order to complete our game loop, we need to allow the player to restart playing. As of now, as soon as the player character dies, they cannot keep playing because they will lose all control of the game. The restart button will allow them to restart playing and collect those glorious coins.

The restart button needs to do a couple of things. The major one is resetting the character so that it can run, collect coins again, and die as well. The next thing is resetting the level

pieces so that the player has the same experience as the first attempt.

We also don't want the restart button to show that the player is alive only when the character has died.

Right-click on the **GameUI** GameObject and navigate to **UI**, select **Button** from the list, and name it `RestartButton`.

With the **RestartButton** selected, in the **Inspector** tab, change the settings as follows:

- **Pos X**: `0`
- **Pos Y**: `93`
- **Pos Z**: `0`
- **Width**: `128`
- **Height**: `64`
- **Anchor**: **center bottom**
- **Source Image**: `RestartButton`

Select the **Text** child GameObject of the **RestartButton**, search for its **Inspector**, and change its settings as follows:

- **Text**: **Restart**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **32**

## The fade object

We need to create a GameObject that will hold a `GUITexture` component so that we can use it to turn the screen from completely black to completely transparent and vice versa, simulating a camera fade.

In **Hierarchy**, right-click and choose **Create Empty**. Name this new GameObject `FadeObject`.

With the **FadeObject** selected, search for the **Inspector** window and click on **Add Component**. Search for `GUITexture` and then select **GUITexture**.

Change its settings as follows:

- **Position X**: `-5`
- **Position Y**: `-5`
- **Position Z**: `0`
- **Texture**: **BlackFade**
- **Color**: `0`—red, `0`—blue, `0`—green, `255`—alpha

We will move the `FadeObject` position in the world by *negative five* for **X** and **Y** just so that the black square doesn't get in the way of our scene. We will position it to the correct location when the game starts.

### The restart functionality

We now need to set up the `GameInfo` class to manage our **RestartButton**, which includes a

fading screen and communicates with the `LevelPieceManager` and `Character` class so that they know when to restart and what to do when they restart.

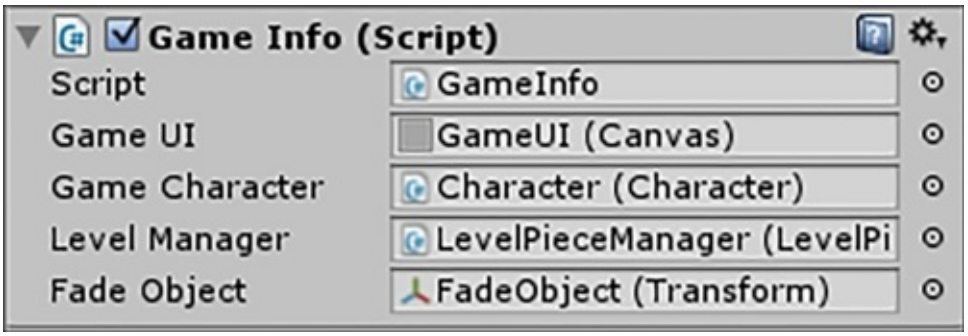
In the `Assets/Scripts` folder, double-click on the `GameInfo` script file. Add the following code at the top of the class:

```
// Reference to our Character
public Character GameCharacter;

// Reference to our levelPieceManager
public LevelPieceManager LevelManager;

// Fade object
public Transform FadeObject;

// Reference to fade GUITexture
private GUITexture FadeTexture;

// Is game restarting
private bool bRestartingGame;

// Is fading active
private bool bStartFading;

// Level and Character need resetting
private bool bLevelAndCharacterRestart;
```

- `GameCharacter`: This will hold a reference to our `Character` class in the level.
- `LevelManager`: This will hold a reference to the `LevelPieceManager` we have in our level.
- `FadeObject`: This will hold the fade texture so that we can fade our game screen from clear to black and from black to clear.
- `FadeTexture`: This is the `GUITexture` component of `FadeObject`, which is what we will be changing from black to clear.
- `bRestartingGame`: This will be used to tell `GameInfo` whether the game will restart.
- `bStartFading`: This will be used to tell `GameInfo` to start fading the screen.
- `bLevelAndCharacterRestart`: This will be used to prevent the `LevelPieces` and the `Character` from continually being reset in the `Update` function as we transition from the completely black to the transparent fade. This is because we reset the `Character` and `LevelPieces` before the fade is completely done.

Now save the `GameInfo` file and navigate back to Unity.

Select the **GameInfo** GameObject in **Hierarchy** and assign it to the object fields by left-clicking and dragging the corresponding GameObject onto the fields:




#### The Awake function

Previously, we changed the `Start` function to `Awake`. If you missed this step, go ahead and change it now and enter the following code in the function:

```
    // When object starts
    void Awake( )
    {
      if ( FadeObject != null )
      {
        FadeObject.transform.position = new Vector3( 0.5f, 0.5f, 0.0f );
        FadeTexture = FadeObject.GetComponent<GUITexture>( );
        FadeTexture.pixelInset = new Rect (0.0f, 0.0f, Screen.width,
Screen.height );

        RestartGame( );
      }
    }
```

Let's start by making sure that we have a reference to `FadeObject`. When we do, we then set the location of `FadeObject` back to default because we have it offset in the editor so that it doesn't get in the way of us working in the scene.

We then assign the global `GUITexture` variable named `FadeTexture` to the component of the `FadeObject`. This way, we can keep it stored in the class without having to always get it in other functions.

When we have the reference to the `GUITexture` variable as `FadeTexture`, we can change its `pixelInset` to a new `Rect` struct that starts at `0X` and `0Y` in the screen space (which is in the top-left corner). We can also set the size of the `pixelInset` rect to that of the screen with `Screen.width` and `Screen.height` for its `X` and `Y` values for size.

This makes the black 2D box fit the size of the screen.

Lastly, we will call a function called `RestartGame`. We will write this function soon; it will call all the things we need to make sure that the game resets, which includes the screen fading to black and back to transparent and resetting the `Character` and `LevelPieces`.

### The Update function

The `Update` function for our `GameInfo` class will handle the `FadeTexture` for fading in and out of black, when to respawn the `Character`, and reset the `LevelPieces`.

Write the following code in the `Update` function:

```
// Updates every frame
void Update( )
{       if ( bRestartingGame )
  {
    if ( bStartFading )
    {
      FadeToBlack( );
      if ( FadeTexture.color.a >= 0.95f )
      {
        bStartFading = false;
      }
    }
    else
    {
      FadeToNormal( );
      if ( FadeTexture.color.a <= 0.5f )
      {
        if (bLevelAndCharacterRestart)
        {
          // Restart level and characters before fade is done.
          RestartLevelAndCharacter();
        }

        if (FadeTexture.color.a <= 0.05f)
        {
          bRestartingGame = false;
          FadeTexture.enabled = false;
        }
      }
    }
  }
}
```

As the `Update` function is called in every frame, we need to manage it in a way that it doesn't do any extra work when we don't want it to. This is why we have a few bool values.

To start with, we need to make sure that `GameInfo` has set `bRestartingGame` to *true*. This value can only be changed when the RestartButton is shown and only when the player has clicked on it.

Next, we need to check whether `bStartFading` is set to *true*. When the `RestartButton` is pressed, we will set this along with `bRestartingGame` to *true*. The first thing we need to do when `bStartFading` is set to `true` is call the `FadeToBlack` function. This function will be written to **lerp** between the current alpha value of `FadeTexture` to a different value over the course of time. The thing with `Lerp` is that you have to call it every time you want to change it, which is why we will use the `Update` function. If we only call it once, it will only change the alpha value of `FadeTexture` once.

## Tip

**Lerp** comes from **Linear Interpolation**. It is a way to get a value that is some percentage

between two values. This is a great tool if you want to slowly move between these two values, or if you want to get a value that is between two values.

After each frame that `FadeToBlack` is called, we can change to see whether the alpha channel of `FadeTexture` is *greater than* or *equal to* `0.95f`. At this value, the alpha channel has gone opaque (or black) enough for us to start fading back out. We can start fading back to transparent (or clear) by setting `bStartFading` to *false*.

Once `bStartFading` is set to *false*, we can start by calling `FadeToNormal`, which will also use the `Lerp` function, but in the opposite direction of `FadeToBlack`, making the alpha channel of `FadeTexture` become transparent.

Now, we can check whether the alpha channel is *less than* or *equal to* `0.5f`. We need to do this because at this time, the user can now begin to see the level, and we don't want the pause effect of the game to show through the fade. Once the alpha layer has reached this `0.5f`, we then check to make sure that `bLevelAndCharacterRestart` is set to `true`.

If `bLevelAndCharacterRestart` is set to *true*, we can call the `RestartLevelAndCharacter` function. This function will do as it's named; it will restart the Level and Character back to default. It also sets the `bLevelAndCharacterRestart` bool to *false*, so this function can only be called once per restart.

We then continue to check the alpha channel of `FadeTexture` until it is *less than* or *equal* to `0.05f`. When the alpha channel reaches this point, we know that the texture is transparent enough to set the `bRestartingGame` bool to *false* and to disable `FadeTexture` with `FadeTexture.bEnabled` set to *false*.

To break this function down in a more simple way, when we restart the game, we can set `bRestartingGame` to *true*. While this is true, we can fade the screen to black and then back to transparent. During the middle point of the transparency, we can reset the character and level so that the gameplay continues on seamlessly. When the transparency is finished, we can stop the reset by setting `bRestartingGame` to *false* and disabling the `FadeTexture`.

### The RestartGame function

The `RestartGame` function is designed to start the sequence of code we just wrote for the `Update` function and make sure that it is called when it is supposed to be.

Write the following function *under* our previously written `PauseGame` function:

```
// Restarts the game
public void RestartGame( )
{
  if (Time.timeScale == 1)
  {
    FadeTexture.enabled = true;
    bLevelAndCharacterRestart = true;
    bRestartingGame = true;
    bStartFading = true;

    HideRestartButton(true);
  }
```

```
}
```

As we do not want to restart the game if the game is paused, we can start by making sure `Time.timeScale` is set to *one*, which means that the game is running at its normal speed. If we were to allow the game to restart with a time scale that is less than one, we might experience an odd behavior, such as character and level restarting.

If `timeScale` is set to *one*, we can enable the `FadeTexture` because we will need it to begin fading in and then fading out the screen from black to transparent.

We then set a series of bool values that allows the `Update` function to run its code in the correct way we want it. `bLevelAndCharacterRestart` allows the level and character to restart. `bRestartingGame` allows the fading to start happening in the correct order. Finally, `bStartFading` allows the screen to start turning black.

Lastly, we will call `HideRestartButton`, which will hide the `RestartButton` from the scene because the player has already clicked on it and doesn't need to see it anymore.

This function is here so that the restart event can begin with the correct values.

### The RestartLevelAndCharacter function

The next step is to write the code that will actually reset the level and character back to default so that the game can continue. If you remember, this gets called from the `Update` function if `bLevelAndCharacterReset` is set to *true* and the alpha value for `FadeTexture` is *less than* or *equal to* `0.5f`.

Under the `RestartGame` function, write the following function:

```
// Restart level and character
private void RestartLevelAndCharacter( )
{
  bLevelAndCharacterRestart = false;

  if ( GameCharacter != null )
  {
    GameCharacter.ReviveCharacter( );
  }

  if ( LevelManager != null )
  {
    LevelManager.ResetLevelPieces( );
  }
}
```

This function is sort of a middle man to the restart process for the character and level. The one thing this function manages for the `GameInfo` class is the first line of the function, which sets `bLevelAndCharacterRestart` to *false*. When we set this to *false*, we can tell the `Update` function that it can't call it again until we set it to *true* in the `RestartGame` function, which is called from the `RestartButton`.

We then check to make sure that the reference for `GameCharacter` exists by verifying that it is *not* null. If we have the reference, we can call `ReviveCharacter` from the `Character` class, which we wrote in Chapter 3, *Player Character, Obstacles, and Pickups*.

Finally, we check to make sure that the `LevelManager` reference exists by again verifying that it is *not* null. If it isn't, we can call the `ResetLevelPieces` function. We will write this shortly. It will be a function that puts all the `LevelPieces` back to their initial locations and resets the `StartingLevelPiece` back to its initial location.

### FadeToBlack and FadeToNormal functions

We now need to write the two functions responsible for fading our `FadeTexture` from opaque to transparent and vice versa.

Under the `RestartLevelAndCharacter` function, write these two functions:

```
    // Fade screen to black
    private void FadeToBlack( )
    {
      if (FadeTexture != null)
      {
        FadeTexture.color = Color.Lerp( FadeTexture.color, Color.black,
2.0f * Time.deltaTime );
      }
    }

    // Fade screen back to clear
    private void FadeToNormal( )
    {
      if (FadeTexture != null)
      {
        FadeTexture.color = Color.Lerp( FadeTexture.color, Color.clear,
2.0f * Time.deltaTime );
      }
    }
```

These two functions do the same thing, but in the reverse order of each other. We first check to make sure that the `FadeTexture` reference exists by verifying that it is *not* null. If we have a valid reference, we can say that the color value of `FadeTexture` is set to the return value from `Color.Lerp`.

Inside the `FadeToBlack` function, we use the `Lerp` function to change the `FadeTexture.color` value from its current color value to the `Color.black` value. This is done by taking the current value of `FadeTexture.color` and adding to it every frame by the return value of `Lerp`, which uses product of `2.0f` and `Time.deltaTime` to get a percentage between the current `FadeTexture.color` value and `Color.black` value.

For the `FadeToNormal` function, it does the same thing, but instead of fading to `Color.black`, it starts to fade to `Color.clear`, making the `FadeTexture` value more transparent.

Remember that both of these functions are called in every frame when the `bRestartingGame` is set to *true*. Each time either of them is called, they change the alpha value a small amount, resulting in a couple of seconds before the screen will fade completely black and then back to transparent, making a nice transition between pressing the `RestartButton` and actually having the game restart.

### The HideRestartButton function

The last function we need to write for `GameInfo` is the `HideRestartButton` function. The reason we need this is because we want the button to be shown only when we want it to be shown, and we do not want the player to reset the game unless they have died.

Under the `FadeToNormal` function, write the following code:

```
    // Shows or hides the restart butt
    public void HideRestartButton(bool bHide)
    {
      if (GameUI != null)
      {
        GameUI.transform.Find( "RestartButton" ).gameObject.SetActive(
!bHide );
      }
    }
```

## Tip

If you get an error telling you that **GameUI** is not set to an instance of an object, make sure to take the **GameUI** GameObject in the **Hierarchy** tab and left-click and drag it into the object field of the **GameInfo** GameObject, which is also in **Hierarchy**.

Again, we need to make sure that the reference of the **GameUI** exists by making sure that it is *not* null. Once we have this reference, we can use the `transform.Find` function to look for the GameObject named **RestartButton**, which is a child of **GameUI**. As `Find` returns a transform, we have its `gameObject` and then set this to the opposite of the function argument. We do this because if we were to take the value passed in and use it, it would make the function name look backwards.

For example, if we didn't use the opposite of what is passed and you passed *true*, it would set the button to active or shown. As the function name is `HideRestartButton`, it makes more sense that if you pass *true*, it hides it, and if you pass in *false*, it shows it.

Now, save the `GameInfo` class.

## The ResetLevelPieces function in LevelPieceManager

In the `Assets/Scripts` folder, double-click on the `LevelPieceManager` file to open it. Under the `isActivePiece` function, at the bottom of the class, write the following function:

```
    // Resets all LevelPieces
    public void ResetLevelPieces()
    {
      for (int i = 0; i < LevelPieces.Length; i++)
      {
        LevelPieces[ i ].transform.position = LevelPieces[ i
].GetInitialLocation();
        LevelPieces[ i ].ResetAllChildrenCoins();
      }

      StartingLevelPiece.transform.position =
```

```
StartingLevelPiece.GetInitialLocation( );
      StartingLevelPiece.gameObject.SetActive( true );

      Start( );
    }
```

This function starts by looping through `LevelPieces`. If you remember, `LevelPieces` were assigned by us from the editor and the number of them is dependent on how many you assigned.

In the `for` loop, we set the `transform.position` of each `LevelPiece` to the value of what is returned from the `GetInitialLocation` function. This means that the level pieces assigned will move to where we had set them in the scene before any of them were used.

We then do the same thing for the `StartingLevelPiece`. The only addition is to make sure that the `StartingLevelPiece gameObject` is active by setting `SetActive` to *true*.

Lastly, we will call the `Start` function, which is designed to begin moving the `StartingLevelPiece` and by getting a secondary piece from the world and setting its location to the `EndLocation` of the `StartingLevelPiece`.

From here, the normal code takes over, and the `LevelPieces` are moved and set based on the code we had originally written to manage them. This function is just there to make sure that they all go back to default before starting over.

Now, save the `LevelPieceManager` file.

### Updating Character To Reset

In the `Assets/Scripts` folder, double-click on the `Character` file to open it.

We need to add a few things to this class in order to make the reset functionality complete.

To begin with, add the following global variables at the top of the `Character` class:

```
    public GameInfo Game;

    [System.NonSerialized]
    public Vector3 RestartLocation;
```

The `Game` reference allows you to tell `GameInfo` that the `RestartButton` needs to show when the character dies.

`RestartLocation` will give us a location to restart the character when it dies.

In the `Start` function, add the following code:

```
  // Use this for initialization
  void Start ()
  {
    RestartLocation = gameObject.transform.position;
  }
```

We now need to update a couple of functions so that we can use these new references. Go to the `KillCharacter` function and use the following code:

```
characterRigidBody.AddForce(new Vector2(Random.Range(-1, 1), 1) * 512);
isDead = true;
isFadeOut = true;
```

Then, add the following code:

```
if (Game != null)
{
  Game.HideRestartButton(false);
}
```

This will show the `RestartButton` on screen when the character is killed, allowing the player to restart the game.

We also need to update the `ReviveCharacter` function so that `DistanceCount` gets reset when the character does, and the character uses the `RestartLocation` when it's revived.

First, change `Vector3 resetLocation = new Vector3( 0.0f,0.0f,0.0f );` to `Vector3 resetLocation = RestartLocation;`.

This will force the location of the character back to where it was assigned in `Start`.

Next, we want to reset `DistanceCount` when the character is revived.

*Under* the following code:

```
        if (gameCharacterSprite != null)
        {
          gameCharacterSprite.color = resetColorAlpha;
        }
```

Add the following code:

```
        DistanceCount = 0;
```

This will set `DistanceCount` back to zero; so, when the game restarts, so does the value of distance and in turn, the text shown on the UI will also be updated to show zero. The reason for this is that we don't want the distance to continue through to multiple attempts.

Save the `Character` file.

**Connecting RestartGame to the RestartButton click**

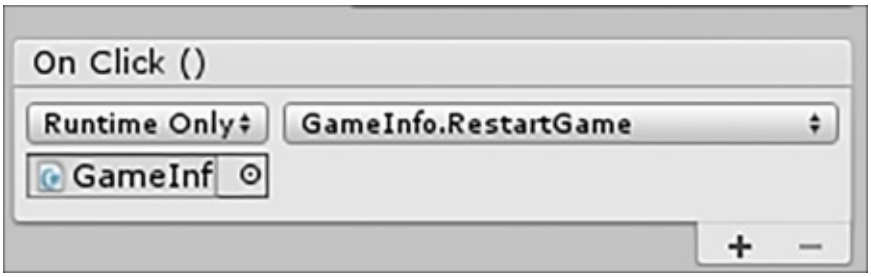
To complete the `RestartButton`, we need to make sure that its click event is set to the `RestartGame` function we wrote in `GameInfo`.

Go back to Unity, select the **RestartButton** GameObject from the **GameUI** children in **Hierarchy**.

In **Inspector**, search for the `Button` script component and click on the small plus symbol in the **On Click ()** area.

Then, left-click and drag the **GameInfo** GameObject onto the new object field of the **On Click ()** area for the **RestartButton**.

Click on the drop-down list next to the object field that is now holding the **GameInfo** GameObject. Then, navigate to **GameInfo** and select **RestartGame**:




If you play the game now, you should see the screen fade from black and have the **Character** animation play running. The `LevelPieces` should now be moving and connecting together. The **GameUI** elements should also be updating the distance traveled and the coins collected.

Also, if you die, the **RestartButton** should show. When you click on the **RestartButton**, it should hide, and the screen should fade to black and then back to transparent. During this transition, the character and level is reset.

## Tip

There has been a lot to do with the **GameInfo** GameObject and getting the game loop completed. If you are lost or having issues with code, take a look at the project files that came with the book and use them as reference.

# PlayerPrefs game save object

Saving data using the `PlayerPrefs` class is not very difficult. Basically, the `PlayerPrefs` class is designed to save the basic variable data so that it can be used between game sessions. In our case, we need to save *two* things. The first is the `CoinCount` in `Character`. We also need to check whether the user has bought the remove ads feature of the game.

For this chapter, we will start by making sure that the coins are saved between game sessions.

Go to the `Assets/Scripts` folder and open the `Character` file.

In the `Start` function, add the following code *under* where we assigned `RestartLocation`:

```
 CoinCount = PlayerPrefs.GetInt( "Coins" );
AddCoins( 0 );
```

This will set `CoinCount` to whatever has been saved. We can also use the `AddCoins` function and pass *zero* so that the **GameUI** updates the UI to show how many coins we have.

Next, we need to see where to save the coins. As we don't have to save the `PlayerPrefs` value every time a coin is picked up, we can use the point at which the player is killed to do so. This way, we aren't saving every coin and saving some overhead as well.

Find the `KillCharacter` function and *under* the following code:

```
 if (Game != null)
{
  Game.HideRestartButton(false);
}
```

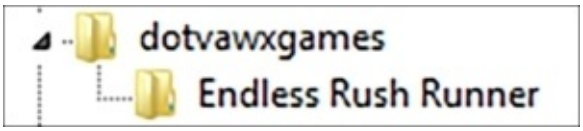
Add the following code:

```
 PlayerPrefs.SetInt("Coins", CoinCount);
 PlayerPrefs.Save();
```

Now, if you play the game, collect some coins, and stop playing, when you play again, it will show the coins you collected in the previous session as how many you have.

If you are on a Windows machine, the save data is saved in the registry. To get there, click on the **Start** button and enter `regedit`. This will show a program called `regedit.exe`. The save data is in `HKEY_CURRENT_USER/Software/[YourCompanyName]/[YourGameName]`.

For example, mine is in `HKEY_CURRENT_USER/Software/dotvawxgames/EndlessRushRunner`:




If you want to set your company and game name, navigate to Unity, click on **Edit** and then

**Project Settings**, and select **Player** from the list. In **Inspector**, the top two settings are **Company Name** and **Product Name**. Set these, and the save file will use them as the directory in the registry.

If you are on a Macintosh computer, the save data is stored in the **Library** | **Preferences** folder in a file named `unity.[company name].[product name].plist`.

# Summary

Our `SceneBackground` is a tool that can take a lot of different types of sprites that can all have their own settings for movement purposes.

We also began working with Unity 5's UI system. This allows you to create a UI screen that will dynamically stay true to what we want, irrespective of the screen size or pixel density. We also created a pause button that will change the time scale of the game down to *zero*, which prevents the game elements from running. Finally, we created a restart button that can be hidden or shown to the user. It also fades out and fades in the screen when you reset the `Character` and `LevelPieces` back to default.

Finally, we used the `PlayerPrefs` class to save data to the local machine so that the user can keep their coin score, irrespective of whether they stop playing.

This chapter has done a lot of things that are a bit more technical than what we have done before. There has been a lot of code updates, fixes, changes, and brand new code as well. If you are new to this, take it slow. The topics in this chapter can make game development challenging, but I assure you that if you keep pace, you will begin to understand it.

In the next chapter, we will take a look at how to create the main menu and make sure that iAds, leaderboards, store purchases, and achievements are all functional in the Main menu.

# Chapter 6. Main Menu, iAds, Leaderboards, Store Purchases, and Achievements

In this chapter, we will go through the steps to get the final aspects of the iOS integration function and set up the main menu UI so that the player can navigate between playing the game, view leaderboards /achievements, and have the option to purchase “remove iAds” for the cost of *ten thousand* coins or *99* cents. We will also display iAds if the player has not purchased the remove iAds.

Similar to the last chapter, this one will have a fair bit of the tedious UI work. As it is difficult to explain the workflow, I will give you the exact locations and anchor points for the UI buttons and images. These locations are not definitive, meaning you can change them however you feel. Also, it won’t affect the players experience with the UI because the logic for them is based in code. Feel free to take the liberty with the UI design if you are comfortable with it.

iAds, Leaderboards, and Achievements are all handled with the iOS plugin. Sadly, Unity has not integrated the native iOS functionality into the engine, and the iOS plugin is required to call the setting and get information for leaderboards and achievements and when and how to display the iAds. As I mentioned in the first chapter, I will use the plugin called IOS Native. It is on the Unity asset store for *twenty* dollars. If you do not want to spend that much money, there are some other options, although I have found that IOS Native is one of the better ones.

## Tip

There are many alternatives to IOS Native. All of them have different methods to do the same thing at different costs. If for whatever reason you would like to use another or want to see the alternatives, you can use the Unity Asset Store to find them.

We will also need to expand our existing class in order to fill the gaps to save information regarding achievements, leaderboards, and store purchases. Apple also requires a restore purchase option.

In this chapter, we will cover the following topics:

- Building the main menu user interface with Unity’s UI tools, including buttons for achievements, leaderboards, and store purchases
- Displaying iAds on the screen if the player has not purchased remove iAds
- Writing code that hides and shows different aspects of the UI, such as the transition between the main menu and the game, any call needed for store purchases, or bringing up the iOS leaderboard and achievement scenes
- Addition to `LevelPieceManager` so that the character continues to run on a flat ground until the player chooses to start playing
- Addition to character and saving the system so that it can check whether

achievements are completed

# Building the main menu UI

The main menu UI will be its own Canvas GameObject. We will then handle the main menu and the game UI via the `GameInfo` class. We will also use the `GameInfo` class to manage button presses and the iOS integration.

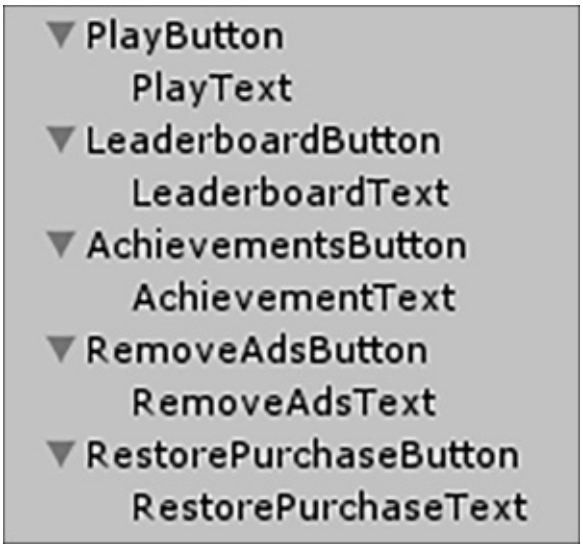
In **Hierarchy**, right-click and select **UI** and then click on **Canvas**. Name this new **Canvas** GameObject `MenuUI`.

Let's start by adding *five* buttons to achievements, playing, leaderboards, remove iAds, and restore purchase.

Right-click on the new **MenuUI** GameObject, navigate to **UI**, and left-click on **Button**. Do this *four* more times, so there are a total of *five* buttons that are children of the Menu UI GameObject.

Name the buttons and text children as follows:

- `PlayButton`, `PlayText`
- `LeaderboardButton`, `LeaderboardText`
- `AchievementButton`, `AchievementText`
- `RemoveAdsButton`, `RemoveAdsText`
- `RestorePurchaseButton`, `RestorePurchaseText`



# Adding button images

Next, we need to import the art that will be used for the main menu UI. In the `Assets/UI` folder, right-click and select **Import New Asset…**. Navigate to the `Art` folder for this book and search for the chapter labeled `ChapterSix_MenuUI`. Import all the images into this folder.

Select all the new images in the `Assets/UI` folder and change their settings as follows:

- **Filter Mode**: **Trilinear**
- **Max Size**: `256`
- **Format**: **Truecolor**

## PlayButton

Select **PlayButton** in **Hierarchy** and search for **Inspector**. Change its settings as follows:

- **Anchor**: **bottom center**
- **Pos X**: `0`
- **Pos Y**: `115`
- **Pos Z**: `0`
- **Width**: `128`
- **Height**: `128`
- **Source Image**: `MenuButton`

Now, select **PlayButtonText**. In the **Inspector** window, change its settings as follows:

- **Text**: **Play**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **36**
- **Alignment**: **Center**

## LeaderboardButton

Select **LeaderboardButton** in the **Hierarchy** tab and search for **Inspector**. Change its settings as follows:

- **Anchor**: **bottom center**
- **Pos X**: `135`
- **Pos Y**: `115`
- **Pos Z**: `0`
- **Width**: `128`
- **Height**: `128`
- **Source Image**: `MenuButton`

Select **LeaderboardText**. In the **Inspector** window, change its settings to:

- **Text**: **Leaderboards**
- **Font**: **Arial**
- **Font Style**: **Bold**

- **Font Size**: **17**
- **Alignment**: **Center**

## AchievementButton

Select **AchievementButton**. In **Hierarchy**, search for **Inspector**. Change its settings as follows:

- **Anchor**: **bottom center**
- **Pos X**: `-135`
- **Pos Y**: `115`
- **Pos Z**: `0`
- **Width**: `128`
- **Height**: `128`
- **Source Image**: `MenuButton`

Now, select **AchievementText** and then in **Inspector**, change its settings to:

- **Text**: **Achievements**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **17**
- **Alignment**: **Center**

## RemoveAdsButton

Select **RemoveAdsButton** in the **Hierarchy** tab and navigate to **Inspector**. Change its settings as follows:

- **Anchor**: **bottom center**
- **Pos X**: `-64`
- **Pos Y**: `55`
- **Pos Z**: `0`
- **Width**: `96`
- **Height**: `42`
- **Source Image**: `RestartButton`

Now, select **RemoveAdsText** and then in the **Inspector** window, change its settings as shown here:

- **Text**: **Remove iAds**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **12**
- **Alignment**: **Center**

## RestorePurchaseButton

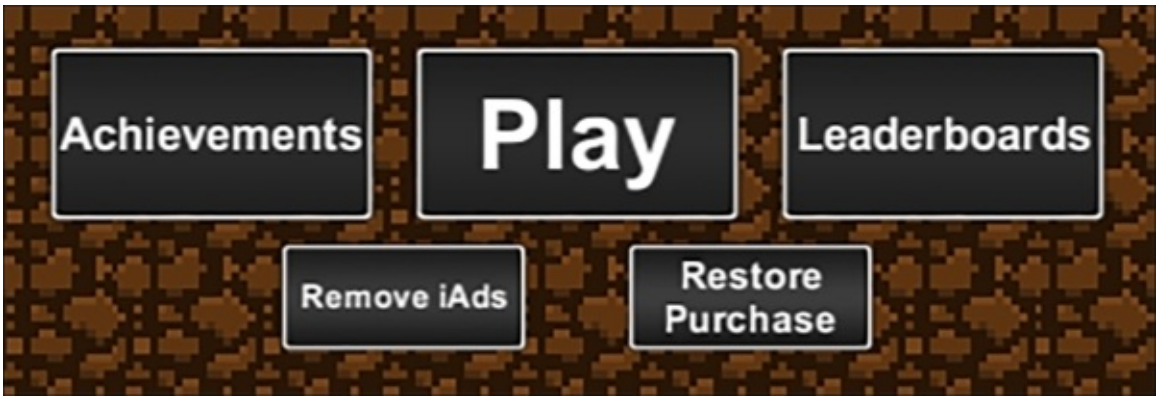
Let’s select **RestorePurchaseButton** in the **Hierarchy** tab and search for **Inspector**. Change its settings as follows:

- **Anchor**: **bottom center**
- **Pos X**: `64`
- **Pos Y**: `55`
- **Pos Z**: `0`
- **Width**: `96`
- **Height**: `42`
- **Source Image**: `RestartButton`

Now, select **RestorePurchaseText** and then in the **Inspector** window, change its settings as follows:

- **Text**: **Restore Purchase**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **14**
- **Alignment**: **Center**

You should now have a button layout that looks similar to the following image:




## The Purchase Remove iAds panel

We need to create a few panels that will show when the player clicks on **RemoveAdsButton** and what selection they choose from here. The selection to purchase remove iAds with *ten thousand* coins will check whether the player has that many. If they don't, we will display an error panel that tells how many coins they have and that the purchase has failed. We also need a panel to show when the player has successfully purchased the remove iAds purchase.

- Select the **MenuUI** GameObject from the **Hierarchy** tab and then right-click on it. Navigate to **UI** and select **Panel**. Name this `RemoveAdsBackgroundScreen`.
- With **RemoveAdsBackgroundScreen** selected, in the **Hierarchy** tab, right-click on it and select **UI** and then **Image**. Name this image `RemoveAdsScreen`.
- Still, with **RemoveAdsBackgroundScreen** selected, right-click on it, navigate to **UI**, and select **Text**. Name this text `RemoveAdsTextOption`.
- Finally, again with **RemoveAdsBackgroundScreen** selected, right-click on it, navigate to **UI**, and select **Button**. Name this button `ForCoin`. Do this twice more. Name the next button `ForCash` and the last button `CloseRemoveAds`.

- Name the child **Text** GameObject for **ForCoin** as `ForCoinText`.
- Name the child **Text** GameObject for **ForCash** as `ForCashText`.
- Delete the child **Text** GameObject for `CloseRemoveAds`.

**RemoveAdsBackgroundScreen**

Select **RemoveAdsBackgroundScreen** in the **Hierarchy** tab and navigate to the **Inspector** window. Change its settings as follows:

- **Anchor**: **Center**
- **Pos X**: `0`
- **Pos Y**: `0`
- **Pos Z**: `0`
- **Width**: `265`
- **Height**: `180`
- **Source Image**: `Background`

**RemoveAdsScreen**

Select **RemoveAdsScreen** in **Hierarchy** and search for **Inspector**. Change its settings as follows:

- **Anchor**: **Center**
- **Pos X**: `0`
- **Pos Y**: `0`
- **Pos Z**: `0`
- **Width**: `256`
- **Height**: `171`
- **Source Image**: None
- **Color**: `32`—red, `32`—green, `32`—blue, `255`—alpha

**RemoveAdsTextOption**

- **Anchor**: **top center**
- **Pos X**: `0`
- **Pos Y**: `-40`
- **Pos Z**: `0`
- **Width**: `155`
- **Height**: `64`
- **Text**: **Purchase Remove iAds**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **24**
- **Alignment**: **Center**

**ForCoin**

- **Anchor**: **bottom left**
- **Pos X**: `68.5`
- **Pos Y**: `68.5`

- **Pos Z**: `0`
- **Width**: `128`
- **Height**: `128`
- **Source Image**: `Pickup_Coin_0`

**ForCoinText**

- **Anchor**: **stretch**
- **Pos X**: `6.1`
- **Pos Y**: `101.5`
- **Pos Z**: `0`
- **Right**: `-6.1`
- **Bottom**: `7.5`
- **Text**: **10,000 Coins**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **14**
- **Alignment**: **Center**

**ForCash**

- **Anchor**: **bottom left**
- **Pos X**: `-68.5`
- **Pos Y**: `68.5`
- **Pos Z**: `0`
- **Width**: `76`
- **Height**: `76`
- **Source Image**: `CentSign`

**ForCoinText**

- **Anchor**: **stretch**
- **Pos X**: `6.1`
- **Pos Y**: `101.5`
- **Pos Z**: `0`
- **Right**: `-6.1`
- **Bottom**: `7.5`
- **Text**: **99 Cents**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **14**
- **Alignment**: `Center`

**CloseRemoveAds**

- **Anchor**: **top left**
- **Pos X**: `10`
- **Pos Y**: `-10`
- **Pos Z**: `0`

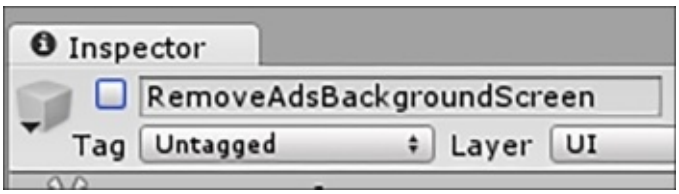
- **Width**: `32`
- **Height**: `32`
- **Source Image**: `CloseButton`

Your **RemoveAdsBackgroundScreen** should now look similar to the following image:

## The Purchase Succeeded panel

The next window we need is the panel that shows when the purchase succeeded.

To make your work easy, select **RemoveAdsBackgroundScreen** and then in the **Inspector** window, click on the checkbox next to its name in order to disable the GameObject. This will hide it from the scene, so when we create the next panel, they won't overlap.




1. First, right-click on the **MenuUI** GameObject, navigate to **UI**, and left-click on **Panel**. Name this Panel `PurchaseSucceededBackgroundScreen`.
2. Then, right-click on **PurchaseSucceededBackgroundScreen**, navigate to **UI**, and left-click on **Image**. Name this image `PurchaseSucceededScreen`.
3. Now, right-click on **PurchaseSucceededBackgroundScreen**, navigate to **UI**, and left-click on **Text**. Name this text `PurchaseSucceededText`.
4. Right-click on **PurchaseSucceededBackgroundScreen**, navigate to **UI**, and left-click on **Button**. Name this button `PurchaseSucceededAccept`. Name its **Text** child as `PurchaseSucceededAcceptText`.
5. Finally, let's right-click on **PurchaseSucceededBackgroundScreen**, navigate to **UI**, and left-click on **Button**. Name this button `PurchaseSucceededClose` and delete its **Text** child.

**PurchaseSucceededBackgroundScreen**

- **Anchor**: **center**
- **Pos X**: `0`
- **Pos Y**: `0`
- **Pos Z**: `0`
- **Width**: `265`
- **Height**: `180`
- **Source Image**: `Background`

**PurchaseSucceededScreen**

- **Anchor**: **center**
- **Pos X**: `0`
- **Pos Y**: `0`
- **Pos Z**: `0`
- **Width**: `256`
- **Height**: `171`
- **Source Image**: None
- **Color**: `19`—red, `19`—G, `19`—blue, `255`—alpha

**PurchaseSucceededText**

- **Anchor**: **center**
- **Pos X**: `0`
- **Pos Y**: `0`
- **Pos Z**: `0`
- **Width**: `254`
- **Height**: `30`
- **Text**: **Purchase Succeeded!**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **24**
- **Alignment**: **Center**

**PurchaseSucceededAccept**

- **Anchor**: **center**
- **Pos X**: `0`
- **Pos Y**: `-50`
- **Pos Z**: `0`
- **Width**: `64`
- **Height**: `64`
- **Source Image**: `MenuButton`

**PurchaseSucceededAcceptText**

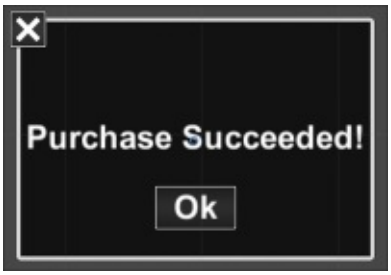
- **Anchor**: **stretch**
- **Pos X**: `0`
- **Pos Y**: `0`
- **Pos Z**: `0`

- **Text**: **Ok**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **24**
- **Alignment**: **Center**

**PurchaseSucceededClose**

- **Pos X**: `10`
- **Pos Y**: `-10`
- **Width**: `32`
- **Height**: `32`
- **Source Image**: `CloseButton`

Your **PurchaseSucceeededBackgroundScreen** should now look similar to the following image:




## The Purchase Failed panel

We need the exact same window for our failed window. There are *two* ways we could handle this. The first and more involved way is to use the succeeded window and change the text, so when something fails, we say it failed. The other way is to duplicate the same window and rename everything that says “Succeeded” to “Failed”. For the sake of ease, go ahead and left-click on **PurchaseSucceededBackgroundScreen** in **Hierarchy** to select it.

Then, click on *Ctrl* + *D*. This will duplicate the GameObject. In this duplicated GameObject, rename everything from **Succeeded** to `Failed`, including the text component of **PurchaseFailedText**.

You should now have *two* separate **Panel** GameObjects in your **Hierarchy**, as shown in the following screenshot:

We now need to write the code for the buttons of the **PurchaseSucceeded** panel and the **PurhcaseFailed** panel.

In the `Assets/Scripts` folder, double-click on the `GameInfo` class to open it.

At the top of the `GameInfo` class, add the following variables:

```
    // Reference to the RemoveAds screen
    private Transform RemoveAdsBackgroundScreen;

    // Reference to the Purchase screen
    private Transform PurchaseSucceededScreen;

    // Reference to the Purchase failed screen
    private Transform PurchaseFailedScreen;
```

Then, in the `Start` function, add the following code:

```
    // Called after Awake but before Update
    void Start( )
    {
      if ( MainMenuUI != null )
      {
        RemoveAdsBackgroundScreen = MainMenuUI.transform.Find(
"RemoveAdsBackgroundScreen" );
        PurchaseSucceededScreen
        = MainMenuUI.transform.Find( "PurchaseSucceededBackgroundScreen" );
        PurchaseFailedScreen = MainMenuUI.transform.Find(
"PurchaseFailedBackgroundScreen" );
      }
    }
```

## Tip

If the `Start` function doesn't exist in the class because it was removed earlier, write it under the `Awake` function.

Under `HideRestartButton`, write the following functions:

```
      // Show or hide Purchase Window
      public void ShowPurchaseScreen( bool bShow )
```

```
    {
      if (RemoveAdsBackgroundScreen != null)
      {
        RemoveAdsBackgroundScreen.gameObject.SetActive( bShow );
      }
    }

    // Show or hide Success Window
    public void ShowSuccessScreen( bool bShow )
    {
      if (PurchaseSucceededScreen != null)
      {
        PurchaseSucceededScreen.gameObject.SetActive( bShow );
      }
    }

    // Show or hide Fail Window
    public void ShowFailScreen( bool bShow )
    {
      if (PurchaseFailedScreen != null)
      {
        PurchaseFailedScreen.gameObject.SetActive( bShow );
      }
    }
```

All three of these functions will handle their connected screen in the same way. If the `bShow` bool value is set to `true`, it will show the associated GameObject. If `bShow` is set to `false`, it will hide it. This is done using the `SetActive` function.

Now, save the `GameInfo` class.

We now need to make sure that the buttons in the **MenuUI** are using the correct function calls.

Let’s go back to Unity and select the **CloseRemoveAds** GameObject in the **MenuUI | RemoveAdsBackgroundScreen** GameObject. In **Inspector**, search for the `Button` component and click on the small plus symbol to add an element to the **On Click ()** list. Left-click and drag the **GameInfo** GameObject onto the object field and then select the drop-down menu located at the right. Navigate to **GameInfo** and then select the `ShowPurchaseScreen` function. Leave the small checkbox under the drop-down list *unchecked*.

Select the **PurchaseSucceededAccept** GameObject in the **MenuUI | PurchaseSucceededBackgroundScreen** GameObject. Again, navigate to the **Inspector** window. In the **Button** component, click on the small **plus** symbol to add an element to the **On Click ()** list. Then, left-click and drag the **GameInfo** GameObject onto the object field. From the drop-down list, select **GameInfo** and then the **ShowSuccessScreen** function. Again, leave the small checkbox *unchecked*.

Do this again for the **PurchaseSucceededClose** GameObject, also a child of the **PurchaseSucceededBackgroundScreen** GameObject, which is in the **MenuUI** GameObject. Use the same **ShowSuccessScreen** function and leave the small checkbox *unchecked*.

Follow the same steps for **PurchaseFailedBackgroundScreen**, but instead of selecting **ShowSuccessScreen**, select **ShowFailedScreen**. Also, do this for the **PurchaseFailedClose** button.

## Connecting the MainMenuUI reference to GameInfo

At the top of the `GameInfo` class, add the following variable:

```
// MainMenu UI reference
public Canvas MainMenuUI;
```

Next, select the **GameInfo** GameObject in **Hierarchy** by left-clicking on it. Then, left-click and drag **MenuUI** onto the **Main Menu UI** object slot of the **GameInfo** GameObject in **Inspector**.

Then, select the **GameUI** GameObject in **Hierarchy**. If it is not already hidden, click on the checkbox at the top of the **Inspector** window near the GameUI name to hide it.

Now, select the children GameObjects in **MenuUI** named **RemoveAdsBackgroundScreen**, **PurchaseSucceededBackgroundScreen**, and **PurchaseFailedBackgroundScreen**. Then, use the checkbox at the top of the **Inspector** window near their names and hide them.

The menu should now only show the **Achievements**, **Play**, **Leaderboards**, **Remove iAds**, and **Restore Purchase** buttons.

# The menu code

My goal is to have the code compile at every section of this book. As of now, many of the alterations in this section will feel disorganized. It was a judgment call on my end to *not* have you do work on something that will break everything until a later chapter, which is why in this chapter, we will go back to some C# classes in order to expand or change the existing code within them. Although it is not uncommon for this sort of thing to happen while creating a video game, I know that it can be a bit irritating when you learn to have some back and forth.

## The GameInfo menu code

For the main menu to have movement without any game play, we need to set a few things so that the level pieces are used only when they are needed. In the `Assets/Scripts` folder, open the `GameInfo` class by double-clicking on it.

At the top of the `GameInfo` class, add the following global variables:

```
// If game is running or at menu idle
[System.NonSerialized]
public bool bGameRunning;
// GameUI reference
public Canvas GameUI;
```

Next, we need a function to call when **Start** button is pressed by the player. This button will transition **MainMenuUI** over to **GameUI** as well as change the way the **LevelPieces** are being connected. At the *bottom* of the `GameInfo` class, add the following function:

```
// Game state button was pressed
public void GameStateButtonPressed( bool bRunning )
{
  bGameRunning = bRunning;

  if (MainMenuUI != null)
  {
    MainMenuUI.gameObject.SetActive(!bRunning);
  }

  if (GameUI != null)
  {
    GameUI.gameObject.SetActive(bRunning);
  }

  RestartGame( );
}
```

This function will be used by `PlayButton` and a button that we will create to return to the main menu. As our game state is either *true* or *false*, we can use the single function and enter what type of game state we are looking for. For `PlayButton`, we want the game state to be set to *true*. For the menu button, we will set it to *false*.

In the last chapter, we saw how the `RestartGame` function began the cycle of fading the screen to black and then back to transparent. At the midpoint of going back to transparent,

we can restart the character and the `LevelPieces`. As of the new flow of getting into the gameplay, we also need to update the `RestartLevelAndCharacter` function so that it sends the correct information to `LevelManager`. Change the `RestartLevelAndCharacter` function as follows:

```
// Restart level and character
private void RestartLevelAndCharacter( )
{
  bLevelAndCharacterRestart = false;

  if ( GameCharacter != null )
  {
    GameCharacter.ReviveCharacter( );
  }

  if ( LevelManager != null )
  {
    LevelManager.ResetLevelPieces( bGameRunning );
  }
}
```

## The LevelPieceManager menu code

We will now pass the `bGameRunning` bool value to the `RestartLevelPieces` function of the LevelManager. To accommodate the new bool value, navigate to the `Assets/Scripts` folder and double-click on the `LevelPieceManager` C# file to open it.

Before we change the `RestartLevelPieces` function, we need to add a couple of global variables to the class. At the top of the `LevelPieceManager` class, add the following code:

```
[System.NonSerialized]
// If the game is being played or at main menu
public bool bGameRunning;

// MainMenu level piece
public LevelPiece IdleLevelPiece;
```

From the `LevelPieceManager` perspective, `bGameRunning` will handle the current state of the game. `IdleLevelPiece` will be the secondary `LevelPiece` that we will use during the menu portion of the game (where the character runs forever).

We need to change the `Start` function as well because it was handling `ActiveLevelPieces` too directly for what our game needs now. Change the `Start` function as follows:

```
// Use this for initialization
void Start( )
{
  ActiveLevelPieces = new LevelPiece[2];
  ResetLevelPieces( bGameRunning );
}
```

This now only creates the `ActiveLevelPieces` array and then uses the `RestartLevelPieces` function to begin running the game in the correct state. As bool

values default to *false*, we know that when we call this using `Start`, the game will start in the idle mode.

Change the `RestartLevelPieces` function in the `LevelPieceManager` class, as shown in the following code:

```
    // Resets all LevelPieces
    public void ResetLevelPieces( bool bRunning )
    {
      bGameRunning = bRunning;

      StartingLevelPiece.transform.position =
StartingLevelPiece.GetInitialLocation();
      StartingLevelPiece.gameObject.SetActive(true);
      IdleLevelPiece.gameObject.SetActive( !bGameRunning );

      for (int i = 0; i < LevelPieces.Length; i++)
      {
        LevelPieces[i].transform.position =
LevelPieces[i].GetInitialLocation();
        LevelPieces[i].ResetAllChildrenCoins();
      }

      if (bGameRunning)
      {
        SetGamePieces();
      }
      else
      {
        SetIdlePieces();
      }
    }
```

You'll see here that this class also has a `bGameRunning` bool value. This is because `LevelPieceManager` is in sync with `GameInfo`. We can also reset `StartingLevelPiece` and make sure that it can be seen because this piece is used in both the menu and game states. We can then set `IdleLevelPiece` so that it is only active if the `bGameRunning` bool is set to `false`. We do not want to use it when the game is running.

No matter what the game state is, we need to reset the `LevelPieces` before running `SetIdlePieces` or `SetGamePieces`. This way, the `LevelPieces` are out of the view of the player until they are needed, which is handled in the `Update` function. We accomplish this with the `for` loop that sets the `LevelPieces` to their `GetInitialLocation` and also resets all of their children, coins, and obstacles so that they can be used again by the player.

This function also has two more function calls: `SetGamePieces` and `SetIdlePieces`. These two functions are designed to make the `ActiveLevelPieces` fit to whichever game state we are in: idle or game.

In the `Assets/Scripts` folder, double-click on the `LevelPiece` code file to open it. At the top of the class, add the following global variable:

```
    // End location of Level Piece
    public Transform EndLocation;
```

This will give us a direct reference to the `EndLocation Transform` so that we don't have to find it every time we use it.

Save this file and go back to Unity. Select each of the `LevelPieces`, take the GameObject with the name of **EndLocation** in **Hierarchy**, and left-click and drag it onto the new **EndLocation** object field of the `LevelPiece`. Do this for *all* the `LevelPiece` GameObjects in the scene.

## Tip

Remember to put **EndLocation**, a child for *each* of the **LevelPiece** GameObjects, and *not* put the same **EndLocation** for all of them.

Back in the `LevelPieceManager`, *under* the `Start` function, write the follow functions:

```
    // Set ActiveLevelPieces to Idle
    void SetIdlePieces( )
    {
      ActiveLevelPieces[ 0 ] = StartingLevelPiece;
      ActiveLevelPieces[ 1 ] = IdleLevelPiece;
      ActiveLevelPieces[ 1 ].transform.position =
StartingLevelPiece.EndLocation.position;
    }

    // Set ActiveLevelPieces to Game
    void SetGamePieces( )
    {
      ActiveLevelPieces[0] = StartingLevelPiece;
      ActiveLevelPieces[1] = GetRandomLevelPiece();
      ActiveLevelPieces[1].transform.position =
StartingLevelPiece.EndLocation.position;
    }
```

`SetIdlePieces` takes `StartingLevelPiece` and `IdleLevelPiece` and assigns them to the *zero* and *one* elements of `ActiveLevelPieces`. It also takes `IdleLevelPiece` and moves it to the `EndLocation` of the first element of `ActiveLevelPieces`, which is more easily referred to as `StartingLevelPiece`.

`SetGamePieces` does the same thing as we had the `Start` function doing earlier. It assigns the *first* element of `ActiveLevelPieces` to `StartingLevelPiece` and then uses `GetRandomLevelPiece` to assign the next element. It then assigns the return `LevelPiece` from the `GetRandomLevelPiece` position to the `EndLocation` of `StartingLevelPiece`.

These two functions behave very similarly. The only difference is what's being assigned.

We now need to change the `Update` function to change between moving the game `LevelPieces` and the idle `LevelPieces`. Change the current `Update` function as follows:

```
  // Update is called once per frame
  void Update ( )
  {
    for (int i = 0; i < ActiveLevelPieces.Length; i++)
    {
      Vector3 newLocation = ActiveLevelPieces[i].transform.position;
      newLocation.x -= LevelPiecesMoveRate * Time.deltaTime;

      ActiveLevelPieces[i].transform.position = newLocation;

      if (ActiveLevelPieces[i].transform.position.x < transform.position.x)
      {
        if (bGameRunning)
        {
          if (ActiveLevelPieces[i] == StartingLevelPiece)
          {
            ActiveLevelPieces[i].gameObject.SetActive(false);
          }

          ActiveLevelPieces[i].transform.position =
ActiveLevelPieces[i].GetInitialLocation();

          ActiveLevelPieces[i] = GetRandomLevelPiece();
          ActiveLevelPieces[i].transform.position =
FindOtherLevelPiece(ActiveLevelPieces[i]).EndLocation.position;
          ActiveLevelPieces[i].ResetAllChildrenCoins();
        }
        else
        {
          LevelPiece nextLevelPiece = ( i == 0 ) ? ActiveLevelPieces[ 1 ] :
ActiveLevelPieces[ 0 ];

          ActiveLevelPieces[ i ].transform.position =
nextLevelPiece.EndLocation.position;
        }
      }
    }
  }
```

You'll see that this time, we are using the `bGameRunning` bool to decide how to transition between the `ActiveLevelPieces`. Luckily, the menu only uses *two*, so if `bGameRunning` is set to *false*, we can assign `nextLevelPiece` with the opposite of what element `i` represents. As we know that there are only *two* `ActiveLevelPieces`, if the game is not running, we can use the value of `i` to get the other `ActiveLevelPieces` element, using the ternary operator to check whether I is equal to zero. If it is, we will assign `nextLevelPiece` to the *second* element of the `ActiveLevelPieces` array. If it isn't, we will assign `nextLevelPiece` to the *first* element of the `ActiveLevelPieces` array.

Whichever piece gets chosen as `nextLevelPiece`, we set the current `i` value of the `ActiveLevelPiece` array position to the end of it. This takes the `ActiveLevelPiece` that has gone out of viewpoint of the player and moves it back so that it can be used again.

The code used if `bGameRunning` is set to *true* is the same as we had before; if a piece gets

out of view of the player, we can choose another with `GetRandomLevelPiece`, reset it to default, and attach it at the end of the moving `ActiveLevelPieces` so that it can be used again.

Now, save the `LevelPieceManager` class.

## The character menu code

The last step to getting the idle menu code working is adding it to the character so that it is aware when it should be counting up distance and when to accept input.

In the `Assets/Scripts` folder, open the `Character` C# file by double-clicking on it. At the top of the class, add the following global variable:

```
public LevelPieceManager LevelManager;
```

As there is a delay fading the `GameInfo` class, we will use the `bGameRunning` bool `LevelPieceManager` as the more accurate one before the `LevelPieceManager` becomes aware that the game is running. This way, the distance calculations and the input will be more in line with when the player starts to see the game scene.

We now need to use the `LevelManager.bGameRunning` bool value in a couple of places. Navigate to the `AddDistance` function and change `if (GameUI != null)` to `if (GameUI != null && LevelManager.bGameRunning)`.

This will prevent any distance calculations before the game is set to running. Next, navigate to the `RecieveInput` function and change `if (characterRigidBody != null && characterAnimator != null)` to `if (characterRigidBody != null && characterAnimator != null && LevelManager.bGameRunning)`.

This again prevents any player input from manipulating the character unless the game is running.

Now, save the `Character` class.

## Completing changes in Unity

For the changes to be complete, we need to add the new references and GameObjects to the scene.

Open the `Assets/Prefabs` folder and left-click and drag the `StartingLevelPiece` prefab onto the scene. Move it away from the view of the camera; I put mine near the other `LevelPieces`. In **Hierarchy**, rename it as `IdleLevelPiece`.

Select the **LevelPieceManager** GameObject in **Hierarchy** and then left-click and drag the new **IdleLevelPiece** GameObject from **Hierarchy** onto the **Idle Level Piece** object slot of **LevelPieceManager** in the **Inspector** window.

Select the **Character** GameObject from **Hierarchy** by left-clicking on it. Then, left-click and drag the **LevelPieceManager** onto the **Level Manager** object slot of **Character** in the **Inspector** window.

Select the **PlayButton** GameObject under the **MenuUI** GameObject. In the **Button**

component, in the **Inspector** window, click on the small plus sign to add a new **On Click ()** list element. Left-click and drag the **GameInfo** GameObject from **Hierarchy** and move it into the object slot of the **On Click** list element. Left-click the drop-down menu for the **On Click ()** list and select **GameInfo** and then select **GameStateButtonPressed**. Also, for **PlayButton**, we need to tick the small box that appears under the **On Click ()** drop-down menu because we want **PlayButton** to enter *true* for the game state to change to *true* so that the **LevelPieceManager** and **GameInfo** will know that the game is running.

You should now be able to play the game and see the character run forever on the menu, but when you press play, **MenuUI** will disappear and **GameUI** will show. The screen will also fade to black and back to transparent, and the game will start normally with the `LevelPieces` being put together and reset as the game continues. If the character dies, the restart button will be shown.

# Adding the BackToMenu button in GameUI

The next step is to allow the player to go back to the menu from the game when they want to. In **Hierarchy**, select **MenuUI** and use the checkbox next to its name in Inspector to hide it from the scene. Then, do the same for **GameUI**, but instead of hiding it, it shows it in the scene.

With the **GameUI** GameObject still selected in **Hierarchy**, right-click on it and navigate to **UI** and then select **Button**. Name this button `BackToMenuButton`. Name its child **Text** GameObject `BackToMenuText`.

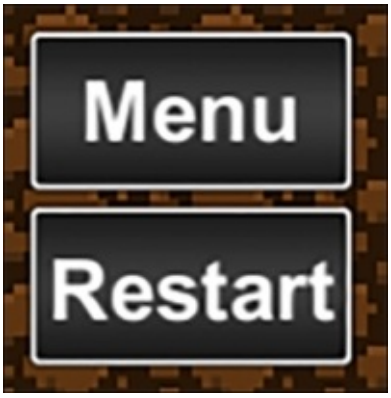
Now, change the **BackToMenuButton** settings in **Inspector** as follows:

- **Anchor**: **bottom center**
- **Pos X**: `0`
- **Pos Y**: `158.5`
- **Pos Z**: `0`
- **Width**: `128`
- **Height**: `64`
- **Source Image**: `RestartButton`

Now, change the **BackToMenuText** settings in the **Inspector** as follows:

- **Anchor**: **stretch**
- **Text**: **Menu**
- **Font**: **Arial**
- **Font Style**: **Bold**
- **Font Size**: **32**
- **Alignment**: **Center**
- **Color**: `255`—red, `255`—green, `255`—blue, `255`—alpha

You should now have another button before the **Restart** button with the **Menu** text, as shown in the following image:




We now need to update the `HideRestartButton` function in the `GameInfo` class so that it also hides or shows the **Menu** button with it. In the `Assets/Scripts` folder, open the `GameInfo` C# class by double-clicking on it.

Find `HideRestartButton` and change it, as shown in the following code:

```
    // Shows or hides the restart butt
    public void HideRestartButton(bool bHide)
    {
      if (GameUI != null)
      {
        GameUI.transform.Find( "RestartButton" ).gameObject.SetActive(
!bHide );
        GameUI.transform.Find( "BackToMenuButton" ).gameObject.SetActive(
!bHide );
      }
    }
```

Save the `GameInfo` class and go back to Unity.

As we wrote the `GameStateButtonPressed` function earlier, we can also use it for the `BackToMenu` button.

Select **BackToMenuButton** in **Hierarchy** under the **GameUI** GameObject. In **Inspector**, click on the small plus symbol to add an element to the **On Click ()** list.

Then, left-click and drag the **GameInfo** GameObject from **Hierarchy** onto the **On Click ()** object field for **BackToMenuButton**. In the drop-down list, navigate to **GameInfo** and select **GameStateButtonPressed**. Leave the checkbox that appears *unchecked*. We don't want this checked because we want to enter *false* so that the `bGameRunning` value changes to *false*. This will call the `SetIdlePieces` function and begin running the `ActiveLevelPieces` for the menu in `LevelPieceManager`.

Make sure that you *hide* the **GameUI** GameObject using the **Inspector** checkbox next to its name and *unhide* the **MenuUI** GameObject using the same checkbox before you run the game. If you do not do this, the first shown menu when the game starts will be of the GameUI, instead of the MenuUI.

You should now be able to click on **Play** button to start playing, **Restart** button to restart a game, and **BackToMenu** button to go back to the main menu.

# Code for leaderboard and achievement buttons

In the `Assets/Scripts` folder, double-click on the `GameInfo` class to open it. Under the `GameStateButtonPressed` function, add these two functions:

```
// Opens the leaderboards window
public void ShowLeaderboards()
{
  GameCenterManager.ShowLeaderboard( "G_Distance" );
}

// Opens the achievements window
public void ShowAchievements()
{
  GameCenterManager.ShowAchievements( );
}
```

Both of these functions are very simple. As `GameCenterManager` will manage most of the `GameCenter` features, it includes the `ShowLeaderboards` and `ShowAchievements` functions that will open their related windows.

Save the `GameInfo` class and go back to Unity.

Left-click on the **Leaderboard** button in **Hierarchy** under the **MenuUI** GameObject to select it. Search for **Inspector**. Now, in the **Button** component, click on the small plus symbol to add the **On Click ()** elements. Left-click and drag the **GameInfo** GameObject onto the object field of the **On Click ()** elements. Click on the drop-down menu that appears at the right-hand side, navigate to **GameInfo**, and select **Show Leaderboard**.

Left-click on **Achievement** Button in **Hierarchy** under the **MenuUI** GameObject to select it. Search for **Inspector**. Then, in the **Button** component, click on the small plus symbol to add the **On Click ()** elements. Left-click and drag the **GameInfo** GameObject onto the object field for the **On Click ()** elements. Click on the drop-down menu that appears at the right-hand side, navigate to **GameInfo**, and select **Show Achievements**.

This will allow the **Leaderboard** button and the **Achievement** button to open the windows so that the player can see the current state of each.

# Code for achievements

At this point, the final steps we need to take is to use the IOS Native plugin to call achievements, leaderboards, and store purchases.

If you don't remember, the three achievements we set up are:

- `10Rounds`
- `100Pickups`
- `100Yards`

This means that we need to begin storing how many rounds the player has played with our `PlayerPrefs` class. As we already know how many coins the player has, we don't need to save this, but we do need to have a condition so that we can award the achievement as soon as they collect 100 coins. Finally, we need to check how far the player has run every time we call the `AddDistance` function in the `Character` class to check whether they have reached 100 yards. If they have, we then can award the achievement.

In the `Assets/Scripts` folder, double-click on the `GameInfo` C# file to open it. At the top of the class, add the following `enum` list:

```
    public enum RunnerAchievements
    {
      RA_Rounds,
      RA_Pickups,
      RA_Yards,
    };
```

We will use this enum list to know which achievement to set the progress for or unlock.

Next, we need to set up `GameCenter` so that it is running and available. Add the `Awake` function so that it looks similar to the following code:

```
    // When object starts
    void Awake( )
    {
      if ( FadeObject != null )
      {
        FadeObject.transform.position = new Vector3( 0.5f, 0.5f, 0.0f );
        FadeTexture = FadeObject.GetComponent<GUITexture>( );
        FadeTexture.pixelInset = new Rect (0.0f, 0.0f, Screen.width,
Screen.height );

        RestartGame( );
      }

      // Register all achievements
      GameCenterManager.RegisterAchievement( "G_100Yards" );
      GameCenterManager.RegisterAchievement( "G_100Pickups" );
      GameCenterManager.RegisterAchievement( "G_10Rounds" );

      // Delegates for Achievements
      GameCenterManager.Dispatcher.addEventListener(
GameCenterManager.GAME_CENTER_ACHIEVEMENT_PROGRESS, OnAchievementProgress
```

```
);
      GameCenterManager.Dispatcher.addEventListener(
GameCenterManager.GAME_CENTER_ACHIEVEMENTS_RESET, OnAchievementsReset );

      // OnAchievementLoaded Delegate
      GameCenterManager.OnAchievementsLoaded += OnAchievementLoaded;

      // Init GameCenter
      GameCenterManager.init( );

      // DEBUGGING ONLY—REMOVE FOR FINAL BUILD
      //     Reset Achievements
      GameCenterManager.ResetAchievements( );
    }
```

## Tip

`GameCenterManager` is a part of the IOS Native plugin and is designed to help use call native code for the `GameCenter` features.

The new code starts by using `RegisterAchievement`, which takes in a string value. We do this so that we have access to these achievements, although they do not have any reported progress.

Next, we will do something that we haven't done yet. We will use the `addEventListener` function to add a delegate function. A **delegate** is a way to call a function when something else happens. In our case, we will use it when we get the progress back from an achievement and when we reset achievements.

We then use the `OnAchievementLoaded` delegate and assign it to `GameCenterManager.OnAchievementLoaded`. Delegates look a bit strange because they are assigned with the `+=` operator. What this will do is tell `GameCenterManager` that when its `OnAchievementLoaded` event is called to call the `OnAchievement` loaded function in our `GameInfo` class. This is the handle part of delegates; you don't have to check whether something has happened, but only assign what to do when it does happen.

After this, we can use the `Init` function for `GameCenterManager`. The `Init` function starts `GameCenterManager` so that the player gets logged in to `GameCenter`, or if they haven't logged in before, the `GameCenter` login screen will show for them.

Lastly, we will call the `ResetAchievements` function. This is purely for testing purposes. This means that every time you load the game, the achievements will be reset. This is *not* ideal for the end user and should only be used for us, so we can test that everything is working. If we didn't reset the achievements, we could only unlock them once. This would become an issue if we wanted to check whether they can be unlocked.

## Game center events

In order to know what is going on with the achievements, we need some functions that get called when specific aspects of the achievement system gets used. At the *bottom* of the `GameInfo` class, add the following function:

```
    // When achievements are loaded
```

```
    private void OnAchievementLoaded(ISN_Result Result)
    {
      if (Result.IsSucceeded)
      {
        foreach (AchievementTemplate template in
GameCenterManager.Achievements)
        {
          print( template.id + ": " + template.progress );
        }
      }
    }
```

This function will let us know when an achievement has been loaded. Right now, it only prints out what the achievement ID is and its progress. If you were making a more complex game with a custom UI, you could use this to enter the information of the achievement that is being loaded.

Beneath the `OnAchievementLoaded` function, add the following code:

```
    // When achievements are reset
    private void OnAchievementsReset()
    {
      // Achievements are reset
    }
```

Again, this is just another function that will be called when the achievements are reset. It doesn’t do anything yet, but I wanted to add it for you if you would like to use it.

Next, add the following code under `OnAchievementReset`:

```
    // When achievements send progress
    private void OnAchievementProgress(CEvent Event)
    {
      ISN_AchievementProgressResult result = Event.data as
ISN_AchievementProgressResult;

      if (result.IsSucceeded)
      {
        AchievementTemplate template = result.info;
        print( template.id + ": " + template.progress.ToString( ) );
      }
    }
```

Once again, this function is only here if you want to use it. This will give you the achievement progress about the event data. It is doing nothing here but printing out the progress and ID that is being sent when its progress has changed. The `CEvent` class as the function argument comes from the IOS Native plugin.

Finally, add the last of the event functions:

```
    // Player is logged into GameCenter
    void OnAuthFinished(ISN_Result res)
    {
      if (res.IsSucceeded)
      {
        IOSNativePopUpManager.showMessage("Player Authored ", "ID: " +
```

```
GameCenterManager.Player.PlayerId + "\n" + "Alias: " +
GameCenterManager.Player.Alias);
      }
      else
      {
        IOSNativePopUpManager.showMessage("Game Center ", "Player auth
failed");
      }
    }
```

If `ISN_Result` is successful, this function will let the player know that they have logged in to `GameCenter` by displaying a message to them that they have (while in the game). If it is not, it will tell the player that the `GameCenter` authentication has failed.

### SubmitAchievementProgress

Next, we need to add the functions that we will use to set our achievements for our game. Add the following function under `OnAchievementProgress`:

```
    // Set achievement for RunnerGame
    public void SubmitAchievementProgress(RunnerAchievements Achievement,
float AchievementValue )
    {
      string lastAchievement = "";
      switch (Achievement)
      {
        case RunnerAchievements.RA_Pickups:
          lastAchievement = "G_100Pickups";
            break;
        case RunnerAchievements.RA_Rounds:
          lastAchievement = "G_10Rounds";
            break;
        case RunnerAchievements.RA_Yards:
          lastAchievement = "G_100Yards";
            break;
      }


GameCenterManager.SubmitAchievement(GameCenterManager.GetAchievementProgres
s(lastAchievement) + AchievementValue, lastAchievement);
    }
```

As the `SubmitAchievement` function takes a string, we need to take the enum value we have for our achievements and use it to create a string of the string ID of the achievements we set up in iTunes Connect.

We can use a switch statement to see what enum value was passed into the function and then based on that, assign `LastAchievement` to the value of the string ID we set in iTunes Connect.

Based on the value of `LastAchievement`, we can use the `SubmitAchievement` function to submit the current progress value of the achievement *plus* the new value being passed. This will update the current progress of the achievement. If it reaches the value we set in iTunes Connect for when it unlocks, `GameCenter` will send the user a broadcast banner that

they have unlocked it.

### SubmitAchievementAsWhole

Because of the way we unlock the distance achievement, we need a way to reset it back to *zero* if the player failed to reach the 100 distance marker. As a result, we don't want the achievement to continue to count upwards each attempt until the distance of 100 has been traveled and instead will only unlock if the player reaches the distance of 100 in a single attempt.

Under the `SubmitAchievementProgress` function, add the following code:

```
    // Instead of adding to an achievement progress value, set it as a
whole
    public void SubmitAchievementAsWhole(RunnerAchievements Achievement,
float AchievementWholeValue)
    {
      string lastAchievement = "";
      switch (Achievement)
      {
        case RunnerAchievements.RA_Pickups:
          lastAchievement = "G_100Pickups";
            break;
        case RunnerAchievements.RA_Rounds:
          lastAchievement = "G_10Rounds";
            break;
        case RunnerAchievements.RA_Yards:
          lastAchievement = "G_100Yards";
            break;
      }

      GameCenterManager.SubmitAchievement(AchievementWholeValue,
lastAchievement);
    }
```

This function is very similar to the `SubmitAchievementProgress` function, but instead of adding the existing progress of the achievement, we set a whole value to it or a complete value. If there is ever a point when you wanted to submit an achievement for a single action or all at once, this is a way to handle it. For the current game, it will be used to either reset the `G_100Yards` achievement to *zero* or complete.

## SubmitLeaderboardScore

Next, we need a function to submit a leaderboard score. In the `GameInfo` class, under the `SubmitAchievementAsWhole` function, add the following code:

```
    // Submits a leaderboard score
    public void SubmitLeaderboardScore(int DistanceScore)
    {
      GameCenterManager.ReportScore( DistanceScore, "G_Distance" );
    }
```

This will submit an integer score to the leaderboard with the ID of `G_Distance`, which is our only leaderboard.

## The character achievement code

We now need to use the `Character` class to both keep track of the data we need, such as `AttemptCount`, and call the `GameInfo` achievement functions in order to update them.

At the top of the `Character` class, add the following code:

```
    [System.NonSerialized]
    public int AttemptCount;
```

This will be used to keep a tally on how many times the player has played the game.

Next, we need to save `AttemptCount` with the `PlayerPrefs` class. Change the `Start` function so that it looks similar to the following code:

```
  // Use this for initialization
  void Start ()
  {
    RestartLocation = gameObject.transform.position;

    AttemptCount = PlayerPrefs.GetInt( "Attempts" );
    CoinCount = PlayerPrefs.GetInt( "Coins" );
    AddCoins( 0 );
  }
```

We set the `AttemptCount` to be set each time the character is loaded so that it is equal to what has been saved in `PlayerPrefs`. This way, when we send it to the `SubmitAchievement` function, it will get updated from the correct `AttemptCount` value.

The last thing we need to do is to add the portion of code that will send the achievement data when the player is being revived.

Change the `KillCharacter` function as follows:

```
    // Kills the character
    public void KillCharacter()
    {
      Rigidbody2D characterRigidBody;

      if (!isDead)
      {
        characterRigidBody = gameObject.GetComponent<Rigidbody2D>();
        if (characterRigidBody != null)
        {
          characterRigidBody.AddForce(new Vector2(Random.Range(-1, 1), 1) *
512);
          isDead = true;
          isFadeOut = true;

          if (Game != null)
          {
            Game.HideRestartButton(false);
          }

          PlayerPrefs.SetInt("Coins", CoinCount);
          PlayerPrefs.Save();
```

```
          // Save attempts
          AttemptCount += 1;
          PlayerPrefs.SetInt("Attempts", AttemptCount);
          PlayerPrefs.Save();

          if (Game != null)
          {

Game.SubmitAchievementProgress(GameInfo.RunnerAchievements.RA_Rounds,
AttemptCount);

Game.SubmitAchievementAsWhole(GameInfo.RunnerAchievements.RA_Pickups,
CoinCount);

            // If DistanceCount is less than 100 AND achievement hasn't
been unlocked, reset it
            if (GameCenterManager.GetAchievementProgress("G_100Yards") <
100.0f && DistanceCount < 100)
            {

Game.SubmitAchievementAsWhole(GameInfo.RunnerAchievements.RA_Yards, 0.0f);
            }
            // If DistanceCount is more than 100, set achievement as
complete
            else
            {

Game.SubmitAchievementProgress(GameInfo.RunnerAchievements.RA_Yards,
DistanceCount);
            }
            Game.SubmitLeaderboardScore( DistanceCount );
          }
        }
      }
    }
```

This moves on to what we already had at `AttemptCount += 1`. From here, we set `AttempCount` to `PlayerPrefs` using the `SetInt` function and then use the `Save` function to save this value.

We then check to make sure that the reference to `Game` exists before we submit the achievement progress using the `SubmitAchievementProgress` function. We do this for `AttemptCount`. We then use `SubmitAchievementAsWhole` for `CoinCount` because this progress will be a whole value from what is saved using `PlayerPrefs`.

Before we set the distance achievement, we want to make sure that the progress of the distance achievement is less than *one hundred* and that the player had reached a `DistanceCount` of *one hundred*. If they didn't or if the achievement hasn't been unlocked, we reset its progress to *zero*. This way, if the player starts again, they won't be able to continue only from where they failed.

If the player did reach the *one hundred* distance, we can use the `SubmitAchievementAsWhole` function to set the progress to the value of `DistanceCount`,

which will be at least *one hundred*, meaning they unlocked the achievement.

We can also call `SubmitLeaderboardScore` that passes through `DistanceCount`, so the leaderboard will reflect how far the player has gone as a record.

This is it for the achievement code. Make sure to save the `Character` and `GameInfo` classes.

# Displaying iAds

As we want to display ads only to those who have not purchased to remove them, we need another value to save the `PlayerPrefs` class. In the `Assets/Scripts` folder, open the `GameInfo` class by double-clicking on it.

At the bottom of the list of global variables, add the following code:

```
// If HideiAds has been unlocked
private int HideiAds;

// Reference to iAd banner
private iAdBanner AdBanner;
```

As the `PlayerPrefs` class doesn't store bool values, we have to store it as an int. We will use the default of *zero* to know that the player is going to see ads. Then, if they buy to remove them, we will save the `HideiAds` value to *one*.

We also need to keep the reference to the created `AdBanner` so that we can hide and show it, depending on the `bGameRunning` value. If the `bGameRunning` value is set to `true`, we will hide the iAds so that it doesn't get in the way of **GameUI**.

At the bottom of the `GameInfo` class, write the following function:

```
// Creates instance of iAd if doesn't exist
// Show or hide it depending on bShow
public void ShowIAds(bool bShow)
{
  HideiAds = PlayerPrefs.GetInt( "ShowiAds" );
  if (HideiAds == 0)
  {
    if (AdBanner == null)
    {
      if (bShow)
      {
        AdBanner =
iAdBannerController.instance.CreateAdBanner(TextAnchor.UpperCenter);
        AdBanner.Show();
        AdBanner.AdViewFinishedAction += BannerLoaded;
      }
    }
    else
    {
      if (bShow)
      {
        AdBanner.ShowOnLoad = true;
        AdBanner.Show();
      }
      else
      {
        AdBanner.ShowOnLoad = false;
        AdBanner.Hide();
      }
    }
  }
```

```
}
```

We start this function by assigning the value of `HideiAds` to the `PlayerPrefs` saved valued of it. If it's set to *zero*, when check whether the `AdBanner` reference is null. We know that if it's *null*, we need to create it, which is what the code does if the banner is null and the bool value being passed is set to *true*. Once it's created, we show it using the `Show` function. We also set a delegate so that we know when it's loaded. This is useful, so we can check whether the game is running when the banner has loaded. If the game is running, we can use the `BannerLoaded` function to hide it.

If `AdBanner` exists, we then check whether the bool value is set to `true` or `false`. If it's set to `true`, we show the banner. If it' set to `false`, we hide it.

Add the following function *before* the `ShowIAds` function:

```
// Banner has loaded
public void BannerLoaded()
{
  if (bGameRunning)
  {
    AdBanner.Hide( );
  }
}
```

This is a very simple function that will be called when `AdBanner` has been loaded. If the game is running, we hide it so that it doesn't get in the way of the gameplay.

We now need to add the `GameStateButtonPressed` function so that it shows or hides the banner based on the value of `bGameRunning`. Change the `GameStateButtonPressed` function as follows:

```
// Game state button was pressed
public void GameStateButtonPressed( bool bRunning )
{
  bGameRunning = bRunning;

  if (MainMenuUI != null)
  {
    MainMenuUI.gameObject.SetActive(!bRunning);
  }

  if (GameUI != null)
  {
    GameUI.gameObject.SetActive(bRunning);
  }

  ShowIAds( !bGameRunning );

  RestartGame( );
}
```

The addition here is the `ShowIAds` function, which passes the opposite value of `bGameRunning`. More simply, it only shows the banner if `bGameRunning` is set to `false`.

Now, at the bottom of the `Awake` function, add the following code:

```
ShowIAds( true );
```

This will try to show the `AdBanner` object as soon as the `GameInfo` class starts running. It will fail if the `HideiAds` value is not set to *zero*.

Save the `GameInfo` class.

## Purchasing remove iAds

The first thing we need to do is finish setting up the iTunes Connect in-app purchases settings. Perform the following steps:

1. Navigate to https://itunesconnect.apple.com.
2. Sign in using your developer account.
3. Then, go to **My Apps**.
4. Select your app, the one we created in Chapter 1, *Requirements and Preparation Work*.
5. Now, click on **In-App Purchases.**
6. It should have a red circle that says **Waiting for Screenshot**.
7. Click on **Edit** next to **In-App Purchase Details**.
8. Make sure that **No** is selected for **Hosting Content With Apple**.
9. You do *not* need to include **Review Notes**.
10. Now, select **File**.
11. Navigate to the book `Art` files and search for the `ChapterSix_InApp` folder.
12. Select `RemoveIAds.png`. (This will only work if you plan on using the UI we created. If you create your own game or your own UI art, you will need to include a screenshot for this.)
13. Go back to **My Apps** and select your game again.
14. Scroll down to the **In-App Purchases** section and click on the plus symbol to add a purchase.
15. Select our **G_RemoveAds** purchase.
16. Then, click on **Save** in the top-right corner.
17. Now, scroll to the top of the web page and click on the **My Apps** drop-down list and then select **Users and Roles**.
18. Then, click on **Sandbox Testers** on the **Users and Roles** page.
19. Now, click on the small plus sign next to **Testers** to add one.
20. Enter the relevant information on this page. The e-mail has to be something other than what you normally use. This is a bit of an annoyance, but this is how the testing needs to be set up (as of writing this book).
21. When on your iOS device, make sure to sign out of your normal account and use this testing account for testing purposes.

## New RemoveiAds Purchase for coins

In order for us to remove iAds for the cost of coins, we need to create another in-app purchase. Perform the following steps:

1. In **iTunes Connect**, navigate to **My Apps** and click on your game.

2. Now, click on the **In-App Purchases** section at the top of the page.
3. Then, click on **Add New** on the **In-App Purchases** page.
4. Select **Non-Consumable**.
5. Then, select **Reference Nam**e as **Remove_Ads_Coins**.
6. Similarly, select **Product ID** as **G_RemoveAdsCoins**.
7. Then, select **Cleared For Sale** as **No**.
8. Now, select **Price Tier** as **Free**.
9. Then, select **Screenshot For Review** as `ChapterSix_InApp`, `Remove IAds.png`, or what's suitable for you.
10. Select **Add Language** as **English** or whatever suits you.
11. Select **Hosting Content With Apple** as **No**.
12. Also, select **Review Notes** as **None**.
13. Finally, click on **Done** to save this.

We want this to be free because we will check to see how many coins the player has before purchasing. We also want the player to keep the purchase if they did use coins to buy it.

## The In App Purchase code

Go back to Unity. In the `Assets/Scripts` folder, double-click on the `GameInfo` class to open it. In the `Awake` function, before the `GameCenterManager` code, but under the `FadeObject` code, add the following code:

```
// RemoveAds as Product
IOSInAppPurchaseManager.instance.addProductId( "G_RemoveAds" );
IOSInAppPurchaseManager.instance.addProductId( "G_RemoveAdsCoins" );

// Add listening for Delegates
IOSInAppPurchaseManager.instance.addEventListener(
IOSInAppPurchaseManager.RESTORE_TRANSACTION_FAILED,
OnRestoreTransactionFailed );
IOSInAppPurchaseManager.instance.addEventListener(
IOSInAppPurchaseManager.VERIFICATION_RESPONSE, OnVerificationResponse );

// Assign delegates
IOSInAppPurchaseManager.instance.OnStoreKitInitComplete +=
OnStoreKitComplete;
IOSInAppPurchaseManager.instance.OnTransactionComplete +=
OnTransactionComplete;

// Load the store.
IOSInAppPurchaseManager.instance.loadStore( );
```

This is similar to what we did with the `GameCenterManager` setup. The only difference this time is that it's specific for `IOSInAppPurchaseManager`.

We start by adding a product ID, using `addProductID`, which takes the name of our purchase: `G_RemoveAds`.

We will then set up a couple of event listeners. This will be used for delegate calls when

events related to `InAppPurchase` are called.

Let's assign a couple of more delegates to `OnStoreKitInitComplete` and `OnTransactionComplete`.

Finally, we will load the store using the `loadStore` function, which prepares the store to be used.

Next, we need to add the function calls that are used for the delegates. At the *bottom* of the `GameInfo` class, under the `ShowIAds` function, start by adding this function:

```
    // Transaction Complete
    private static void OnTransactionComplete(IOSStoreKitResponse Response)
    {
      GameInfo game = FindObjectOfType<GameInfo>();
      switch (Response.state)
      {
        case InAppPurchaseState.Purchased:
        case InAppPurchaseState.Restored:
          game.ShowSuccessScreen( true );
          game.HaveUnlockedIAds( Response.productIdentifier ==
"G_RemoveAdsCoins", Response.state == InAppPurchaseState.Restored );
            break;
        case InAppPurchaseState.Deferred:
          break;
        case InAppPurchaseState.Failed:
          game.ShowFailScreen( true );
            break;
      }
    }
```

This is called when a transaction has been completed, including if the transaction fails for any reason. We start using this by getting a reference to the `GameInfo` class by using the `FindObjectOfType` function. We do this because this function is a **static** function, meaning it is usable even if there is no reference to the class instance. Due of this, the static function sort of lives in its own portion of code and doesn't know of the reference of the `GameInfo`, although it exists within it, which is why we need a reference to it.

From there, we use a switch statement and check the `Response.state`. If the state is `Purchased` or `Restored`, we show the success screen with `ShowSuccessScreen` and then call a function in `GameInfo` called `HaveUnlockIAds` and pass `productIdentifier` as *equal* to `G_RemoveAdsCoins`. This means that if the product ID of the purchase is from coins, we know that because the bool value passed `HaveUnlockedIAds`, it will be set to *true*. This is the same for if the response state is *equal* to `Restored`.

The next three functions are delegates used by the store manager, but for our game, we do not need them. Just in case you do, use the following code:

```
    // Verification response
    private static void OnVerificationResponse(CEvent e)
    {
      // Verified response on purchase
    }
```

```
// Transaction failed
private static void OnRestoreTransactionFailed()
{
  // Transaction failed
}

// StoreKit Init
private static void OnStoreKitComplete(ISN_Result result)
{
  // Store was completed Init
  if (result.IsSucceeded)
  {
    // succeeded
  }
  else
  {
    // failed
  }
}
```

`OnVerificationResponse` is used when a store purchased has been verified.

`OnRestoreTransactionFailed` is used when a restore transaction has failed; we will use our `OnTransactionComplete` function to handle restore purchases.

`OnStoreKitComplete` is used when the store has been initialized.

## The Purchase RemoveIAds functions

Under `OnStoreKitComplete`, add the following function:

```
// Purchase was successful, remove coins, and hide ad banner
public void HaveUnlockedIAds(bool bForCoins, bool bRestored)
{
  if (bForCoins && !bRestored)
  {
    GameCharacter.AddCoins(-10000);
  }

  PlayerPrefs.SetInt("ShowiAds", 1);
  if (AdBanner != null)
  {
    AdBanner.Hide();
    AdBanner = null;
  }
}
```

This is the function we will call from `OnTransactionComplete` if the purchase succeeded or the restore purchase succeeded.

Let’s start by checking whether the `bForCoins` argument is set to *true* and `bRestored` is set to *false* because we don’t want to dock the player if the purchase is being restored. If it is, we subtract the *ten thousand* coin cost from the character using `AddCoins`. If we pass a negative value into it, it will subtract the amount.

Next, we will set the `ShowIAds` int value, which we save with `PlayerPrefs` to *one*. This

way, `AdBanner` won't show anymore. Then, we need to check whether the `AdBanner` exists by checking it *against* null. If it does, we hide it using the Hide function and then set the `AdBanner` reference to null.

Lastly, we need a function that gets called from **MenuUI** when the player taps a button to buy something. Add the following code under the `HaveUnlockedIAds` function:

```
// Tries to purchase RemoveiAds
public void UnlockRemoveAds(bool bForCoins)
{
  if (bForCoins)
  {
    if ( GameCharacter.CoinCount >= 10000)
    {
      IOSInAppPurchaseManager.instance.buyProduct("G_RemoveAdsCoins");
      ShowPurchaseScreen(false);
    }
  }
  else
  {
    IOSInAppPurchaseManager.instance.buyProduct("G_RemoveAds");
    ShowPurchaseScreen(false);
  }
}
```

If the purchase is for coins, we then check whether the character has *ten thousand* or more coins. If they do, we can use `IOSInAppPurchaseManager` to buy the product with `buyProduct`. We also pass the product ID for the remove ads with coins. We then close the purchase screen so that it doesn't stay open during the purchase process. If the player is trying to pay with coins, but doesn't have enough coins, the window will simply stay open and nothing will happen.

If the purchase is *not* for coins, we will again use `IOSInAppPurchaseManager` to buy a product, but we can pass the product that costs *ninety nine* cents. Again, close the purchase screen.

Now, save the `GameInfo` class and go back to Unity.

## Purchasing buttons

If `RemoveAdsBackgroundScreen` is hidden, select it in **Hierarchy** and use the **Inspector** window to unhide it by clicking on the small checkbox next to its name.

Select the button called **For Coins** and search for **Inspector**. In the **Button** component, click on the small plus symbol to add an element to the **On Click ()** list. Then, left-click and drag the **GameInfo** GameObject onto the object field of the **On Click ()** list. Click on the drop-down menu, navigate to **GameInfo**, and select **UnlockRemoveAds.** Then, *check* the small checkbox under the drop-down list because this purchase is for coins.

Next, select the button called **For Cash** and repeat the preceding steps. This time, do *not* click on the small checkbox under the **On Click ()** drop-down list because this purchase is for money.

## Restoring purchases

Open the `Assets/Scripts` folder and double-click on the `GameInfo` class to open it. At the *bottom* of the class, add the following function:

```
// Attempts to restore purchases
public void RestorePurchases()
{
  IOSInAppPurchaseManager.instance.restorePurchases( );
}
```

This will attempt to restore purchases and then call `OnTransactionComplete` so that we can handle how to use it.

## Tip

Apple requires you to have an extra button in the game to restore purchases. You cannot simply do this on purchase; they will not allow the game on the iTunes store if you do not have this extra **Restore Purchases** button.

Now, save the `GameInfo` class and go back to Unity.

Select the **RestorePurchase** button in **MenuUI** and search for Inspector. In the Button component, click on the small plus symbol to add the **On Click ()** list. Left-click and drag the **GameInfo** GameObject onto the **On Click ()** object field and then click on the drop-down list. Now, select **GameInfo** and then **RestorePurchase**.

Then, save the Unity scene.

# Summary

This has been a long chapter, but it has covered the basics of how to create a game for iOS and utilize its `GameCenter` features. Feel free to explore these features and add to them. Adding more store purchases, achievements, and leaderboards is simply repeating the steps that we have already done.

When you get used to the workflow, the system can be set up to be fairly easy to manipulate to do what you want to do.

In the next chapter, we will look at how to create builds with Xcode on your Macintosh computer. This way, you will be able to run the game and test the features that we have completed.

# Chapter 7. Game Builds and Submission to Apple

The last thing we need to do is create a build to test our game and then make a release build that we can submit to Apple for approval.

A few important things need to be included to make sure that our build will be submitted. The most important thing is that any application for the iOS / Apple Store needs to be compiled on a Macintosh computer with Xcode. You will not be able to submit the game without a Macintosh computer and Xcode. This is a requirement from Apple, and there isn't any solution to get around it. You also need the newest version of Xcode, which comes with the current version of OS X.

If you do not own a Macintosh or do not have access to one, you will not be able to submit your game to Apple. This restriction can be a bit frustrating and expensive, but it is part of the cost when you create games for iOS.

The first step is to download the most recent Xcode version. If you already have Xcode, you can get any updates through the Application Manager on your Macintosh machine. If you do not have Xcode, you can also use the Application Manager to download it. As of now, the most recent version of Xcode is version 6.4 and is free of charge for any OS X Yosemite user.

The next step is to create the build for the game. Although it is possible to create a build on a Windows machine and transfer files to a Macintosh, it requires several extra steps and many more solutions to problems. It is more easily solved by simply building the game on a Macintosh. Although the steps to create the build are the same for Windows and Macintosh, the transferring process creates many unique and different challenges that is not possible for me to plan for. Therefore, I cannot help you with it.

If you are able to download and install Unity 5 on your Macintosh machine, transfer the Unity project folder to your Macintosh machine before you create any build. I will use **GitHub** to keep the version of my games. This makes the transferring process very easy because I am able to download the same files on my Windows and Macintosh machine.

GitHub is a version control tool that is exceptionally easy to use and set up. There are many tools that can help you use GitHub; I recommend you to use a program called Source Tree, which can be found at https://www.sourcetreeapp.com/.

# Setting up Unity for iOS builds

After you have your game files transferred, or if are already working on a Mac, open Unity and then your game scene if it is not already loaded.

Right-click on the `Assets` folder in the **Project** window. Then, select **Create** and then **Folder**. Name this folder `IconsAndSplash`.

Right-click on the `IconsAndSplash` folder and select **Import New Asset…**. Navigate to the book files and search for the folder named `ChapterSeven_iOSIcons_Splash`. Import the files, *except* the one named `Icon_Template.png`. This is here so that you have a reference to what the iOS icon sizing should be if you are creating a custom icon, which I recommend because the one I am including here is not specific to any game or product.

Once the icons and the splash screen are imported, select the icons; icons are the ones with `numberXnumber` as its name. In the **Inspector** window, change the settings for the icons as follows:

- Select **Filter Mode** as **Point**
- Select **Max Size** as `256`
- Select **Format** as **Truecolor**

Next, select the `Splash_Screen.png` file and back in the **Inspector** window, use the following settings:

- Select **Filter Mode** as **Point**
- Select **Max Size** as `2048`
- Select **Format** as **Truecolor**

Apple products have different screen sizes and resolutions, which is why there are so many icons. Unity does have it built-in, so you can import just the 180x180 image. Also, it will match the other resolutions in the list. I included all the sizes so that you can both know them and manipulate and customize them based on icon size if you want.

Perform the following steps:

1. Left-click on **Edit** at the top selection bar of the editor, navigate to **Project Settings**, and select **Player** from the list.
2. In the **Inspector** window, change **Company Name** and **Product Name** to what fits your project. For example, I will use `dotVawxGames` for the company name and `Endless Rush Runner` for the product name.
3. Now, left-click on **Select** for the default icon and search for the `180x180.png` image that was imported and left-click on it to select it.
4. Then, left-click on the **Resolution and Presentation** tab. Now, under **Default Orientation**, select **Landscape Right**.
5. Now, let's left-click on the **Icon** tab and select **Override for iPhone.**
6. Then, click on the **Select** tab for the **180x180** option and search for the `180x180.png` file we imported. With the **Override for iPhone** box selected, it should fill the rest of the boxes for you. If you want to have your own unique icon images for all the sizes,

use the rest of the PNG files to enter the selections.

7. Next, left-click on the **Splash Image** tab, and for the image fields, click on **Select** and then the **Splash_Screen.png** file that was imported:

8. Finally, left-click on the **Other Settings** tab. Now, under **Configuration**, make sure that **Scripting Backend** is set to **IL2CPP**.

   IL2CPP is now the standard to create and submit builds to Apple. If you use **Mono(2.x)**, the build will fail to submit because Apple requires iOS Apps to support the 64-bit iOS platform.

9. Click on **File** in the top-left corner of the editor and then select **Save Project**.

## Tip

If you were to use your own art, the fields in the **Icon** and **Splash Image** fields are what you'd fill.

# Creating a test build

Now that we are ready to create a build, left-click on **File** and navigate to **Build Settings**. Make sure that your **Build Settings** screen looks similar to the following screenshot:

All of these settings are important.

**Scenes In Build** is the scenes that you want to include in your game. In our case, we only have the one, so this is what we include. If the scene is not in the **Build Settings** list and you have it open, simply click on **Add Current**. If you do *not* have the scene open, open it first and then go back to **Build Settings** and click on **Add Current**.

The **Platform** is also important. As we will create an iOS game, the engine will compile assets in a specific way to match the platform. If you are *not* on the right platform, select iOS from the list and then select **Switch Platform.** This will take a minute to convert your game assets to the iOS project.

**Development Build** is just that. It is a setting we can use to make sure that the game is build for development and allows you to test the game in **sandbox mode**, which allows special cases for achievements, leaderboards, and in app purchases for Game Center.

Next, click on **Build**. Save the build in a location that you will remember. I generally have a “builds” folder that is stored so that I can keep track of specific versions if I need to go back.

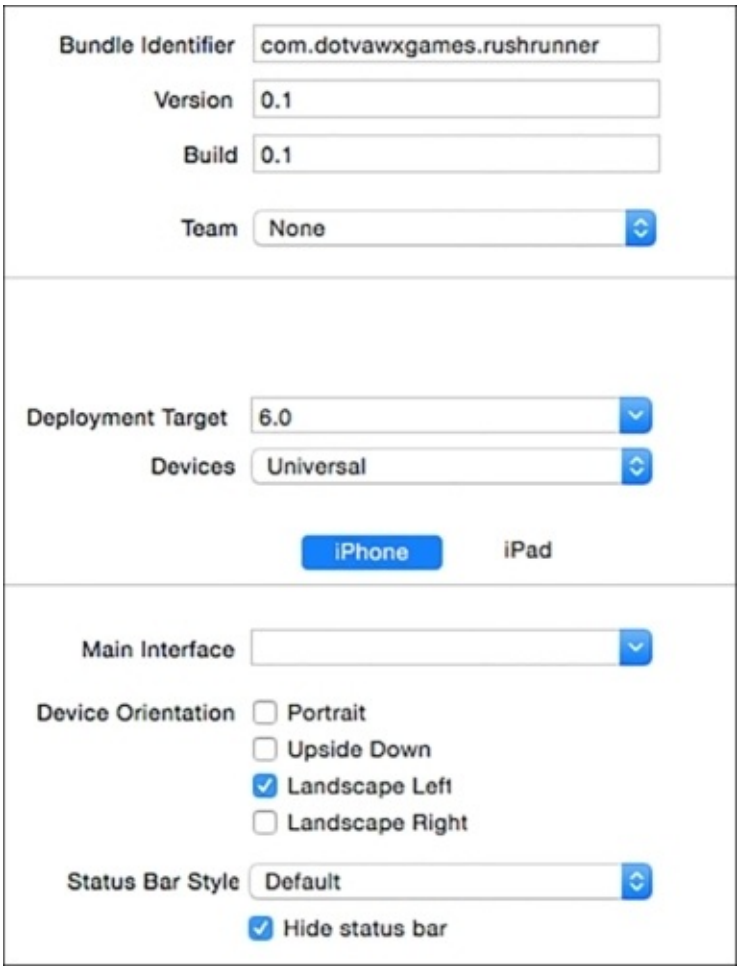
## Tip

If you do not have your iPad, iPhone, or iPod plugged into your Mac via USB, do so now. You can close any programs that open when you do plug it in, including iTunes.

Navigate to the build folder you just created on your machine and double-click on it to open the folder. At the bottom of the folder, there is a file named `Unity-iPhone.xcodeproj`. Double-click on it to open it, which will open Xcode in the process, as shown in the following screenshot:

In the top-left corner of the Xcode project, you will find a **Unity-iPhone** project listed. Click on it to select it.

You should see a window similar to the following screenshot:



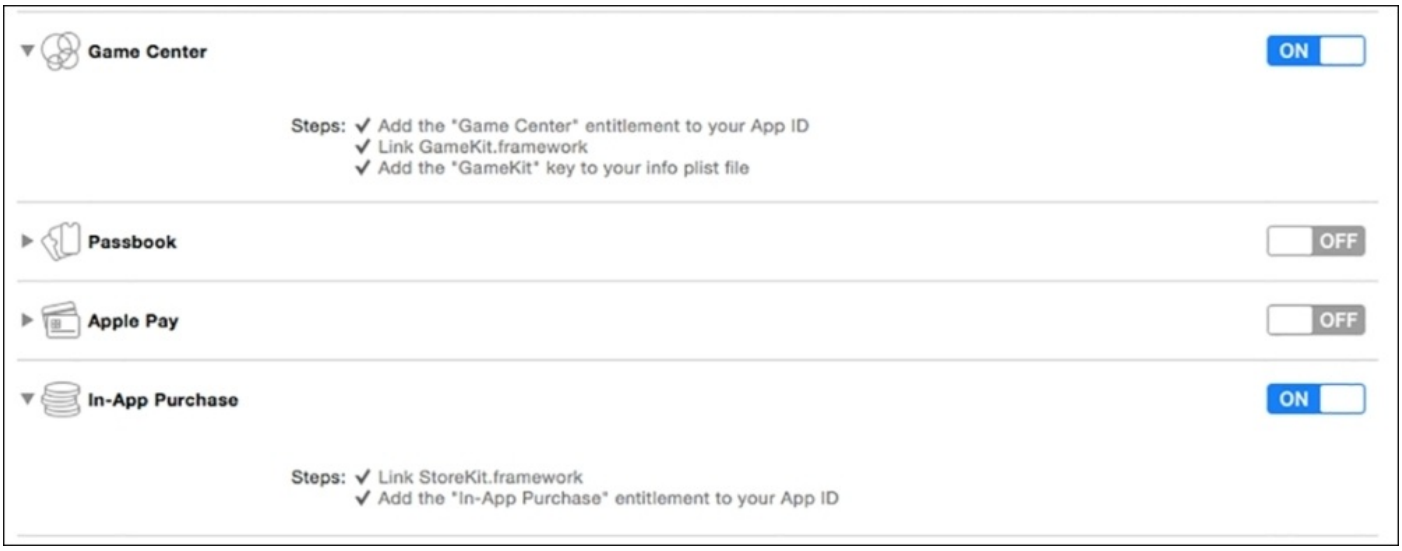

The **Build Identifier** field will be unique to what you had set in the Unity settings, **Version**, and **Build**.

Select the **Team** drop-down menu. If it asks you to log in and you already have a login save, select it.

If your device orientation is not set to as it is in the image, set the following parameters:

- Select **Portrait** as **OFF**
- Select **Upside Down** as **OFF**
- Select **Landscape Left** as **ON**
- Select **Landscape Right** as **OFF**

Next, click on the **Capabilities** text at the top of the Xcode project and make sure that it looks similar to the following screenshot:




It is very important that you have both **Game Center** and **In-App Purchase ON**. If you don't have one of these sets, at best, the project will compile, but it won't work, or you will get a compiling error because we intend to use them.

When you click on the **Game Center** capability, you will likely be asked to log in. Select the login that is in the list, or if there is not one saved, log in to one and then select it.

You should now be able to click on the small play button in the top-left corner of Xcode, and the project will compile and then automatically deploy to the device you have plugged in. The game will also automatically launch on your device. You can now test the game.

When you are done playing on the device, click on the stop button next to the play button to stop Xcode from running the project.

# Making a release build

To start, close out of Xcode.

If you have not already, navigate to the `Assets/Scripts` folder (you can change this freely on your Mac too if you are working on it now) and double-click on the `GameInfo.cs` file to open it. In the Start function, remove the following code:

```
// DEBUGGING ONLY—REMOVE FOR FINAL BUILD
//     Reset Achievements
GameCenterManager.ResetAchievements( );
```

This will make sure that the achievements are reset every time the player loads the game. If you have to change this, make sure to save the `GameInfo` file and go back to Unity so that it compiles.

To create a release build, the first step to do is navigate to the **File | Build Settings** and make sure to *uncheck* the **Development** checkbox.

Then, left-click on **File** and select **Save Project**.

Now, click on the **Build** button, *not* build and run. If you are planning on saving this project in the same location as the test build, go ahead, but make sure to click on **Replace** when the window asks you to.

When the build is done, again navigate to the build folder and double-click on the `Unity-iPhone.xproj` file.

Back in Xcode, again click on the Unity-iPhone project in the top-left corner. Now, on the main screen, change **Version** to 1.0. You can leave **Build** as 0.1.

You will also want to check the **Capabilities** area to make sure that both **Game Center** and **In-App Purchase** are *on* and have the correct account attached.

Now, in order to make a build that we can upload, which is in the format called **IPA**, navigate to the top menu of Xcode and click on **Product** and then select **Archive**.

## Tip

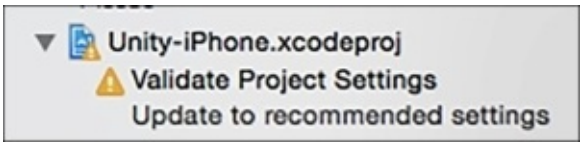
The IPA file format is a bit like a ZIP file. It stores all the relevant information the app needs to run. In our case, it will be art assets, code we wrote, and what Unity uses, and so on.

There are a lot of things that could happen here. I want to give you all the issues that I experienced, so if you experience them, you can fix them too.

The first issue is you get an error that tells you that the app needs to be built in x64 bit.

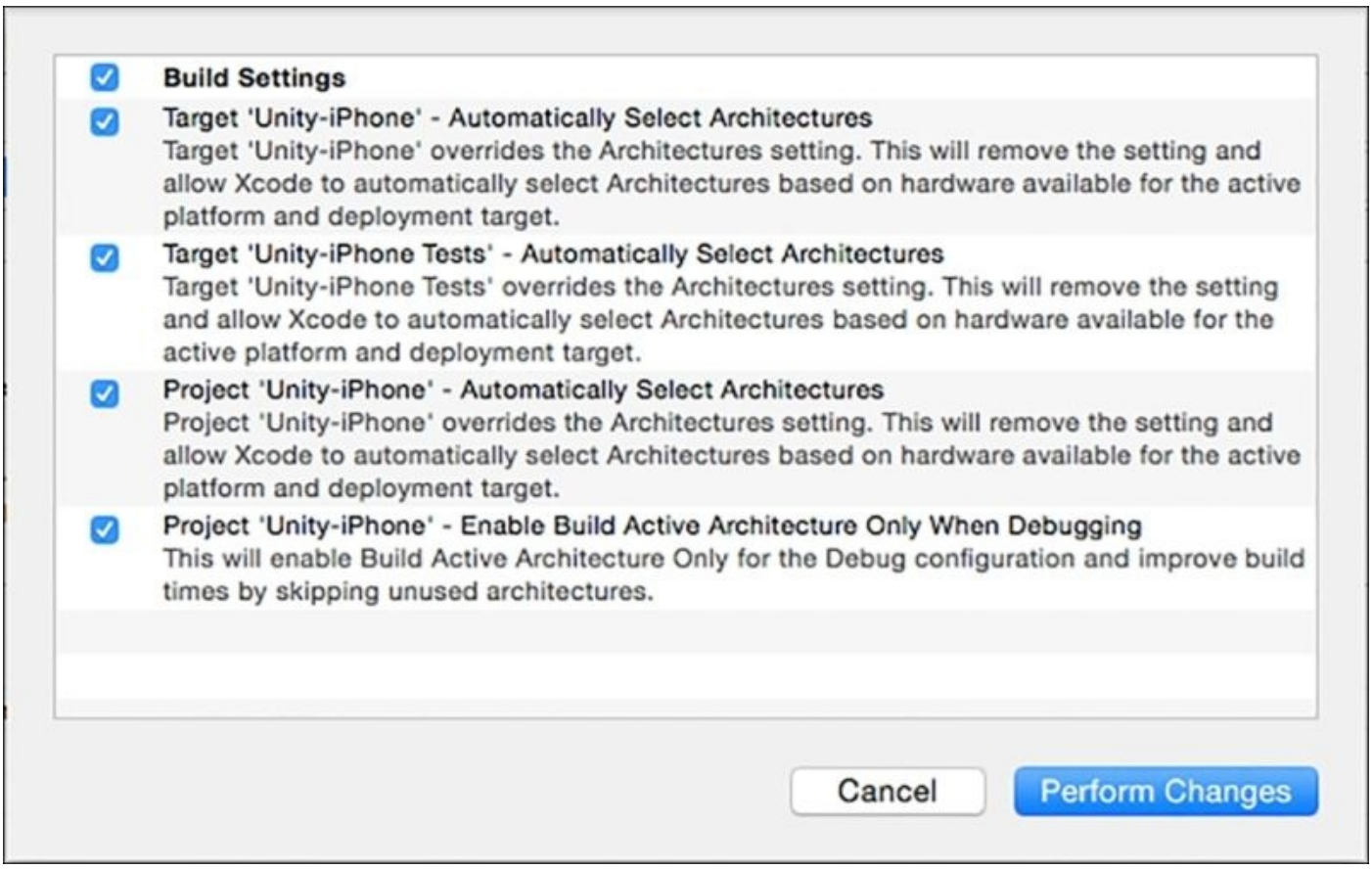
This is solved by making sure that in the Player Settings option, you select **IL2CPP.** If you see this, make sure to change the setting in Unity, save the project and then create another build.

The second issue could be that you get a warning that looks similar to the following screenshot:




In order to fix this, left-click on the part of the warning that says **Validate Project Settings**.

This will open another window that looks similar to the following screenshot:




Click on Perform Changes and then Enable when the next window opens. What is going on here is that Xcode noticed that there are issues with the build. Also, it is offering to fix them for you.

You will have to recompile the project once you change these settings by clicking on **Perform Changes**. To do this, again click on **Product** and then select **Archive**. This will recompile the project and also create another IPA if you already have one.

## Tip

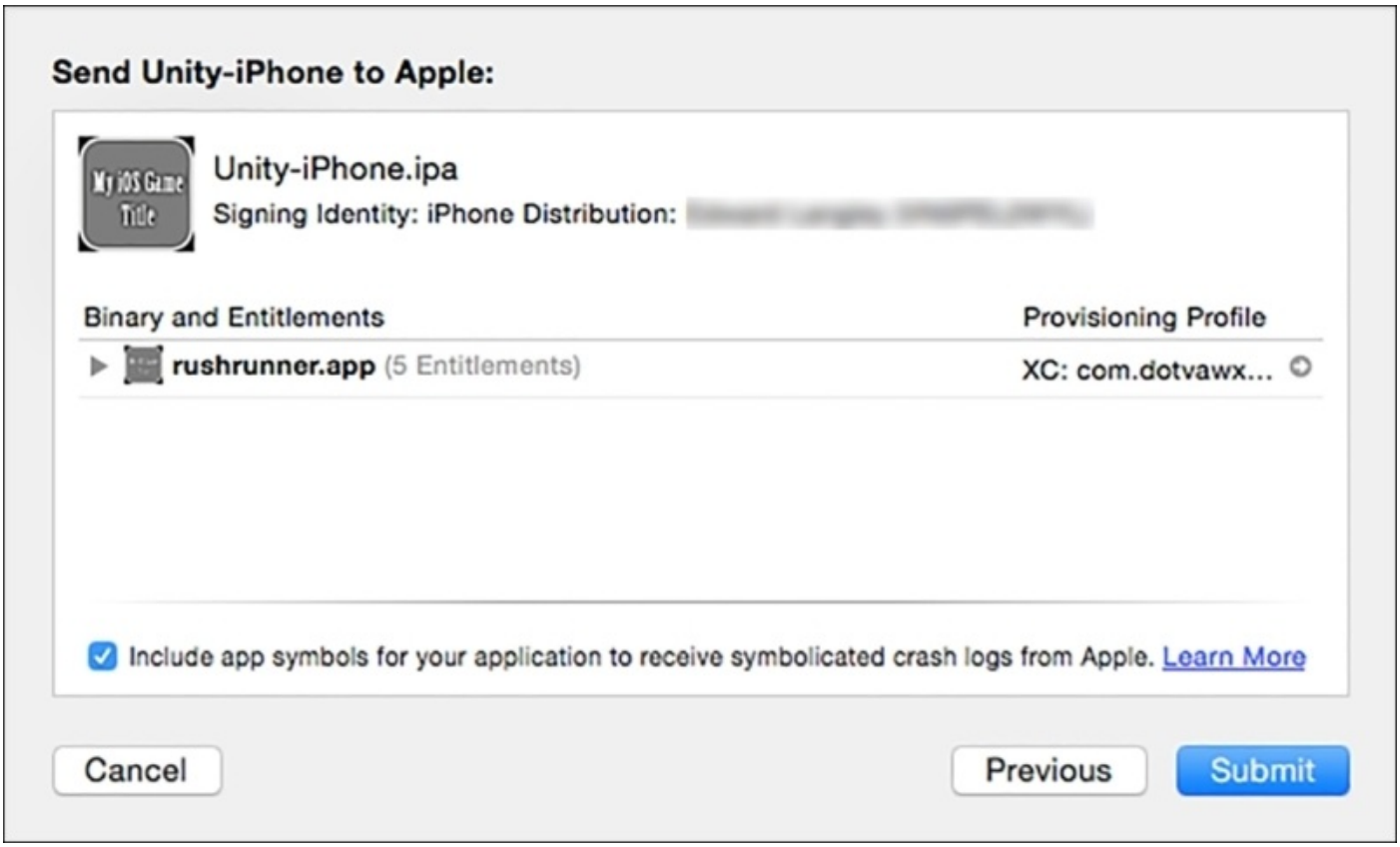
These are two issues I ran into. If you end up experiencing something different, the Internet is full of support. Also, an answer is often as simple as a Google search away. Part of creating apps and games is fixing the issues that are somewhat unique to them.

# Submitting the game to Apple

When the archive process is complete, it will open the archives window. From here, you will be able to select the IPA you want to submit. Before I commit, I usually click on the Validate button. This will run the app through a process that makes sure that it is acceptable. If it is, it will say successful. If is it not, you will see an error as to why.

If you want to validate, do so now. Alternatively, if you want to submit, make sure that the correct IPA is selected and then left-click on **Submit**:

You will see a window like:




Again, click on **Submit**.

This will take some time. It has to verify the build and then upload it to Apple. Depending on your Internet connection speed, it could take more than ten minutes.

If all is well, you will be presented with a success screen, and iTunes will send you a e-mail informing you that the build was accepted. It will also offer any suggestions for settings if something is not as they expect.

## Tip

In my e-mail, it told me Push Notifications were disabled, but the game had settings that could use them. It also mentioned that if I want to use them, I need to update my app settings to allow push notifications. If you see this and do not care, do not worry. Your app will still be accepted and will work fine.

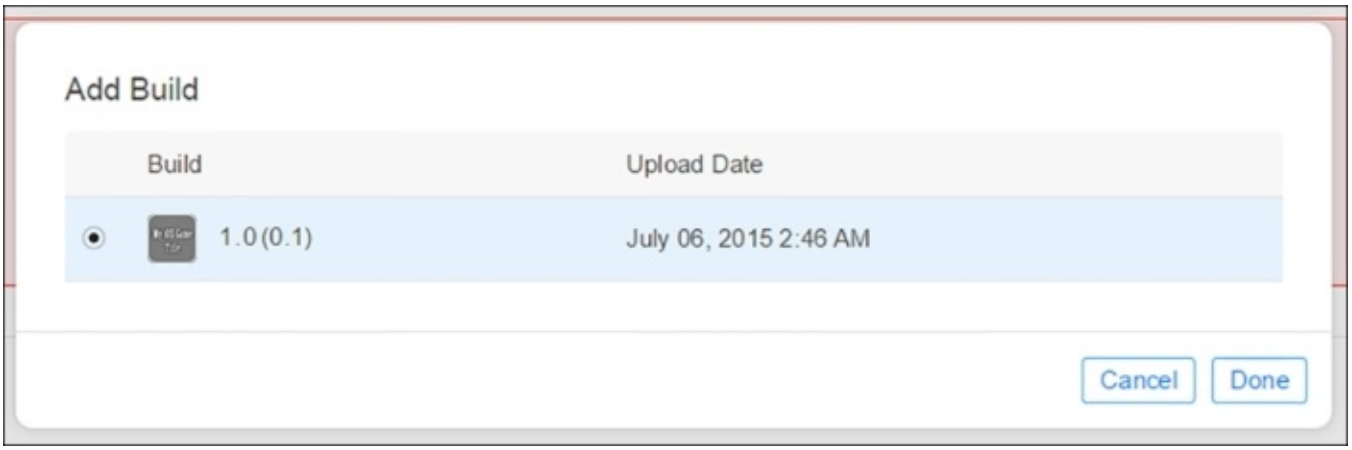
The final step is to log in to **iTunes Connect** and then click on **My Apps**.

First, add at least one screen shot of the game; you must submit at least *one* for the screen shot option. You should also enter the **Description**, **Keywords**, **Support URL**, **Marketing URL**, and **Private Policy URL** parameters if not already done.

You must include an **App** icon that needs to be sized to 1024 x 1024 pixels and include the copyright as well. I will use my name; you can use your name or a company name if you have one.

Also, change your version number to 1.0 if it is not already.

Next, scroll down further where it says Build and click on it. It should show the following screenshot:




Select the build and then click on **Done**.

Next, click on the small **Edit** text next to the **Rating** portion. A window will come up with a long list of themes, fill this out as best as you can for how your game is suited. For the game we made in the book, **Cartoon or Fantasy Violence** is checked to **Infrequent / Mild** and everything else is selected to **None** or **No**. Made for kids is also selected with **Ages 9 – 11**.

Then, click on **Done**.

At the bottom of the page, there is a section called **Version Release**. This allows you to have iTunes automatically release your game when/if its approved. Alternatively, you can manually release this version if you want to release it on a specific date and time.

In the top-right corner of the screen, click on **Save**.

Now, in the top-right corner of the page, there is the **Submit for Review** button. If you are ready, you can click on it now. It will load another page asking three questions:

- **Is your app designed to use cryptography or does it contain or incorporate cryptography? (Select Yes even if your app is only utilizing the encryption available in iOS or OS X.)**

  This answer is almost always no, but if you have a special case where your app does

use cryptography or incorporate cryptography, click on yes.

- **Does your app contain, display, or access third-party content?**

  This is most often *no*, but pay attention to your game and make sure that you answer correctly.

- **Does this app use the Advertising Identifier (IDFA)?**

  This checks whether you are using a third-party ad revenue service that has an IDFA. As we used iAds, we can say no here, but if you are creating a different game and using a service (such as Ad Colony), you will want to select yes.

# Congratulations!

You’ve made it! The game you created is now awaiting approval by Apple.

# What next?

For teaching and continuity reasons, this book does rush through a few of the topics related to creating games for iOS, although it does give you a starting point for almost everything you need to do if you want to create your own game.

# Testing

Testing is a big part of creating video games. Generally, games see many iterations of bug fixes and changes before submitting or even after submitting the game to the Apple Store. To aid this process, Apple does have a service called **test flight.** You can use this service by first navigating to your iTunes Connect account and then navigating to **Users and Roles**. Here, you can set up the Test Flight users who can then download the `TestFlight` app, so they will be alerted and will be able to test the game that you are creating.

You can even hire QA groups to play and test your game. Alternatively, you can use many forums to ask people to test. Feedback is a huge part of creating a quality product. It is highly advised that you find an avenue to get the said feedback from as many people as possible.

It can be difficult to be objective when you create a game because you are the one writing every line of code and putting together all the aspects of the game, making the viewpoint of the experience of playing it very different from a person who has never seen it before. Bigger studios even use focus test groups, where they hire a group of people to come to the studio and play the alpha and beta versions of the game so that they can see how people react and play. A lot can be said about a person getting confused or bored when playing your game. Also, you can find valuable information on how to fix such issues.

# I want to learn more about programming

Again, as this is an entry-level book, there are many aspects of the code that can be viewed as an inefficient practice. I did try to give you a view of the process of how to store references to objects and components so that you do not have to reach and search for objects at every single use, which is the very beginning of writing memory-efficient code.

- For general code references and tutorials, refer to: http://docs.unity3d.com/ScriptReference/ and http://unity3d.com/learn/tutorials/modules/beginner/scripting
- If you want to learn more about C#, refer to https://msdn.microsoft.com/en-us/library/618ayhy6.aspx.
- If you want to learn more about JavaScript, refer to https://www.codecademy.com/en/tracks/javascript and http://www.unity3dstudent.com/

Learning to program requires practice, attention to detail, and takes a lot of time. This attention to detail is ever evolving, and you will have to continue to find better methods to write the code you want so that the iOS devices can perform at their best efficiency. Programming for iOS devices is a challenging feat because these devices have limited hardware that needs code that pay attention to how these devices manage running games.

The Unity editor ships with a profiling tool. This tool tells you how much memory is being used and how well your game is running from the Unity editor. To access this tool, click on the **Window** drop-down menu and navigate to **Profiler**.

## Note

A fine tutorial on memory management using C# in Unity is available at http://www.gamasutra.com/blogs/WendelinReich/20131109/203841/C_Memory_Managem

# I want to create a 3D game

Optimization of 3D games is often much more intensive than a 2D game because you will be handling scenes that are much larger and more detailed. When you create 3D models for an iOS game, it is important to follow a couple of rules.

Make the mesh as low detailed as possible, although it should still be pleasant to look at. The more triangles the mesh has, the more difficult it is for the device to render it.

Use as few materials as possible. Similar to all devices, each object is rendered to the screen on each frame. The more objects you have, the more time is spent rendering them. For iOS games, the objects being rendered are called **draw calls**. The fewer draw calls you have, the better the game will run. If you have too many draw calls, the devices will use too much memory and/or take too long to render, which will either outright crash the game with an *out of memory* message or start to render at lower frames-per-second, making the game appear slower than it should be.

# 3D programs

Many mobile games are in 3D and require the use a 3D modeling program. A common free option is called **Blender**.

You can find more information about Blender at https://www.blender.org/.

Another option is from Autodesk. This has both Maya and 3D Studio Max. Both cost money, have month-to-month licensing plans, and allows you to pay for a commercially acceptable license over a smaller period of time. Refer to http://www.autodesk.com/products/maya/overview for more information on Maya.

You can find more information about 3Ds Max at http://www.autodesk.com/products/3ds-max/overview.

All of these 3D programs also offer animation tools, such as creating skeletons and handling of animation controls. Animation is well documented online. Also, there are many video tutorial series for each of these programs on how to animate a static 3D model.

These programs also provide you tools for UV mapping, which is the process of taking faces of the model and laying them out in a flat sheet to make textures fit the mesh correctly. UV mapping is well documented online. Also, a Google search will give you literally thousands of resources on how to do it.

# Creating textures

A commonly used option is a program called Photoshop created by Adobe. Adobe provides a license for Photoshop that is very cheap; currently, it is $9.99 a month for Photoshop and Lightroom.

You can find more information about Photoshop at https://creative.adobe.com/plans?promoid=KSPAK.

Photoshop tutorials have a massive presence online. You can also find many excellent resources through a Google search.

Two more resources on how to make textures a bit easier are DDO/NDO (created by Quixel) and Substance Painter / Substance Designer (created by Allegorithmic).

You can find more information about DDO/NDO at http://quixel.se/dev/ddo.

Refer to https://www.allegorithmic.com/products/substance-live for more information on **substance tools**.

# Furthermore

Unity has had an amazing community for years, creating thousands of hours of tutorials and support topics on how to use Unity and its tools. You can learn or ask anything you want through the Unity answers forum.

Unity answers are available at http://answers.unity3d.com/.

Unity also offers a lot of tutorials for all sorts of different game types and platforms. If you want to look up anything that the Unity engine can do, refer to the documentation section of Unity's website.

Unity tutorials are available at http://docs.unity3d.com/Manual/index.html.

# Using the Internet

The Internet itself is a fantastic tool to research and learn all the aspects of video game development. I recommend you to use Google for anything more that you would like to learn. It is how I have taught myself everything I know.

# Summary

Good work! I know that if you are new to game design, Unity, or programming, you will probably be overwhelmed. The best advice I can give you is that if you keep at it and try to learn something every day, before you know it, you will be able to create just about anything you want. Remember to keep your ideas within the scope of what you can manage. It is very often that a beginner game designer wants to create an AAA-equivalent game, and when they dive into it, they are completely lost and confused. Keeping yourself in check is how you get to the point where you can eventually create the game you dream about playing.

# Index

## A


- AchievementButton / AchievementButton
- achievement buttons
  - code for / Code for leaderboard and achievement buttons
- achievements
  - code for / Code for achievements
- Apple
  - game, submitting / Submitting the game to Apple
- Apple Developer
  - URL / Creating and logging in to the iTunes Connect profile, Certificates and provisions on Mac
- Awake function / The Awake function

# B


- BackToMenu button
  - adding, to GameUI / Adding the BackToMenu button in GameUI
- Box Collider 2D component
  - adding / Adding Box Collider 2D
- Bundle ID
  - creating / Creating the Bundle ID and Game App

# C


- C#
  - URL / I want to learn more about programming
- C# input class
  - creating / Creating the C# input class
- character
  - about / What is a character?
  - creating / Creating the character
  - GameObject, creating / Creating the Character GameObject
  - attributes / Character attributes
- character achievement code / The character achievement code
- character code
  - writing / Writing the character code and connecting the PlayerInput class to it
  - PlayerInput class, connecting / Writing the character code and connecting the PlayerInput class to it
- character controller
  - creating / Creating the character controller
- coin background / The coin background
- coin class
  - updating / Updating the coin class
- coin pickup
  - creating / Creating the coin pickup
- coins
  - showing / Updating code to show coins and distance
- collider / Adding Box Collider 2D
- CreateSpriteTiledGameObject function / The CreateSpriteTiledGameObject function

# D

- 2D artwork
  - URL / Creating the character
- 3Ds Max
  - URL / 3D programs
- distance background / The distance background

# F


- fade object
  - about / The fade object
  - restart functionality / The restart functionality
  - Awake function / The Awake function
- FadeToBlack function / FadeToBlack and FadeToNormal functions
- FadeToNormal function / FadeToBlack and FadeToNormal functions

# G


- game
  - submitting, to Apple / Submitting the game to Apple
  - testing / Testing
- game app
  - creating / Creating the Bundle ID and Game App
  - pricing, setting up / Setting up App pricing and territories
  - territories, setting up / Setting up App pricing and territories
  - in-app purchase, creating / Creating an in-app purchase
  - Game Center / Game Center, leaderboards, and achievements
  - achievements / Game Center, leaderboards, and achievements
  - leaderboards / Game Center, leaderboards, and achievements
- Game center events / Game center events
- GameInfo class / The GameInfo class
  - Update function / The Update function
  - RestartGame function / The RestartGame function
  - RestartLevelAndCharacter function / The RestartLevelAndCharacter function
  - FadeToBlack and FadeToNormal function / FadeToBlack and FadeToNormal functions
  - HideRestartButton function / The HideRestartButton function
- GameObject
  - creating / Creating the Character GameObject
- gameObject / Creating the coin pickup
- game user interface
  - creating / Creating the game user interface
  - GameInfo class / The GameInfo class
  - UI images, importing / Importing UI images
  - PlayerPrefs game save object / PlayerPrefs game save object
- global variables
  - about / Global variables

  / Global variables
- GUILayout / The GUILayout and OnGUI setup

# H


- HandBrake
  - URL / The Main App settings
- HideRestartButton function / The HideRestartButton function

# I


- iAds
  - displaying / Displaying iAds
- in-editor input logic
  - about / The in-editor input logic
  - testing / Testing in editor
- In App Purchase code / The In App Purchase code
- Input GameObject
  - creating / Creating the Input GameObject
- Integrated Development Environment (IDE) / Creating the C# input class
- iOS builds
  - Unity, setting up for / Setting up Unity for iOS builds
- iOS Development / Certificates and provisions on Mac
- iOS Distribution / Certificates and provisions on Mac
- iOS SDK plugin
  - installing / Installing the iOS SDK plugin
- iTunes Connect profile
  - creating / Creating and logging in to the iTunes Connect profile
  - logging in / Creating and logging in to the iTunes Connect profile
  - URL / Creating and logging in to the iTunes Connect profile

# J

- JavaScript
  - URL / I want to learn more about programming

# K


- KillCharacter function
  - about / The KillCharacter function

# L


- leaderboard
  - code for / Code for leaderboard and achievement buttons
- LeaderboardButton / LeaderboardButton
- LevelPiece code
  - obstacles, managing / Managing coins and obstacles with the LevelPiece code
  - coins, managing / Managing coins and obstacles with the LevelPiece code
- LevelPieceManager GameObject
  - setting up / Setting up the LevelPieceManager GameObject
- level piece prefabs / Level piece prefabs
- LevelPieces
  - coins, adding / Adding coins and obstacles to LevelPieces
  - obstacles, adding / Adding coins and obstacles to LevelPieces
- level pieces
  - setting up / Further setup of the level pieces
  - code / The level piece code
  - manager code / The level piece manager code

# M


- Mac
  - certificates and provisions / Certificates and provisions on Mac
- Main App settings
  - about / The Main App settings
  - iAds, enabling / Enabling iAds
  - iOS Native, setting up in Unity / Setting up iOS Native in Unity
  - certificates and provisions, on Mac / Certificates and provisions on Mac
  - iTunes Connect, importance / Why iTunes Connect is important
- main menu
  - LevelPieceManager menu code / The LevelPieceManager menu code
- main menu UI
  - building / Building the main menu UI
  - button images, adding / Adding button images
  - PlayButton / PlayButton
  - LeaderboardButton / LeaderboardButton
  - AchievementButton / AchievementButton
  - RemoveAdsButton / RemoveAdsButton
  - RestorePurchaseButton / RestorePurchaseButton
  - Purchase Remove iAds panel / The Purchase Remove iAds panel
  - Purchase Succeeded panel / The Purchase Succeeded panel
  - Purchase Failed panel / The Purchase Failed panel
  - menu code / The menu code
  - GameInfo menu code / The GameInfo menu code
- MainMenuUI reference
  - connecting, to GameInfo / Connecting the MainMenuUI reference to GameInfo
- Maya
  - URL / 3D programs
- menu code
  - GameInfo menu code / The GameInfo menu code
  - character menu code / The character menu code
- MenuItem creation / The MenuItem creation
- MonoDevelop / Creating the C# input class

# O


- obstacle
  - about / What is an obstacle?
  - creating / Creating the obstacle
- OnGUI create tiled button / The OnGUI create tiled button
- OnGUI function
  - GUILayout / The GUILayout and OnGUI setup
  - OnGUI setup / The GUILayout and OnGUI setup
- OnGUI setup / The GUILayout and OnGUI setup

# P


- pause button / The pause button
- pickup
  - about / What is a pickup?
- PlayButton
  - about / PlayButton
- PlayerInput class / Wrapping up
  - adding, to character code / Writing the character code and connecting the PlayerInput class to it
  - about / PlayerInput
- PlayerPrefs game save object / PlayerPrefs game save object
- prefab creation
  - about / The prefab creation
  - coin prefabs / Coin prefabs
  - level piece prefabs / Level piece prefabs
  - level pieces, further setup / Further setup of the level pieces
  - level piece, code / The level piece code
  - level pieces, manager code / The level piece manager code
  - LevelPieceManager GameObject, setting up / Setting up the LevelPieceManager GameObject
  - coins, adding to LevelPieces / Adding coins and obstacles to LevelPieces
  - obstacles, adding to LevelPieces / Adding coins and obstacles to LevelPieces
  - coins, managing with LevelPiece code / Managing coins and obstacles with the LevelPiece code
  - obstacles, managing with LevelPiece code / Managing coins and obstacles with the LevelPiece code
  - coins, resetting / Resetting coins
  - coins, resetting in LevelPiece / Resetting the coin in LevelPiece
- Purchase Failed panel
  - about / The Purchase Failed panel
- Purchase RemoveIAds functions / The Purchase RemoveIAds functions
- Purchase Remove iAds panel
  - about / The Purchase Remove iAds panel
  - RemoveAdsBackgroundScreen / RemoveAdsBackgroundScreen
  - RemoveAdsScreen / RemoveAdsScreen
  - RemoveAdsTextOption / RemoveAdsTextOption
  - ForCoin / ForCoin
  - ForCoinText / ForCoinText, ForCoinText
  - ForCash / ForCash
  - CloseRemoveAds / CloseRemoveAds
- purchases
  - buttons / Purchasing buttons
  - restoring / Restoring purchases

- Purchase Succeeded panel
  - about / The Purchase Succeeded panel
  - PurchaseSucceededBackgroundScreen / PurchaseSucceededBackgroundScreen
  - PurchaseSucceededScreen / PurchaseSucceededScreen
  - PurchaseSucceededText / PurchaseSucceededText
  - PurchaseSucceededAccept / PurchaseSucceededAccept
  - PurchaseSucceededAcceptText / PurchaseSucceededAcceptText
  - PurchaseSucceededClose / PurchaseSucceededClose

# R


- release build
  - creating / Making a release build
- RemoveAdsButton / RemoveAdsButton
- RemoveAdsScreen / RemoveAdsScreen
- RemoveAdsTextOption / RemoveAdsTextOption
- remove iAds
  - purchasing / Purchasing remove iAds
- RemoveiAds Purchase
  - for coins / New RemoveiAds Purchase for coins
- requisites, Unity 5
  - installing / Requirements
- ResetLevelPieces function
  - in LevelPieceManager / The ResetLevelPieces function in LevelPieceManager
  - Character, updating to Reset / Updating Character To Reset
  - RestartGame, connecting to RestartButton click / Connecting RestartGame to the RestartButton click
- restart button / The restart button
- RestartGame function / The RestartGame function
- RestartLevelAndCharacter function / The RestartLevelAndCharacter function
- RestorePurchaseButton / RestorePurchaseButton
- reverse-domain name / Creating the Bundle ID and Game App, Setting up iOS Native in Unity
- Rigidbody 2D component
  - adding / Adding Rigidbody 2D

# S


- sandbox mode / Creating a test build
- scene background
  - creating / Creating the scene background
  - movement, adding to SceneBackgroundElement / Adding movement to SceneBackgroundElement
- SceneBackgroundElement
  - movement, adding / Adding movement to SceneBackgroundElement
- sprite tile editor tool
  - making / Making the sprite tile editor tool
  - SpriteTiler C# class / The SpriteTiler C# class
  - global variables, adding / Global variables
  - MenuItem creation / The MenuItem creation
  - OnGUI function / The OnGUI function
  - OnGUI create tiled button / The OnGUI create tiled button
  - CreateSpriteTiledGameObject function / The CreateSpriteTiledGameObject function
  - SpriteTiler file, testing / Testing the SpriteTiler file
  - Sprite Tiler wrap-up / Sprite Tiler wrap-up
- SpriteTiler C# class / The SpriteTiler C# class
- SpriteTiler file
  - testing / Testing the SpriteTiler file, Sprite Tiler wrap-up
- Start function
  - about / The Start function
- Stock keeping unit (SKU) / Creating the Bundle ID and Game App
- SubmitAchievementAsWhole / SubmitAchievementAsWhole
- SubmitAchievementProgress / SubmitAchievementProgress
- SubmitLeaderboardScore / SubmitLeaderboardScore
- system
  - including / Including system

# T


- test build
  - creating / Creating a test build
- textures
  - creating / Creating textures
- Touch
  - about / What is Touch?, Our own Touch
  - global variables / Global variables
- Touch device
  - input / Input for the Touch device
- TouchInput function
  - calculating / Calculating the TouchInput function

# U


- UI Canvas, creating
  - coin background / The coin background
  - distance background / The distance background
  - pause button / The pause button
- UI images, importing
  - about / Importing UI images
  - UI Canvas, creating / Creating the UI Canvas
- Unity
  - changes, finishing / Completing changes in Unity
  - setting up, for iOS builds / Setting up Unity for iOS builds
  - test build, creating / Creating a test build
  - release build, creating / Making a release build
  - URL / Furthermore
- Unity 5
  - requisites / Requirements
  - installing / Downloading and installing Unity 5
  - URL / Downloading and installing Unity 5
  - downloading / Downloading and installing Unity 5
- Update function
  - about / Starting to work with Update

  / The Update function