Community Experience Distilled

Unity 5.x 2D Game Development Blueprints

Explore the features of Unity 5 for 2D game development by building three amazing game projects

Francesco Sapio
Abdelrahman Saher

[PACKT] PUBLISHING open source* community experience distilled

www.allitebooks.com

Unity 5.x 2D Game Development Blueprints

Explore the features of Unity 5 for 2D game development by building three amazing game projects

Francesco Sapio
Abdelrahman Saher

BIRMINGHAM - MUMBAI

www.allitebooks.com

Unity 5.x 2D Game Development Blueprints

Copyright © 2016 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 authors, 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 2016

Production reference: 1230916

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78439-310-6

www.packtpub.com

www.allitebooks.com

Credits

Authors

Francesco Sapio

Abdelrahman Saher

Reviewers

Elizabeth Keegan

Melody Kaye Cariaga

Acquisition Editor

Larissa Pinto

Content Development Editor

Shali Deeraj

Technical Editor

Sachit Bedi

Copy Editor

Safis Editing

Project Coordinator

Devanshi Doshi

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Graphics

Disha Haria

Production Coordinator

Arvindkumar Gupta

www.allitebooks.com

About the Authors

Francesco Sapio obtained his computer science and control engineering degree from Sapienza University of Rome, Italy, a couple of semesters in advance, scoring summa cum laude. Now, he is studying a master of science in engineering in artificial intelligence and robotics.

He is a Unity3D and Unreal expert, a skilled game designer, and an experienced user of the major graphics programs.

Recently, he authored the book *Unity UI Cookbook (*Packt Publishing) which teaches readers how to develop exciting and practical user interfaces for games within Unity, and a short e-guide *What you need to know about Unity* (Packt Publishing). Furthermore, he has also been a reviewer for the following books: *Unity Game Development Scripting* (Packt Publishing) and *Unity 5.x by Example* (Packt Publishing).

Francesco is also a musician and a composer, especially of soundtracks for short films and video games. For several years, he worked as an actor and dancer. He was a guest of honor at the theater Brancaccio in Rome.

In addition, he is a very active person, having volunteered as a children's entertainer at the Associazione Culturale Torraccia in Rome. He also gives private lessons in mathematics and music to high school and university students.

Finally, Francesco loves math, philosophy, logic, and puzzle solving, but most of all, creating video games, thanks to his passion for game designing and programming.

You can find him at `www.francescosapio.com`

I'm deeply thankful to my parents for their infinite patience, enthusiasm, and support throughout my life. Moreover, I'm thankful to the rest of my family, in particular my grandparents, since they always encouraged me to do better in my life with the Latin expressions "ad maiora" and "per aspera ad astra." Finally, a huge thanks to all the special people around me whom I love, in particular to my girlfriend; I'm grateful for all your help in everything.

www.allitebooks.com

Abdelrahman Saher *graduated with a BSc in Computer Science in 2012. After graduation, he worked for the video game company EverylPlays, where he participated in the programming of a couple of mobile games. Later, in 2013, he moved into the challenging role of lead programmer with the video game company Appslnnovate. Apart from his full-time job, Abdelrahman recently started his own start-up video game company called Robonite.*

www.allitebooks.com

About the Reviewers

Elizabeth Keegan is a practitioner in the field of game design and art. She obtained her bachelors degree from Cleveland's very own Institute of Art and masters degree from UC Berkeley in Fine Art. Currently, she's heading the continued growth of the Game Design program at Notre Dame College. Game design has been her primary focus for many years and spans a range of disciplines. She has worked both collaboratively and independently on many projects concerning topics such as Artificial Intelligence, Air Pollution, Autism, and Mental Therapy. More importantly, Elizabeth has taught across a range of age groups and classrooms. She has worked alongside high school students in summer camps, 9th grade year, round programs, and college classrooms. All of these experiences have taught her the power of game development and its ability to facilitate critical thinking and creative problem solving among many other crucial skills in today's workplace.
She is the current a game design instructor at Notre Dame College in South Euclid, Ohio, and has worked alongside Rachel Morris who is the head of the art department.

I would like to thank both Abdelrahman Saher and Francesco Sapio,for providing yet another accessible resource to those interested in game development. I would also like to thank Sanchita Mandal and Paushali Desai for including me in the review process.

www.allitebooks.com

Melody Kaye Cariaga graduated from De La Salle College of St. Benilde with a BS in Information Technology and a specialization in game design and development. This program was a first in the Philippines; on February 2013, she was a part of the very first batch of graduates with the title of cum laude under her name.

Since February 2013, she has been working as a game developer. On her first job, she developed HTML5 games for three major American cable and satellite television networks aimed mainly at the child and adolescent demographic. In May 2015, she started working for Xurpas Inc., the first tech startup in Southeast Asia to go on initial public offering, as one of their Unity developers that create Android games.

Melody is a strong, hardworking woman, who always aims to do better and be better in life. She is a well-organized individual who never fails to give more than 100% in her work and never loses sight of her goal. Even though she is driven, she still has time to enjoy playing games, listening to music, and watching movies. She never loses sight of who she really is.

> *For all the success and achievements I have received throughout my career, I would like to thank God Almighty for His everlasting grace, my family for their never-ending guidance and support, and my special someone for his trust in everything I do.*

www.allitebooks.com

www.PacktPub.com

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.

www.allitebooks.com

Table of Contents

www.allitebooks.com

Preface

In this book, there are three projects presented: a Platformer game, an RPG-style game, and a Tower Defense game. Their purpose is to teach you Unity 2D game development with a practical approach.

What this book covers

`Chapter 1`, *Sprites*, introduces the reader to the basic elements of 2D game development in Unity, Sprites. Furthermore, it gets the reader to start building the platform game.

`Chapter 2`, *Animations*, explains how to animate a Sprite in Unity and how to trigger different animations depending on the state of the character.

`Chapter 3`, *Physics*, teaches how to deal with 2D physics in Unity and how to use it to achieve believable movements.

`Chapter 4`, *Level Design*, introduces the reader to the Tiled Map Editor by showing the workflow from the creation of the map, and how to import it into Unity. Furthermore, it concludes the platform game.

`Chapter 5`, *Creating Our Own RPG*, starts with the foundations for creating the RPG game, by explaining the basic concepts needed for it.

`Chapter 6`, *AI and Pathfinding*, introduces the reader to basic Artificial Intelligence techniques, such as Pathfinding, and completes the RPG game.

`Chapter 7`, *Tower Defense Basics*, explains the basics for the creation of a Tower Defense game, using all the concepts learned so far.

`Chapter 8`, *User Interface for the Tower Defense Game*, dives into more in detail about creating a user interface for your games, with particular focus on the Tower Defense game.

`Chapter 9`, *Finishing Tower Defense Game*, wraps everything up and completes the Tower Defense game by explaining gameplay elements and how to make all the elements of before they interact with each other.

What you need for this book

For this book, you will need Unity 5.x and Tiled Map Editor 0.16.1.

Who this book is for

If you've got the basics of 2D development down, push your skills with the projects in this hands-on guide. Diversify your portfolio and learn the skills to build a range of awesome 2D games in different genres.

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: "We only need the `base pack` folder."

A block of code is set as follows:

```
// The Player's speed
public float speed = 10.0f;

//Game boundaries
private float leftWall = -4f;
private float rightWall = 4f;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
void Awake () {

        moneyCounter =
GameObject.Find("MoneyCounter").GetComponent<MoneyCounterScript>();
        uiImage = GetComponent<Image>();
    TowerScript.towerMenu = this.gameObject;
  }
```

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: "Click on **Asset packages** and select **2D**."

Warnings or important notes appear in a box like this.

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 for this book 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.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Unity-5x-2D-Game-Development-Blueprints`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

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/Unity5x2DGameDevelopmentBlueprints_ColorImages.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.

1
Sprites

As we start our journey into the world of 2D game development, let's start this chapter by talking about the most important elements of creating 2D games. A 2D sprite is a two-dimensional image that is rendered on your screen while the game is still running.

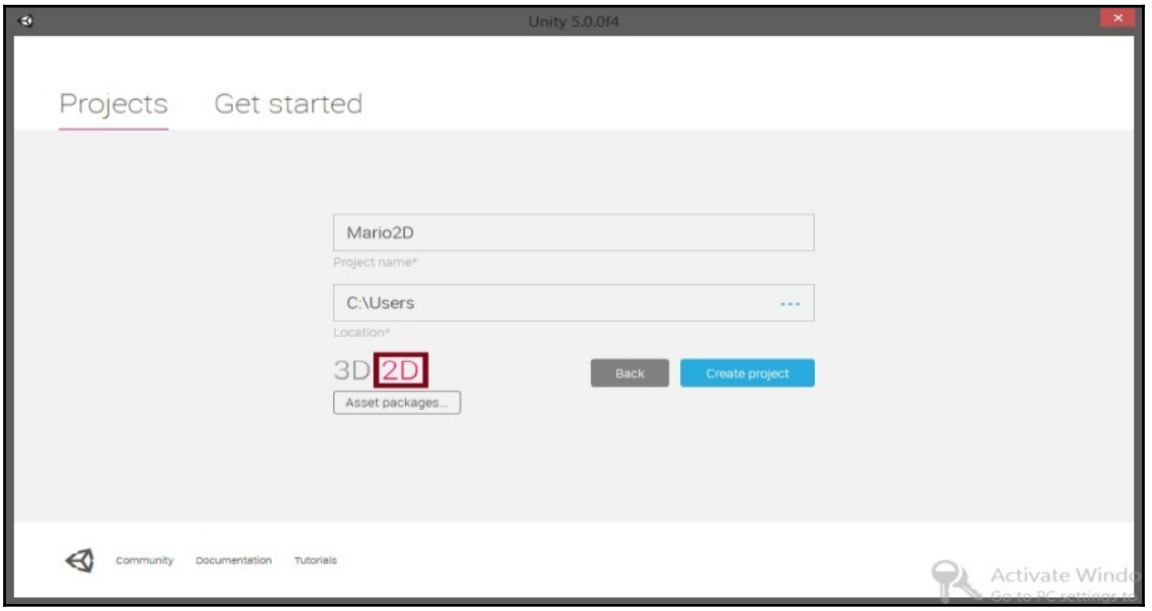
In this chapter, we will start working on our own Platformer game. It consists of a character that must navigate through a platform level by jumping and running to achieve certain tasks. Along the way, we will learn how to use sprites and how Unity handles them. In this particular chapter, we will cover:

- Using the 2D mode within Unity
- Importing and rendering sprites
- Creating sprite sheets and atlases
- Beginning to use scripting

2D mode

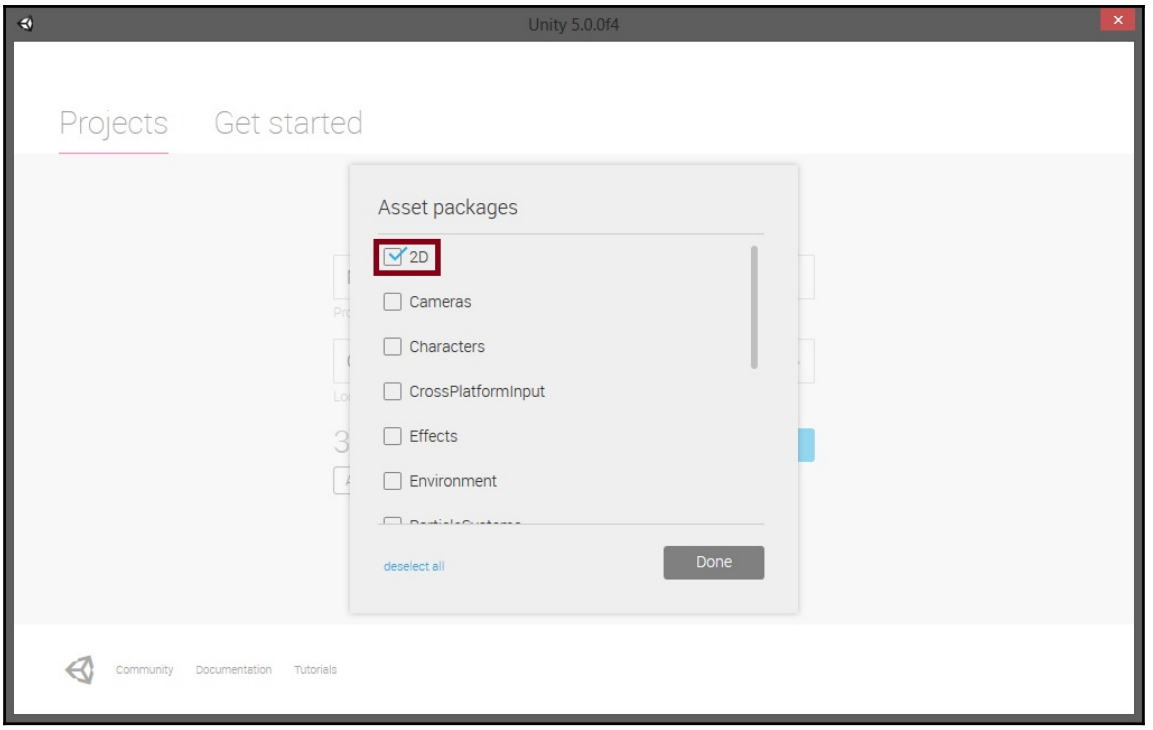
Unity has a 2D mode that allows us to quickly set up the project for 2D game development. In fact, the main reason to use this mode is to automatically import new assets as `Sprites`.

When creating a new project, you have the option to choose between the 3D and the 2D mode. Let's select the 2D mode, as shown in the following image:

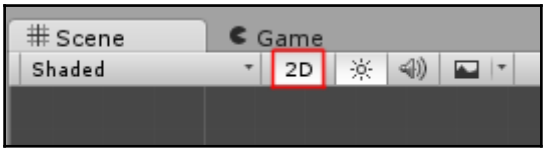


Now, we need to import the standard assets that we will use to build our game. Click on **Asset packages** and select **2D**.

Unity doesn't come with the standard packages, you need to download them from the official website.



Finally, we can click on **Create project**. If you have used Unity before for 3D game development, you will notice a few differences in the default interface. In particular, the 2D view is already selected:



Furthermore, the camera in new scenes will always be orthographic, which is exactly what we want:

Having selected the 2D mode doesn't mean that you cannot change it any more. In fact, you can change it to 3D mode whenever you want by going to **Edit** | **Project Settings** | **Editor** and selecting **3D** under **Default Behavior Mode**.

This can come in handy when adding 3D models to your 2D project and vice versa. It is recommended that you switch before importing 2D sprites or 3D models to the appropriate mode since Unity will import textures accordingly.

Custom packages

During the course of this book, we will use custom packages, since we don't have time to create all the graphics on our own.

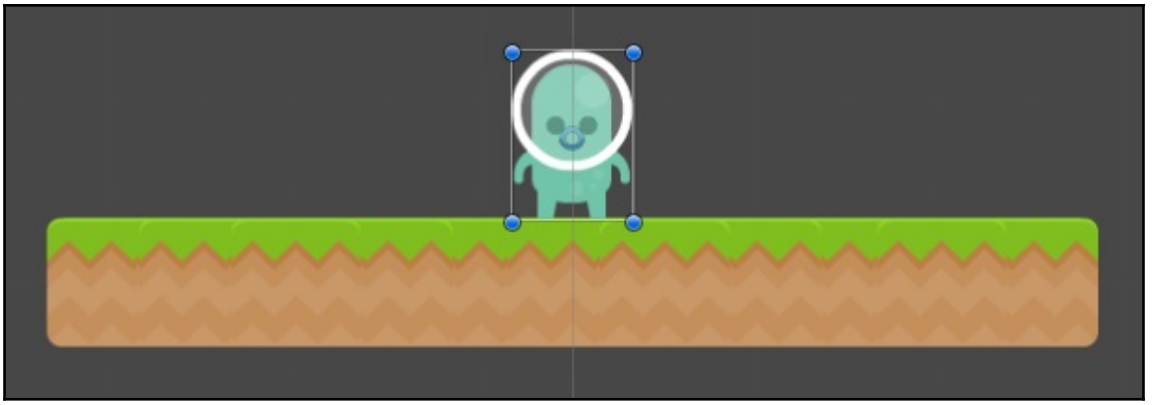
For our first game, we are going to use a package from `http://kenney.nl`, which is a website full of interesting graphic packages, and free to use in any project. In particular, you need to download the following package: `http://kenney.nl/assets/platformer-art-deluxe`.

Once the file has been downloaded, it is compressed, so we will need to open it with a software for decompression.

As you can see, there are many folders containing different expansions of the pack. We only need the `basepack` folder. We need to copy this folder into the `Asset` folder in Unity. You can do this by just dragging and dropping it in the **Project** panel. Finally, we can rename it `PlatformerPack`, so our project will be better organized.

Dealing with sprites

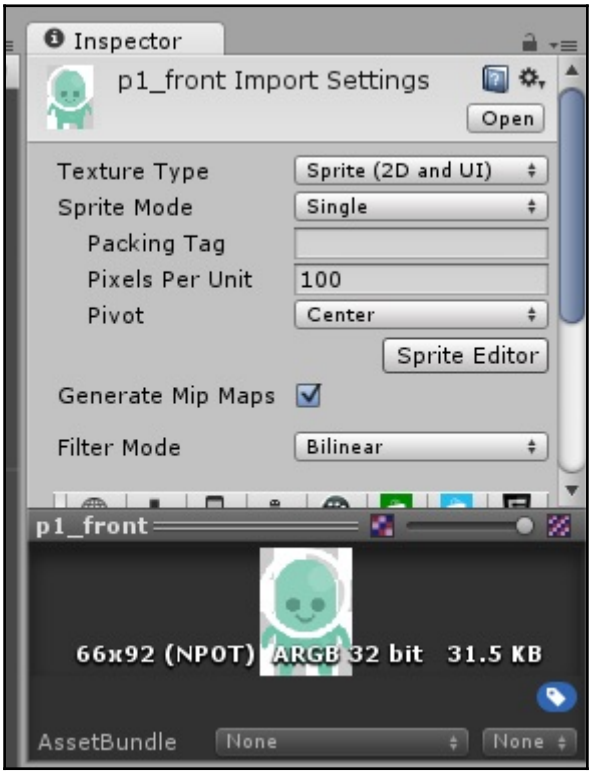
When working on our 2D game, we need sprites to fill in our environment, and also to display characters. Several sprites can be used to create an animation, like a walking character: each sprite represents a certain frame in the animation.



Importing sprites

If we don't import the package when we have created the project, we can do it at any moment by clicking on **Assets** | **Import Package**.

In the **Project** panel, click on **p1_front** to see its setting in the **Inspector**. It can be found under **Platformer Pack** | **Player**, or can be searched by using the small search box in the right upper corner of the **Project** panel. Once selected, the **Inspector** looks like the following:



You can control the properties of the sprites by changing the values in the **Inspector**. In order to understand them, let's break them down:

- **Sprite Mode**: This mode consists of two options `Single` and `Multiple`. `Single` should be selected when the image contains only a single object or character that will be used as a single sprite. `Multiple`, instead, it should be selected when multiple elements are contained within the same image .These may include different variations of the same object or its animation sheet.

- **Packing Tag**: This is an optional variable used to specify the name of the `Sprite Sheet` in which this texture will be packed. This is useful when we need to optimize our game.
- **Pixels Per Unit**: This controls the scale of the sprite. This variable defines how many pixels correspond to one world space unit. The default value for this is `100`.
- **Pivot**: This allows us to change the pivot point for our sprite, which, by default, is set to `Center`. When required, you can change it to one of the other predefined points or place it in a custom position by selecting `Custom`.

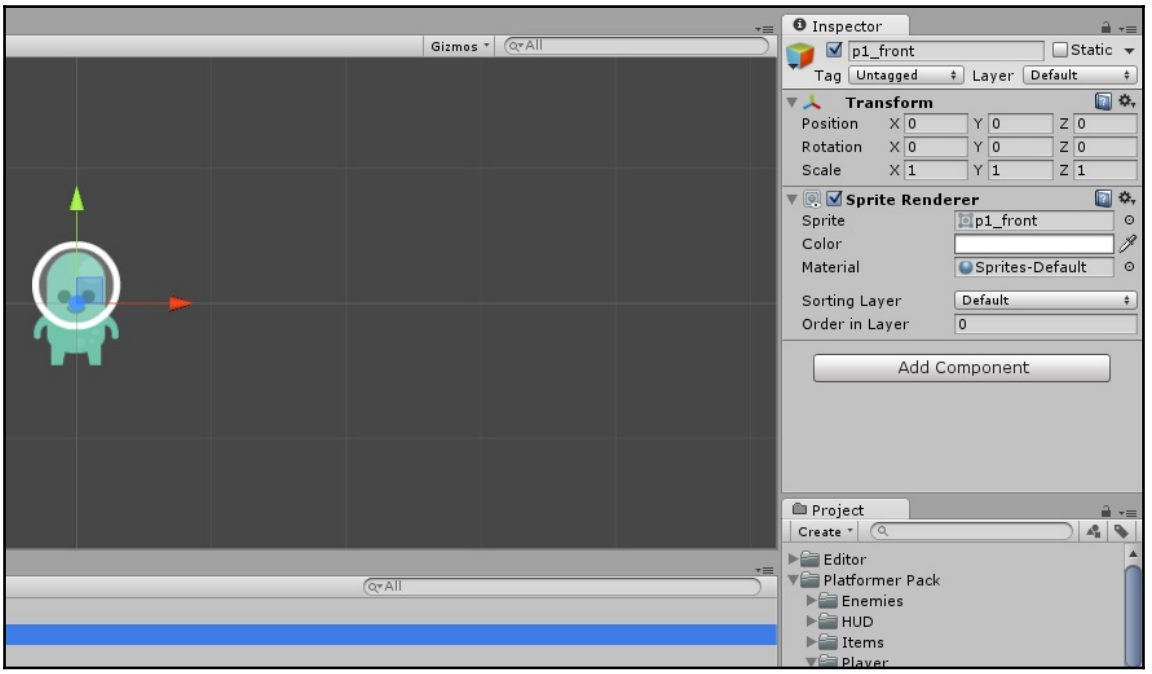
Choosing `Multiple` instead of `Single` in the sprite mode will remove the pivot option. In fact, the pivot point for each sprite in the image can be selected by using the **Sprite Editor**.

The Sprite Renderer component

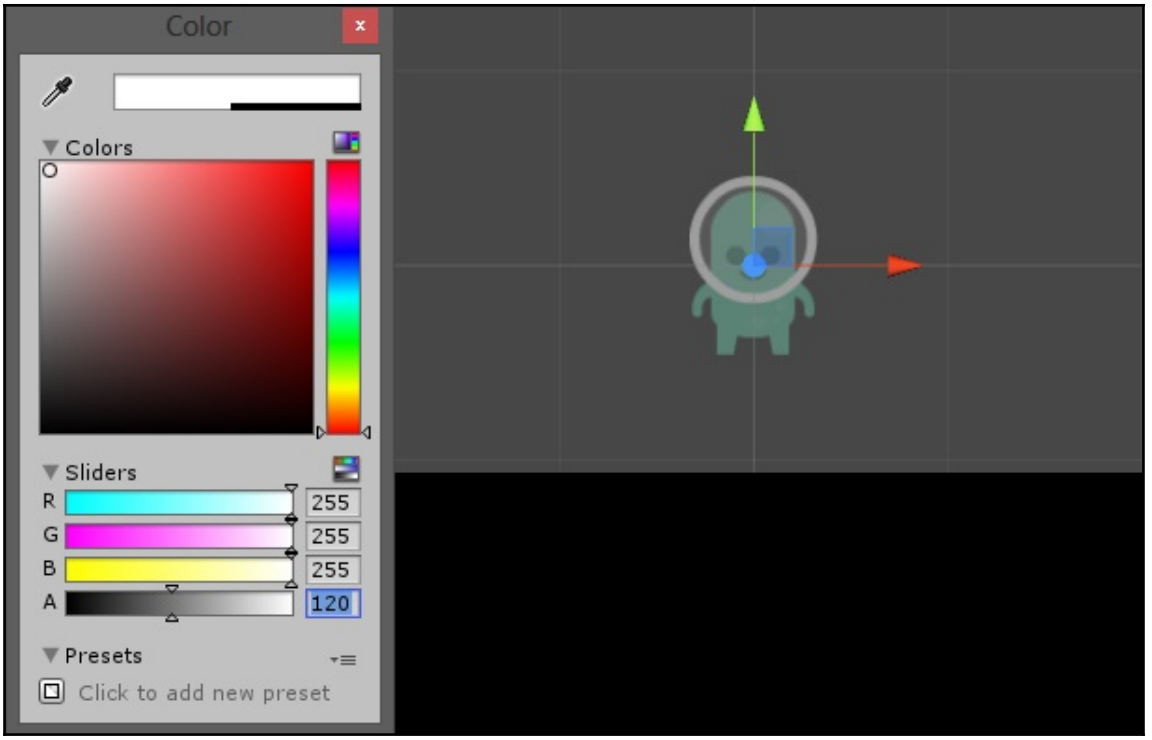
Since we already have the `p1_front` sprite selected, let's drag it into the **Hierarchy** panel.

You can place it in the scene by dragging it directly inside the **Scene** view.

When we add a sprite in our scene, a game object is created with a Sprite Renderer component attached to it. This component is responsible for rendering a `Sprite` in the game; without it, the game object would be empty.

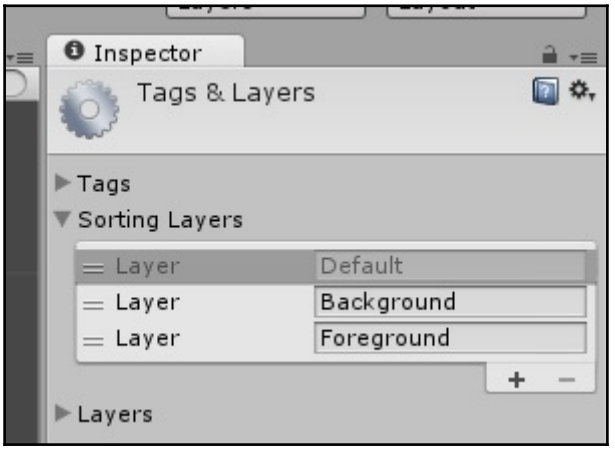


Let's explore the options in this component. First, **Sprite** is the variable that will store the sprite to render on screen. In this case, it is automatically with the sprite we have dragged in. The **Color** variable controls the color of the **Sprite** along with its alpha channel. If we click on it, a color picker shows up; we can see the effect of our change immediately in the **Scene** view:



The **Material** variable stores information about the material of the sprite. By default, it is set to `Default Sprite Material`. Usually, we don't want to change this during 2D game development; however, it may be necessary in particular cases, for instance when the sprite needs to be affected by lights.

Then, the **Sorting Layer** and **Order In Layer** variables are used to define the order of visualization of the sprites in the scene. In fact, not all the sprites are on the same level. Think about a background, a cloud, and our character. The cloud should be located on top of the background and the character should be on top of both of them. By default, the **Sorting Layer** is set to `Default` and **Order In Layer** to 0. So far, `Default` is the only layer available. Since we want to order our sprites later, let's create a few more layers. Click on **Add Sorting Layer** under **Sorting Layer**. As a result, the **Inspector** shows us the **Tags & Layers** settings. By clicking on the **+** button in the right bottom corner, we are able to add other sorting layers. Let's add two more layers, and name them respectively `Background` and `Foreground`, as shown in the following image:



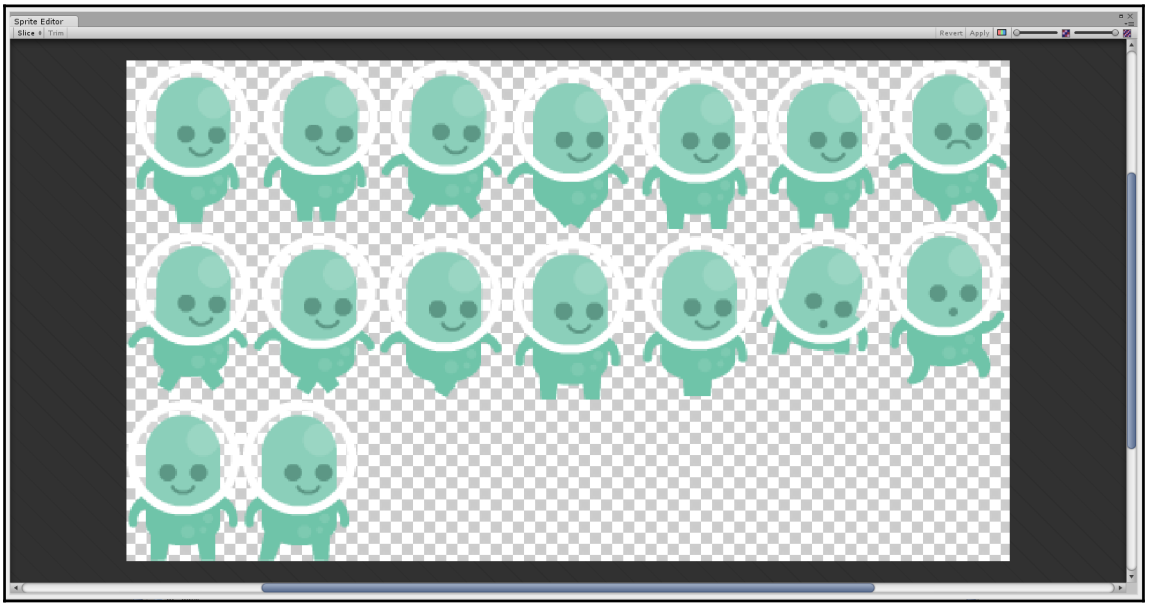
The order of these layers is important. We can easily change it by clicking on the designated layer and dragging it above or below another one.
Now, select the game object we have created in the **Hierarchy** panel again. In the **Inspector**, we are able to change its sorting layer to `Foreground`.

Before going forward, create a new folder in the **Project** panel named `Scenes` and save the scene inside it as `Scene1`. We can do this by clicking on **File** | **Save Scene**.

The Sprite Editor

In the importing settings, we can find the **Sprite Editor** button. After we have selected `p1_spritesheet` again from **Platformer Pack** | **Player**, we can now click on the button. As a result, the **Sprite Editor** window shows up.

The **Sprite Editor** should be used when dealing with a sprite that contains multiple elements (If this is the case, don't forget to set `Multiple` in the **Sprite Mode**). So, we should see something like the following:



Our goal is to slice all the single positions of the character in the image, so that we can use them as an individual sprites in our scene. There are different ways to achieve this. Let's discuss some of them:

- **Click and drag**: This allows you to simply click and drag over the desired elements to create rectangular selections that will define each sprite. You can change each selection as preferred. You are able to change the position by dragging the rectangle, its size by clicking on the corners of each rectangle, and the **Pivot** point by clicking and dragging it. Furthermore, clicking on the trim button in the sprite window will change the size of the rectangle to fit the selected sprite.

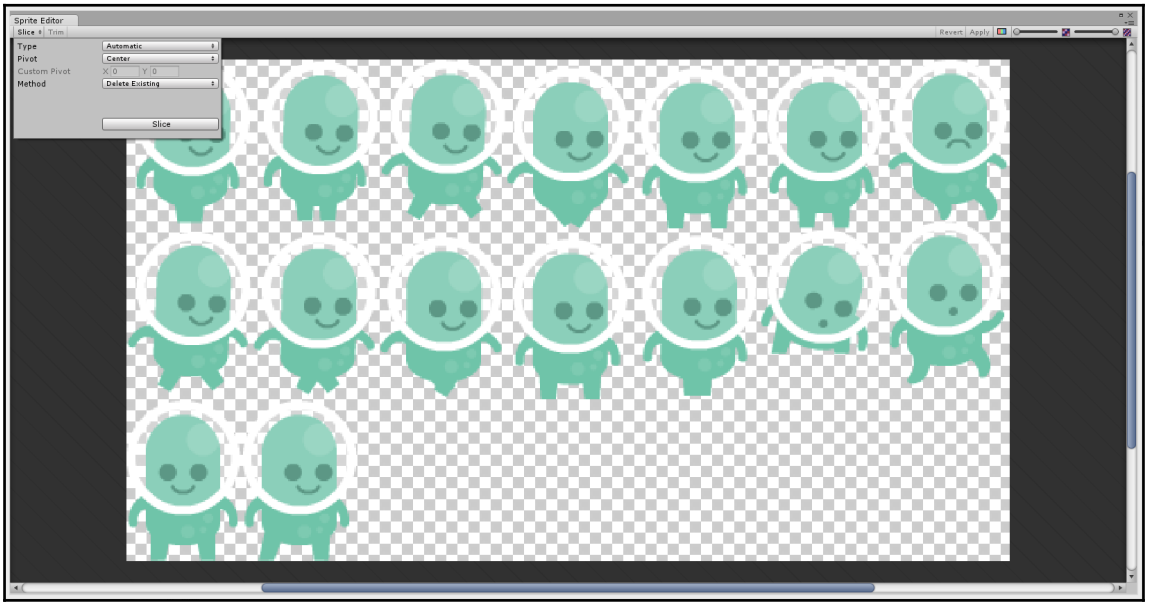
  By clicking on the slice button in the top left of the **Sprite Editor**, a new window appears. This allows you to select other ways to slice your `Sprite Sheets`. The default slicing type is set to `Automatic`, but you can also choose different kinds of grids too.

- **Automatic**: When using the automatic method, Unity will detect each sprite and draw a trimmed rectangle around it. With `Automatic` selected, you can also change the **Pivot** position for each sprite. We can also select a **Method** to tell Unity what should happen to the sprites that are already defined. The `Delete Existing` method deletes all the previous selections and creates new ones from scratch. The `Smart` method attempts to create new selections for undefined sprites, while adjusting them to fit with the older selections. Finally, the `Safe` method adds new selections over the previous ones without changing them.
- **Grid**: This allows you to create equal size selections for all the sprites in the image. Once we have set `Grid` as slicing type, then we will be able to change the size used for the slicing, and eventually adjust the position of the **Pivot** point for each sprite.

www.allitebooks.com

In particular, for our project, we can just use the `Automatic` method – it will work fine.

By clicking on the **Revert** button, we can restart from scratch, removing all the previous selections.



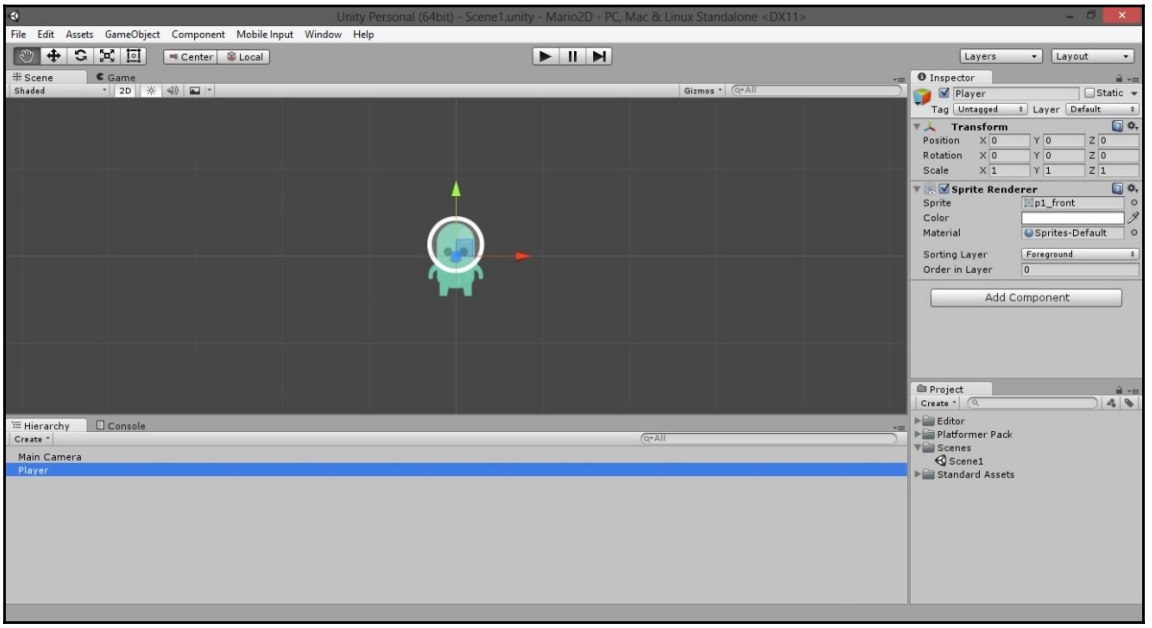
After closing the **Sprite Editor**, let's check the image file in the **Project** panel. As you can see, a little icon appeared and it allows us to expand the file and see all the single `Sprites` we have created in the **Sprite Editor**. Now, these can be used as normal `Sprites` in our game and can be placed in our scenes and scripts.

Having a single image with different Sprites is a technique called Sprite sheets. When importing sprites to your game project, it is preferred to group them into one image. This is because Unity can send a single file in the graphic card, and, as a result, enhance the performance by saving both memory and computational time.
Furthermore, sprite sheets are useful for creating 2D animations (such as walking, jumping, breathing, and explosions). In fact, it allows us to keep everything organized and easy to use. In the next chapter, we will discuss the process of animation in more detail.

Our character makes its first steps

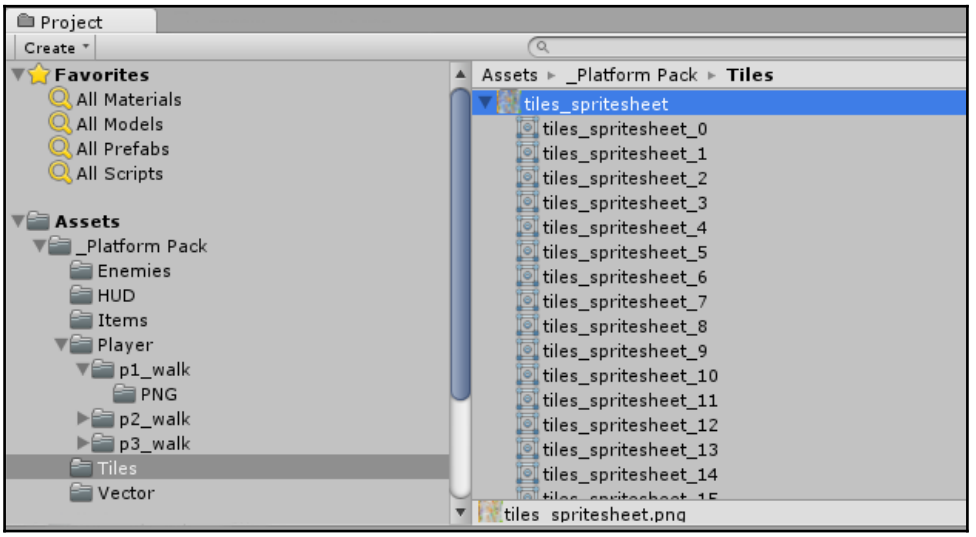
Starting from this chapter, and continuing in the next two, we will work on our first project. It is a Platformer game that resembles the original Mario game.

In this chapter our objective is to create a character that will move on a platform across the *x*-axis. We can achieve this by using the player input. Now, let's continue from where we stopped. We should have a game object named `p1_front`, now we can rename it as `Player`. As a result, your scene should look like the following:

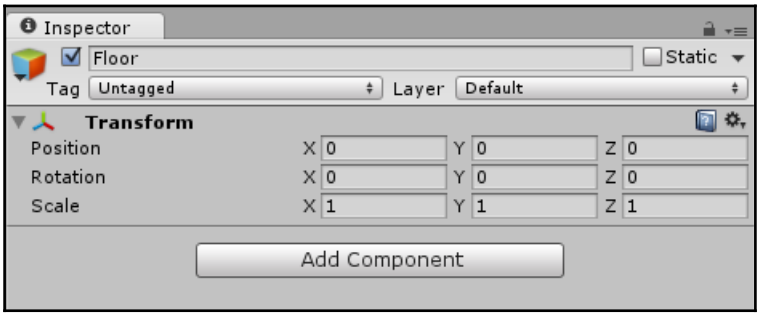


However, our character is now floating in the middle of nowhere. Therefore, adding some ground for him to move on is a good start. To achieve this, go to the **Project** panel and from the **Platformer Pack** folder select `tiles_spritesheet`. We can see its **Import Settings** in the **Inspector**. We need to change the **Sprite Mode** to `Multiple` and then click on **Apply**.

Open the **Sprite Editor** for our selected asset by clicking on the **Sprite Editor** button. As we did before, we can slice the image by using the `Automatic` method and it will work fine. Now that each sprite has been sliced, click on **Apply** and close the editor. You will now notice that more sprites have been generated under `tiles_spritesheet`, as shown here:

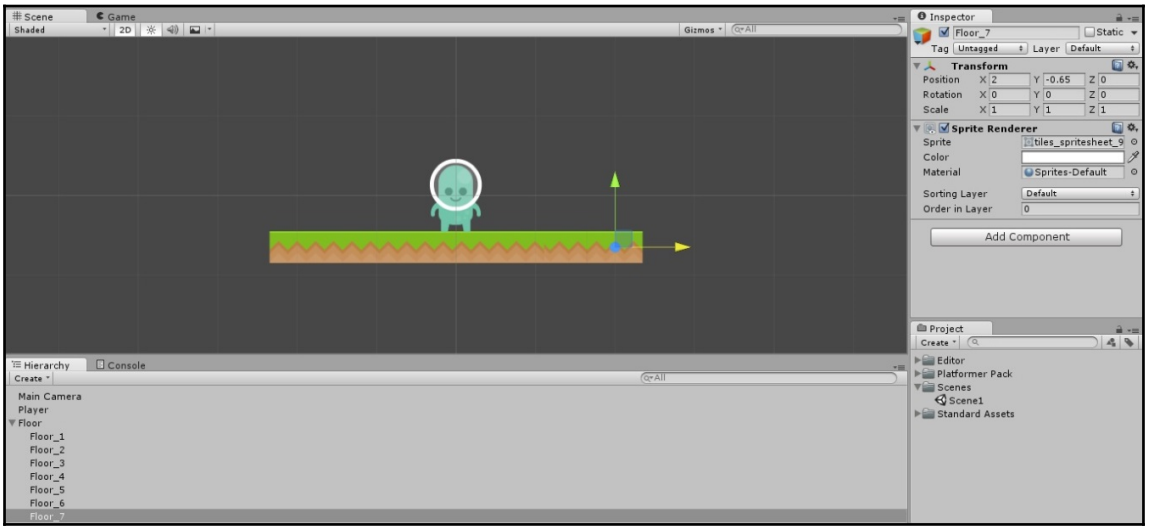


In order to keep things organized, let's create a new empty game object. It can be done by right-clicking in the **Hierarchy** panel, and then selecting `Empty` game object. In the **Inspector**, we can rename this `Floor` and change its position in `(0,0,0)`, as shown in the following image:



Now we can drag the sprite named `tiles_spritesheet_9` onto the `Floor` object we just created. Rename the file to `Floor_1` and set its position to `(-2,-0.65, 0)`. Now we also need to change the sorting layer to `Foreground` and the sorting order to `1` so it doesn't overlap with our player.

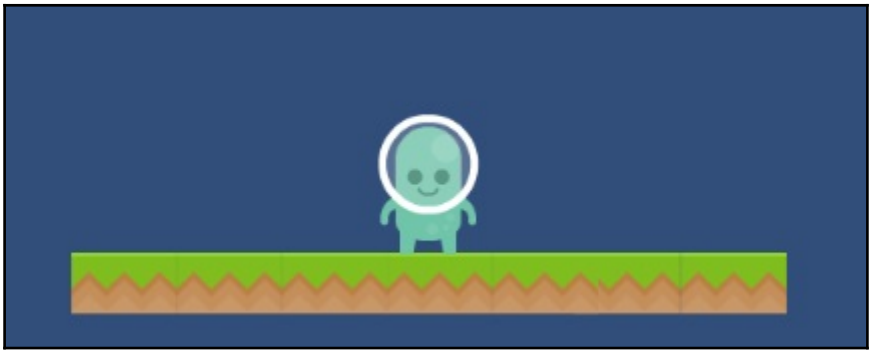
As you can see, we have the first tile of our floor under the player and now we want to create more tiles in order to create a floor that the player can walk on. We could scale the `Floor_1` object to make it larger. However, the advantage to having titles is the possibility to use more of them next to each other in a seamless way. Therefore, let's duplicate it. To achieve this, select the `Floor_1` object and then you can duplicate either by right-clicking and selecting `Duplicate`, or by using *Ctrl* + *D* (for Mac users *cmd* + *D*). You will notice that the new object is automatically named `Floor_2`, which is what we would like to have to keep things organized. Click on the new object and by using the `Rect` tool, move it to the left from the scene view, next to the original object. Now, to get our floor, let's repeat this procedure and move the tiles, until we have something like the following:



If you want to reproduce the scene like in the previous image, the last tile on the right is positioned at `(2,-0.65,0)`.

You can duplicate groups of objects when you need to duplicate large pieces of the map.

After finishing the floor tiles, let's check the game view. It appears that the objects are rather small. We can fix this by scaling the game objects. This is not best practice, because usually the graphics should already be created without the need to scale. On the other hand, we may want to change the camera settings to look at our world more closely. However, for the purpose of learning something new, we can change the scale of the player to `(2, 2, 1)`. This way, we can make our player bigger. Repeat the same with the `Floor` game object by setting its scale to `(2, 2, 1)`. As a result, all of its children (for example, the single tiles of the floor) will be scaled. As a result, the game view should look like the following:



Our next step is allowing the player to move the character, in this case, along the *x*-axis. To achieve this, we need to create a script. In the **Project** panel, create a new folder and rename it as `Scripts`. In this new folder, let's create a new C# script by right-clicking and then selecting **Create** | **C# Script**. Rename it to `PlayerMovement` and then attach the script to our player by dragging and dropping it onto the player object. Double-click on the script to open it.

First, we need to add some variables to control the movement. In particular, one for the speed, and another two for the boundary of our world. Let's add the following variables:

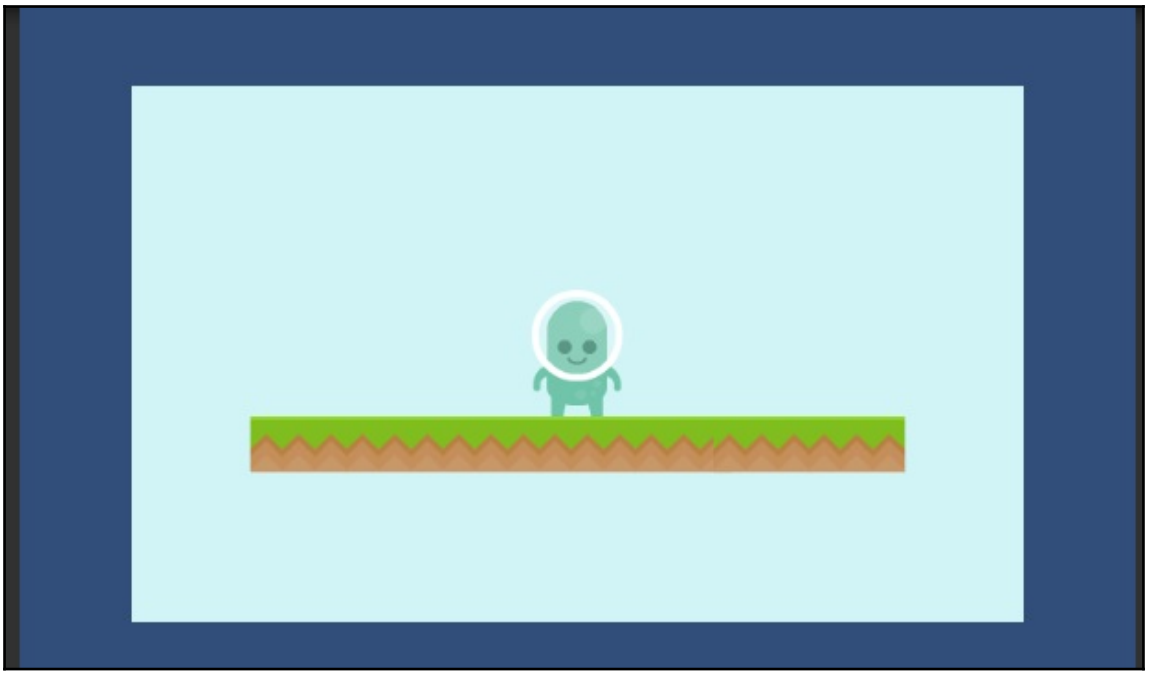
```
// The Player's speed
public float speed = 10.0f;

//Game boundaries
private float leftWall = -4f;
private float rightWall = 4f;
```

Now, in the `Update()` function we first need to calculate how far it will be translated and where, based on the input of the player. In this case, we take the horizontal axis, which, by default, is bound to the arrow keys. In order to calculate this, we need to take into consideration how much time is passed from the last frame. This can be done by adding `Time.deltaTime` into the equation. Then, we need to translate the character so it doesn't fall off the boundary after the translation. So we will write the following:

```
    // Update is called once per frame
    void Update () {
        // Get the horizontal axis that by default is bound to the arrow
keys
        // The value is in the range -1 to 1
        // Make it move per seconds instead of frames
        float translation = Input.GetAxis("Horizontal") * speed *
Time.deltaTime;
        // Move along the object's x-axis within the floor bounds
        if (transform.position.x + translation < rightWall &&
           transform.position.x + translation > leftWall)
            transform.Translate(translation, 0, 0);
    }
```

Save the changes and hit the play button. By using the left and right arrow (alternatively *A* and *D*), you will be able to move the character across the platform. You can change the speed of the player by adjusting the speed value from the **Inspector**, as shown here:

Now, our scene needs a background. In the **Project** panel in the **Platformer Pack** folder drag the file `bg` into the scene. Rename the game object to `Background` and set the scale to `(5,3,0)`. As a result, the game view should look like the following:



Summary

In this chapter, we covered 2D sprites and started working on our first project. In particular, we have discussed the 2D mode, importing `Sprites`, the `Sprite Renderer` component, and the **Sprite Editor**. We also learned how to use sprites in our game and how to script a character to make it move along the *x*-axis.

In the next chapter, we will bring our character to life by adding animations into our game!

2
Animations

Animating sprites is what makes them and the game come to life. Some 2D games are made just with static images by design. However, in our case, we will make animations and go through the process of animating 2D characters in Unity.

In this chapter, we will learn how to create and play animations for the player character to see as Unity controls the player and other elements in the game. This is what we will go through:

- Animating sprites
- Integrating animations into animators
- Continuing our platform game

Animating sprites

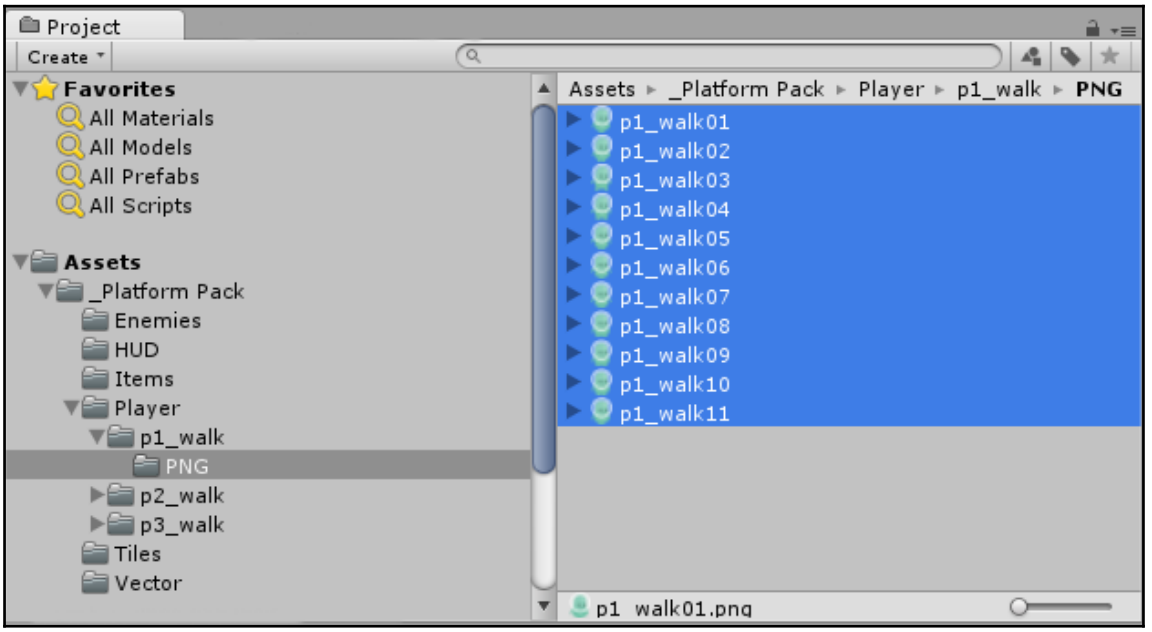
Creating and using animations for sprites is a bit easier than other parts of the development stage. By using animations and tools to animate our game, we have the ability to breathe some life into it. Let's start by creating a running animation for our player. There are two ways of creating animations in Unity: automatic clip creation and manual clip creation.

Automatic clip creation

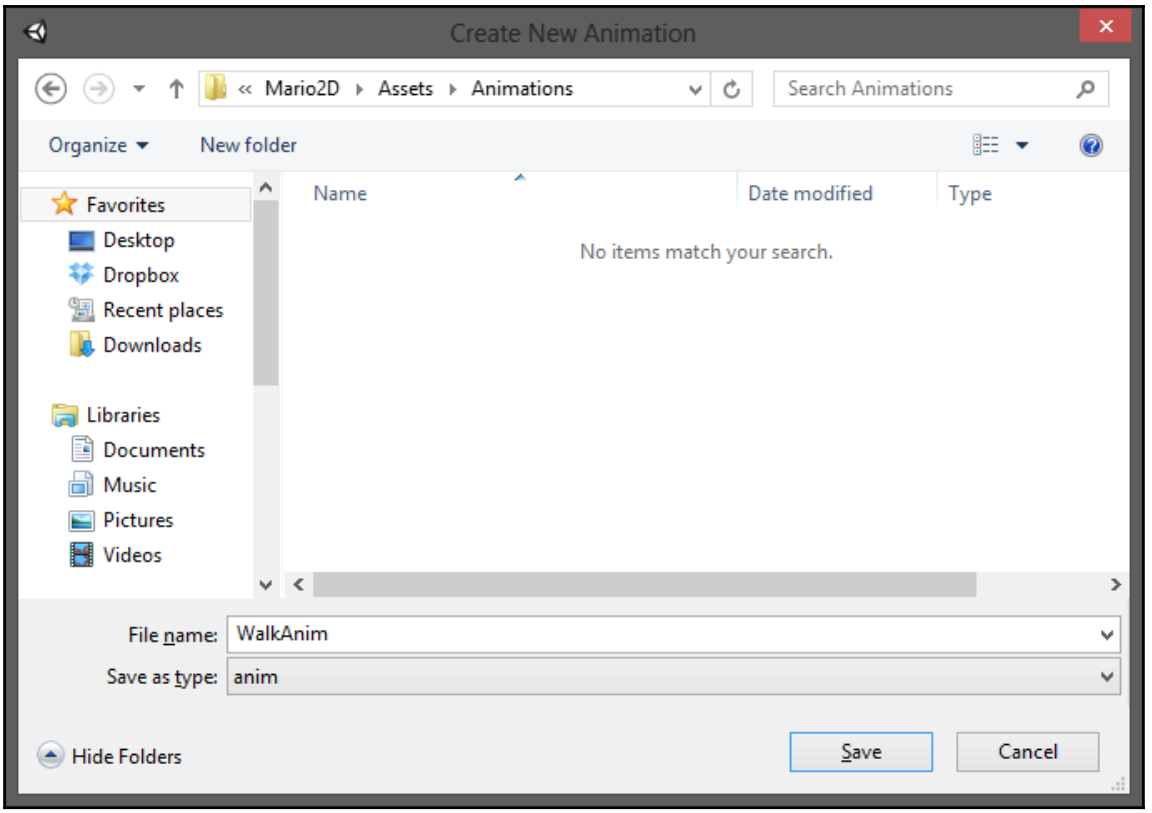
This is the recommended method for creating 2D animations. Here, Unity is able to create the entire animation for you with a single click.

If you navigate in the **Project Panel** to **Platformer Pack** | **Player** | **p1_walk**, you can find an animation sheet as a single file `p1_walk.png` and a folder of a PNG image for each frame of the animation. We will use the latter. The reason for this is because the single sprite sheet will not work perfectly as it is not optimized for Unity.

In the **Project Panel**, create a new folder and rename it as `Animations`. Then, select all the PNG images in **Platformer Pack** | **Player** | **p1_walk** | **PNG** and drop them into the **Hierarchy** Panel:

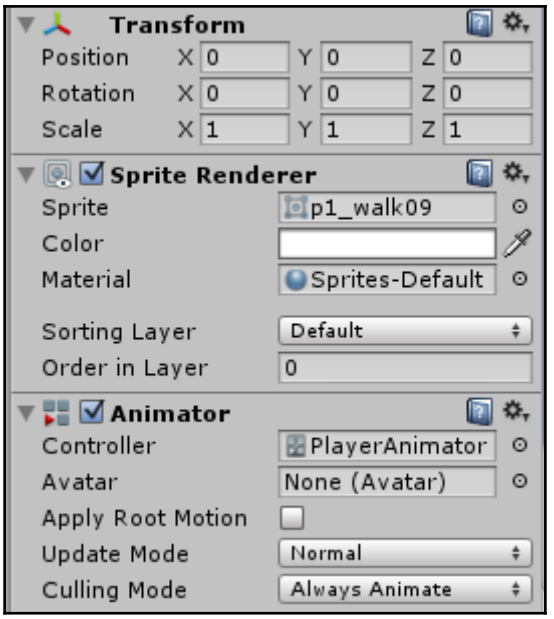


A new window will appear that will give us the possibility to save them as a new animation in a folder that we choose. Let's save the animation in our new folder, titled `Animations`, as `WalkAnim`:



After saving the animation, look in the **Project Panel** next to the animation file. Now, there is another asset with the name of one of the dropped sprites. This is an `Animator Controller` and, as the name suggests, it is used to control the animation. Let's rename it to `PlayerAnimator` so that we can distinguish it later on.

In the **Hierarchy** panel, a game object has been automatically created with the original name of our controller. If we select it, the **Inspector** should look like the following:



You can always add an Animator component to a game object by clicking on **Add Component** | **Miscellaneous** | **Animator**.

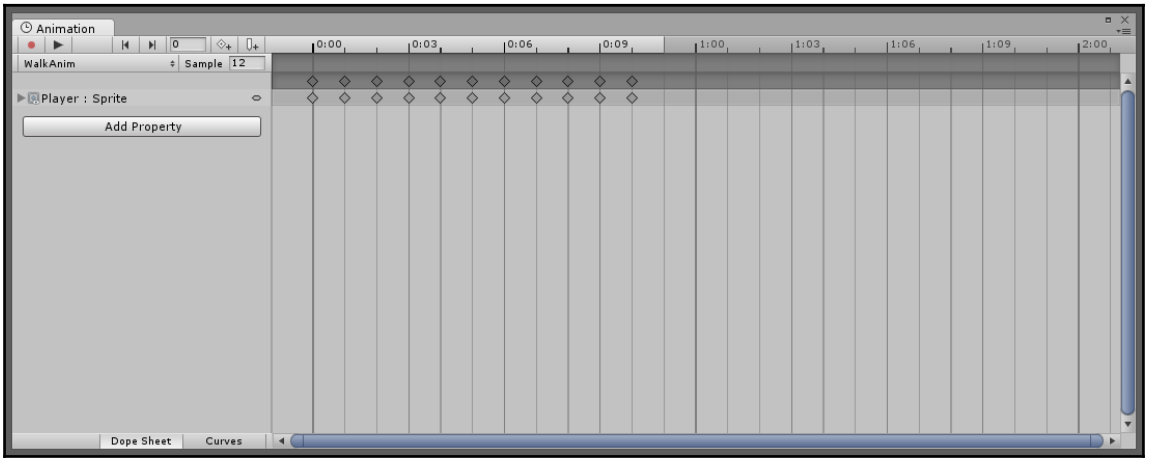
As you can see, below the **Sprite Renderer** component there is an **Animator** component. This component will control the animation for the player and is usually accessed through a custom script to change the animations. We will do this later on. So, for now, delete this new object that we have created, since we will use our character from the previous chapter.

For now, drag and drop the new controller `PlayerAnimator` on to our `Player` object.

Manual clip creation

Now, we also need a jump animation for our character. However, since we only have one sprite for the player jumping, we will manually create the animation clip for it.

To achieve this, select the `Player` object in the **Hierarchy** panel and open the **Animation** window from **Window** | **Animation**. The **Animation** window will appear, as shown in the following image:

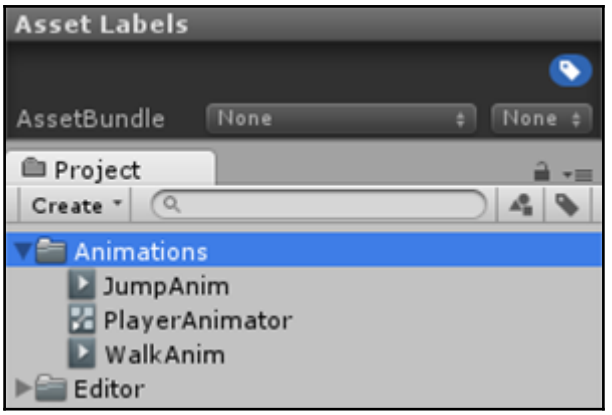


As you can see, our animation `WalkAnim` is already selected. To create a new animation clip, click on where you see the text `WalkAnim`. As a result, a drop-down menu appears and here you can select **Create New Clip**. Save the new animation in the `Animations` folder as `JumpAnim`.

On the right, you can find the animation timeline. Select from the **Project Panel** the folder `Platformer Pack/Player`. Drag and drop the sprite `p1_jump` on the timeline. You can see that the timeline for the animation has changed. In fact, now it contains the jumping animation, even if it is made out of only one sprite. Finally, save what we have done so far.

The **Animation** window's features are best used to make fine tunes for the animation or even merging two or more animations into one.

Now the `Animations` folder should look like this in the **Project** panel:



By selecting the `WalkAnim` file, you will be able to see the **Preview** panel, which is collocated at the bottom of the **Inspector** when an object that may contain animation is selected. To test the animation, drag the `Player` object and drop it in the **Preview** panel and hit play:

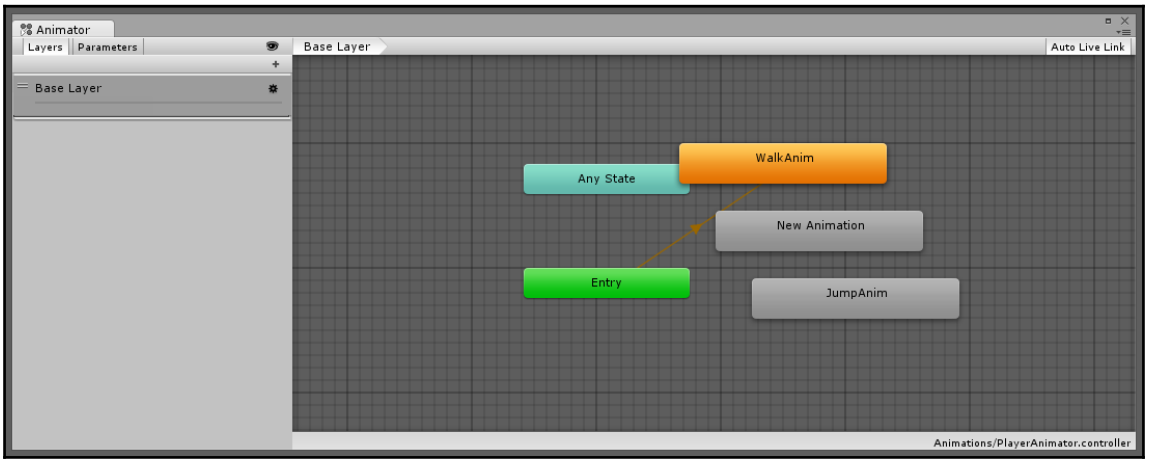
In the **Preview** panel, you can check out your animations without having to test them directly from code. In addition, you can easily select the desired animation and then drag the animation into a game object with the corresponding `Animator Controller` and drop it into the **Preview** panel.

The Animator

In order to display an animation on a game object, you will be using both `Animator Components` and `Animator Controllers`. These two work hand in hand to control the animation of any animated object that you might have, and are described below:

- `Animator Controller` uses a state-machine to manage the animation states and the transitions between one another, almost like a flow chart of animations.
- `Animator Component` uses an `Animator Controller` to define which animation clips to use and applies them on the game object when needed. It also controls the blending and the transitions between them.

Let's start modifying our controller to make it right for our character animations. Click on the `Player` and then open the Animator window from **Window** | **Animator**. We should see something like this:

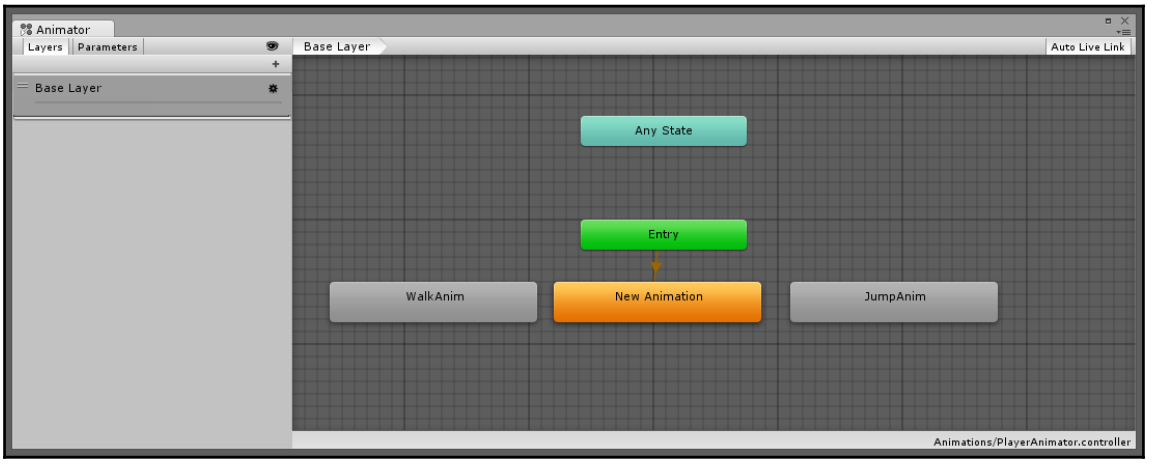


Although it is automatically generated, this is a state machine. To move around the grid, hold the middle mouse button and drag around.

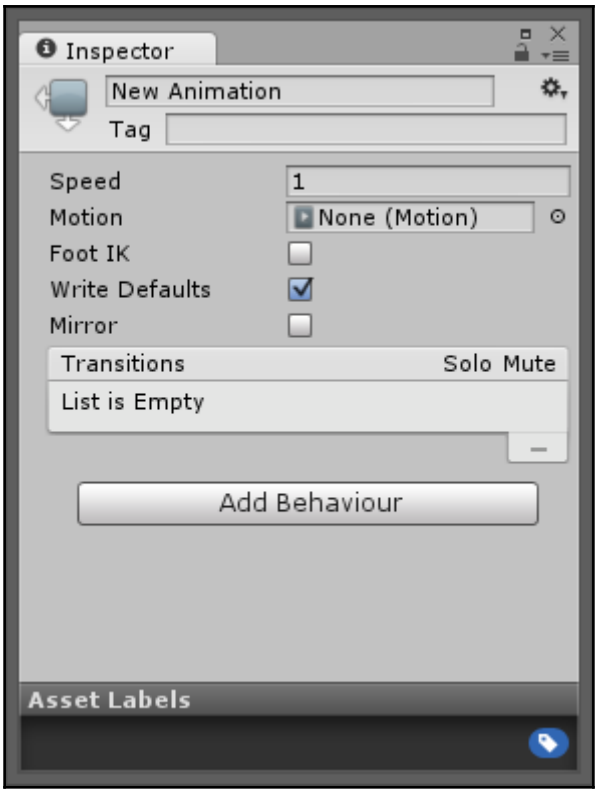
First, let's understand how all the different kinds of nodes work:

- **Entry node** (marked green): This is used when transitioning into a state machine, provided the required conditions were met.
- **Exit node** (marked red): This is used to exit a state machine when the conditions have been changed or completed. By default it is not present, as there isn't one in the previous image.
- **Default node** (marked orange): This is the default state of the **Animator** and is automatically transitioned to from the entry node.
- **Sub-state nodes** (marked grey): These are also called custom nodes. They are used typically to represent a state for an object where an event will occur (in our case, an animation will be played).
- **Transitions** (arrows): These allow state machines to switch between one another by setting the conditions that will be used by `Animator` to decide which state will be activated.

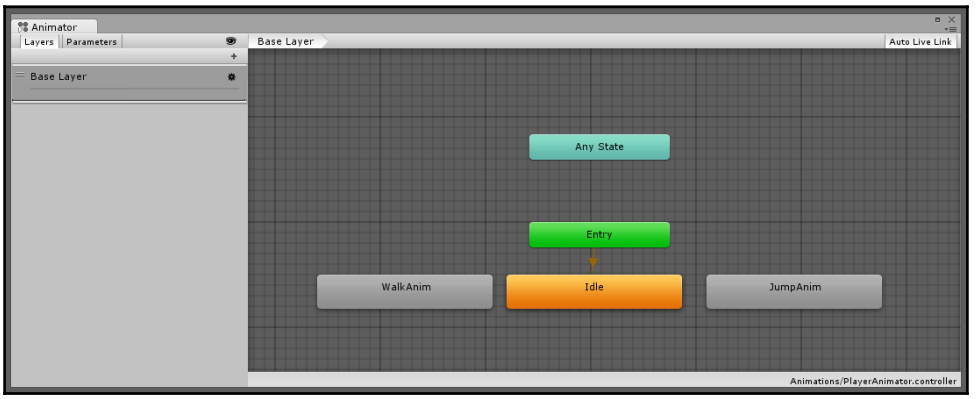
To keep things organized, let's reorder the nodes in the grid. Drag the three sub-states right under the **Entry** node. Order them from left to right **WalkAnim**, **New Animation**, and **JumpAnim**. Then, right-click on **New Animation** and choose **Set as Layer Default State**. Now, our **Animator** window should look like the following:



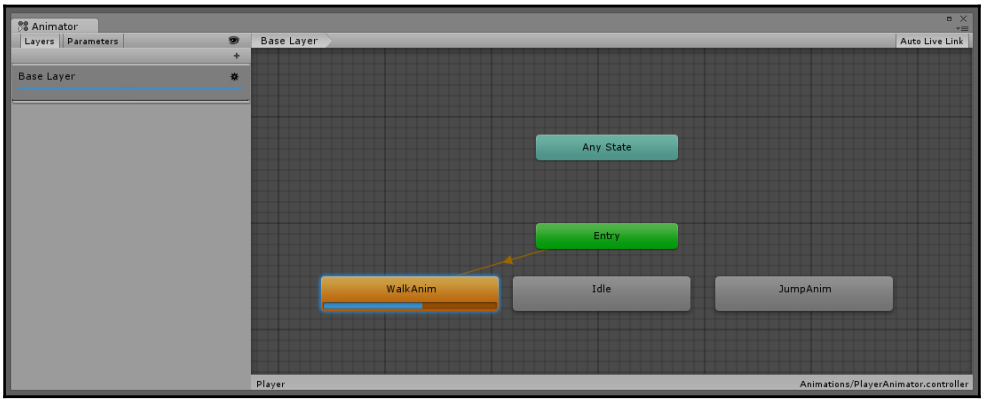
To edit a node, we need to select it and modify it as needed in the **Inspector**. So, select **New Animation,** and the **Inspector** should look like the following image:



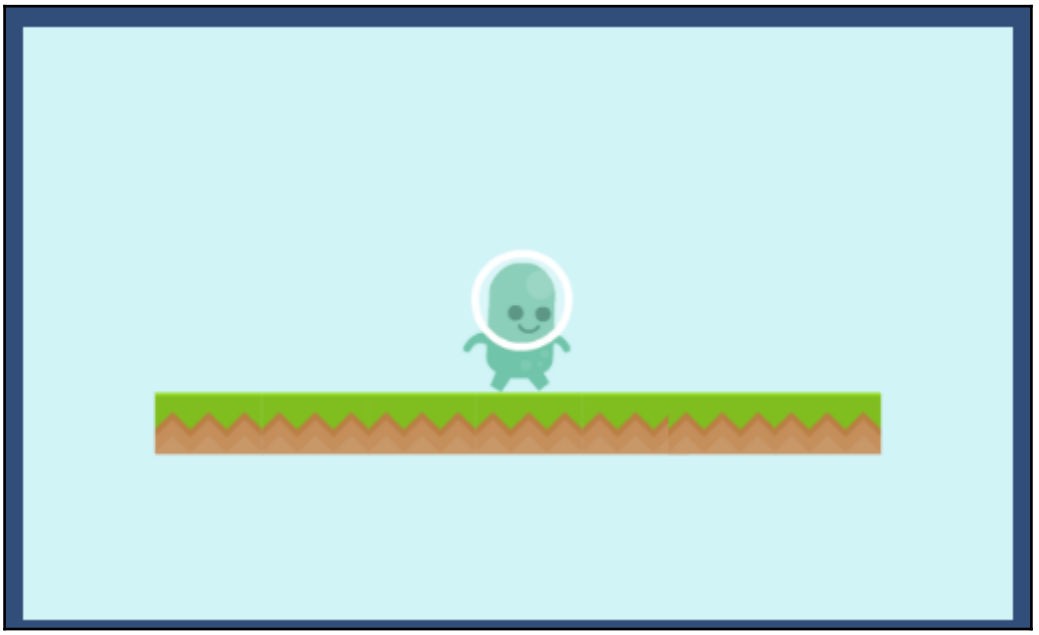
Here, we can have access to all the properties of the state or node `New Animation`. Let's change its name to `Idle`. Next, we need to change the speed of the state machine, which controls how fast the animation will be played. Next, we have **Motion,** which refers to the animation that will be used for this state. After we have changed the name, save the scene, and this is what everything should look like now:



We can test what we have done so far, by hitting play. As we can see in the **Game** view, the character is not animated. This is because the character is always in the `Idle` state and there are no transitions to let him change state. While the game is in runtime, we can see in the Animator window that the `Idle` state is running. Stop the game, and right-click on the `WalkAnim` node in the **Animator** window. Select from the menu `SetasLayerDefaultState`. As a result, the walking animation will be played automatically at the beginning of the game.



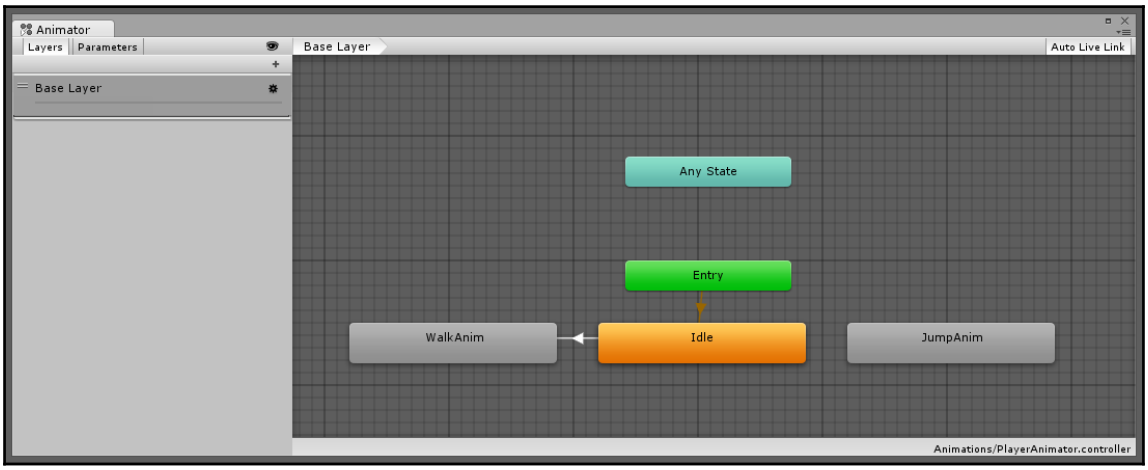
If we press the play button again, we can see that the walk animation is played, as shown in the following image:



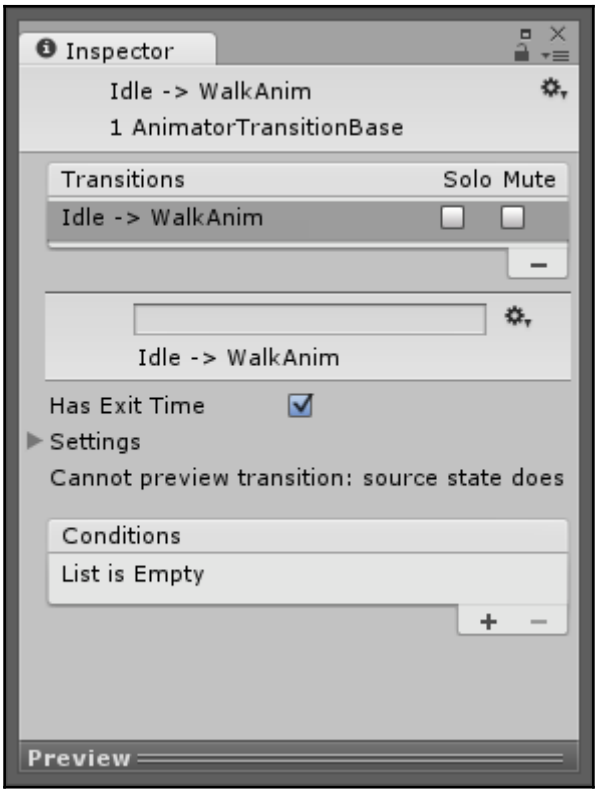
You can experiment with the other states of the `Animator`. For example, you can try to set `JumpAnim` as the default animation, or even tweak the speed of each state to see how they will be affected.

Now that we know the basics of how the `Animator` works, let's stop the playback and revert the default state to the `Idle` state.

To be able to connect our states together, we need to create transitions. To achieve this, right-click on the `Idle` state and select **Make Transition,** which turns the mouse cursor into an arrow. By clicking on other states, we can connect them with a transition. In our case, click on the `WalkAnim` state to make a transition from the `Idle` state to the `WalkAnim` state. The animator window should look like the following:



If we click on the arrow, we can have access to its properties in the **Inspector**, as shown in the following image:

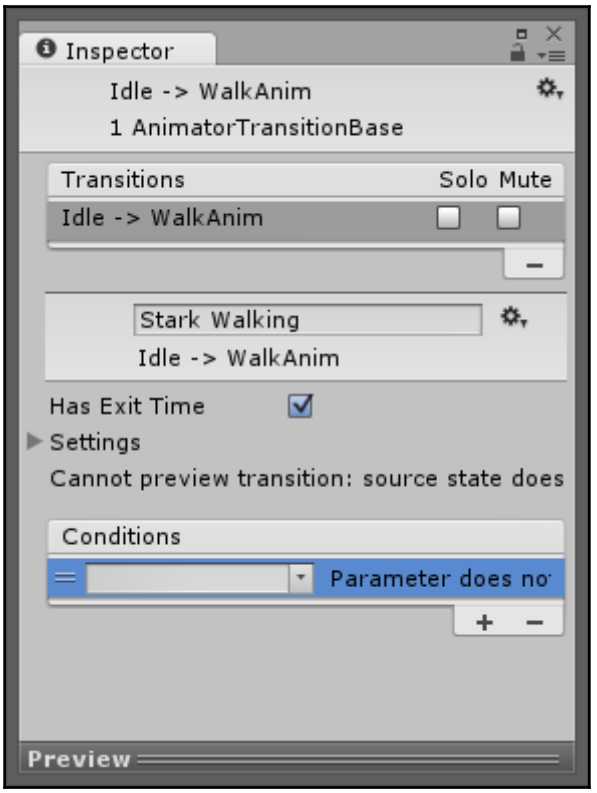


The main properties that we might want to change are:

- **Name** (optional): We can assign a name to the transition. This is useful to keep everything organized and easy to access. In this case, let's name this transition `StartWalking`.
- **Has Exit Time**: Whether or not the animation should be played to the end before exiting its state when the conditions are not being met any more.
- **Conditions**: The conditions that should be met so that the transition takes place.

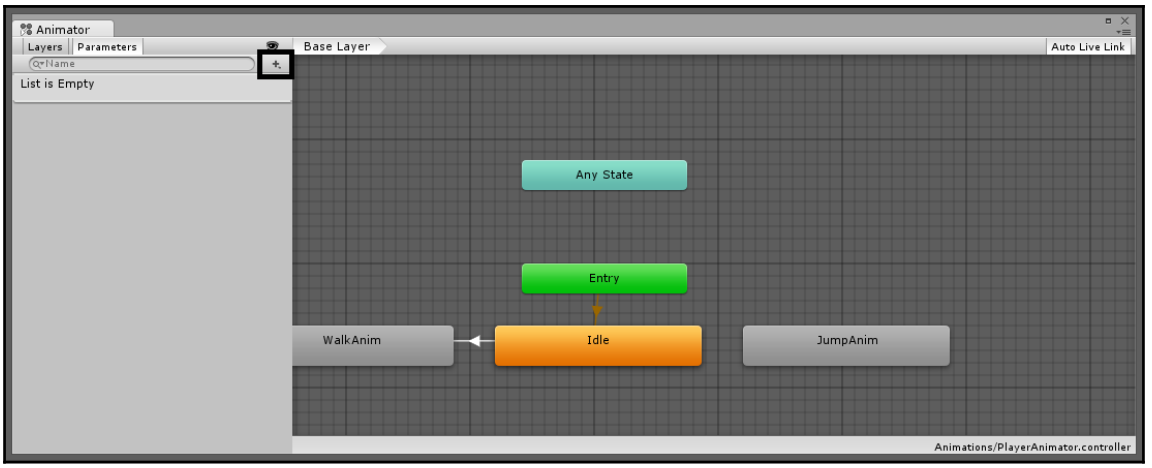
www.allitebooks.com

Let's try adding a condition and see what happens:

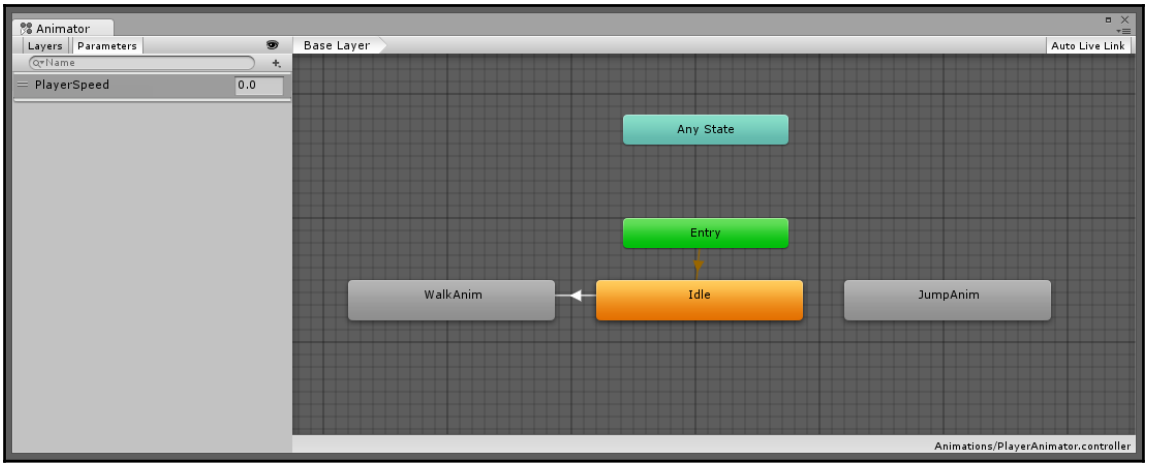


When we try to create a condition for our transition, the following message appears next to **Parameter does not exist in Controller** which means that we need to add parameters that will be used for our condition.

To create a parameter, switch to `Parameters` in the top left of the **Animator** window and add a new float using the **+** button and name it `PlayerSpeed`, as shown in the following image:



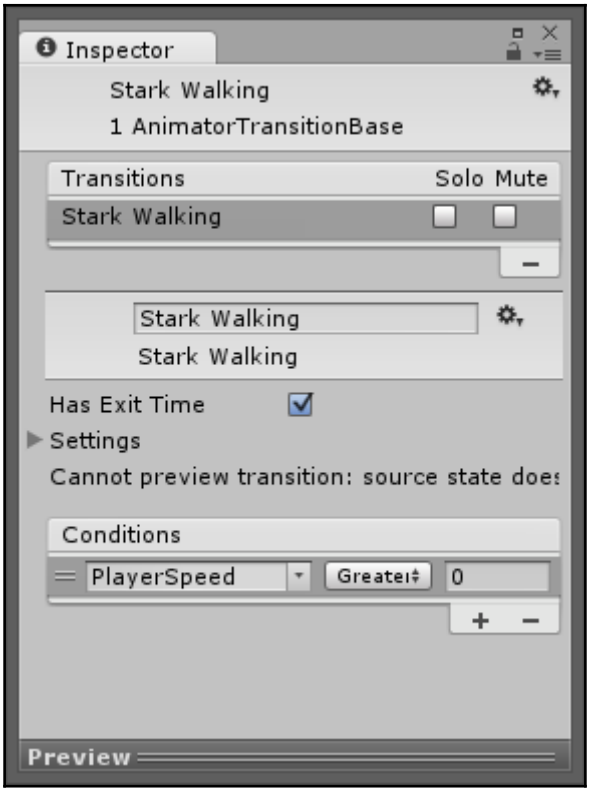
Any parameters that are created in the `Animator` are usually changed from code and those changes affect the state of animation. In the following image, we can see the `PlayerSpeed` parameter on the left side:



Now that we have created a parameter, let's head back to the transition.

Click the drop down button next to the condition we created earlier and choose the parameter **PlayerSpeed**. After choosing the parameter, another option appears next to it. You can either choose **Greater** or **Less**, which means that the transition will happen when this parameter is respectively less than *X* or greater than *X*. Don't worry, as that *X* will be changed by our code later on.

For now, choose **Greater** and set the value to `1`, which means that when the player speed is more than one, the walk animation starts playing.

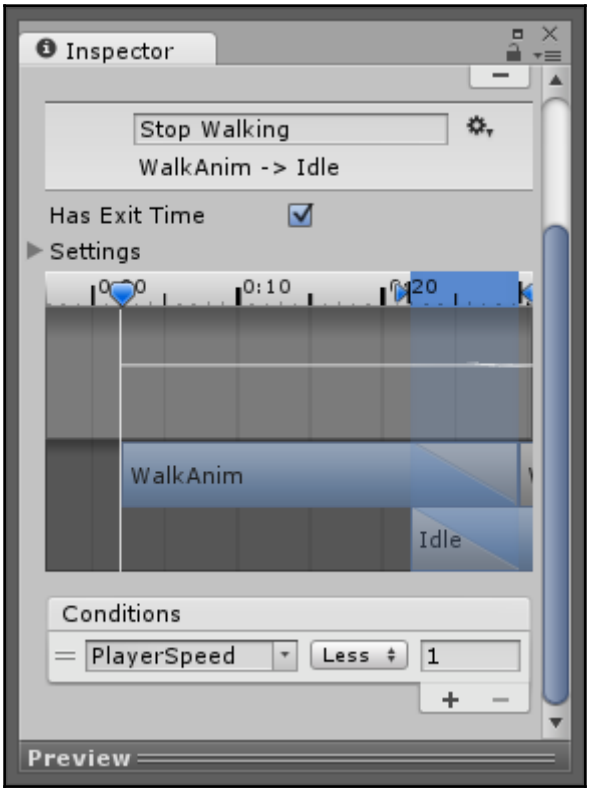


You can test what we have done so far and change the **PlayerSpeed** parameter in runtime.

The game

In this chapter, the goal for our game is to apply the walk animation when the player is moving. Furthermore, we will prepare the jump animation for the next chapter. To achieve that, we need to set up the rest of the transitions.

In the **Animator** window, let's add a new transition from `WalkAnim` to `Idle` and name it `StopWalking`. Then, we need to add a condition that will trigger the transition. This transition will use the parameter **PlayerSpeed** so that when its value becomes less than one the transition is triggered.

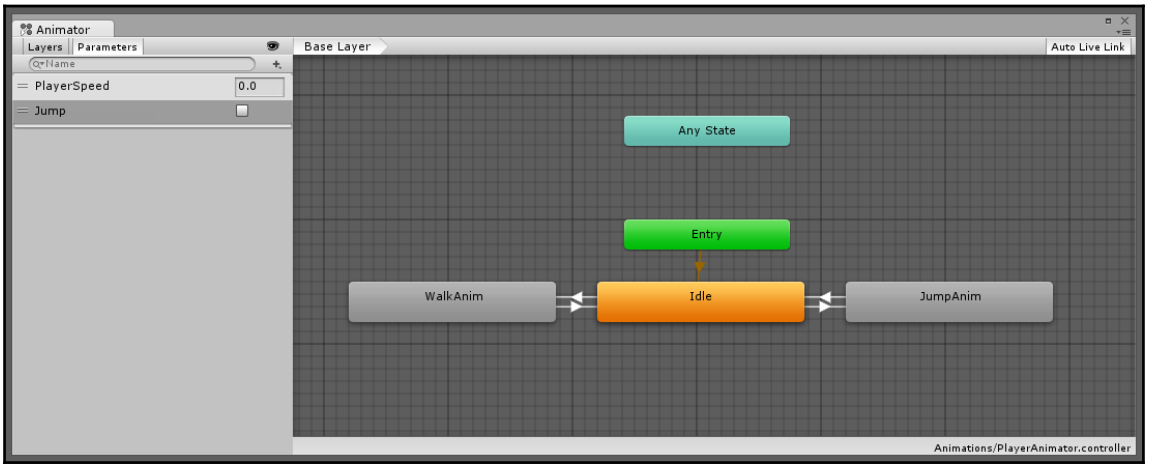


Now that the transitions between `Idle` and `WalkAnim` are ready, let's do the same to the `JumpAnim` state.

We need to make a transition from the `Idle` state to the `JumpAnim` state and vice-versa; we will name them `StartJump` and `EndJump`, respectively.

However, we are going to need another parameter to see if the player is currently jumping. We can do this by adding another parameter named `Jump` as a Boolean in the **Animator** window and set it to `false`.

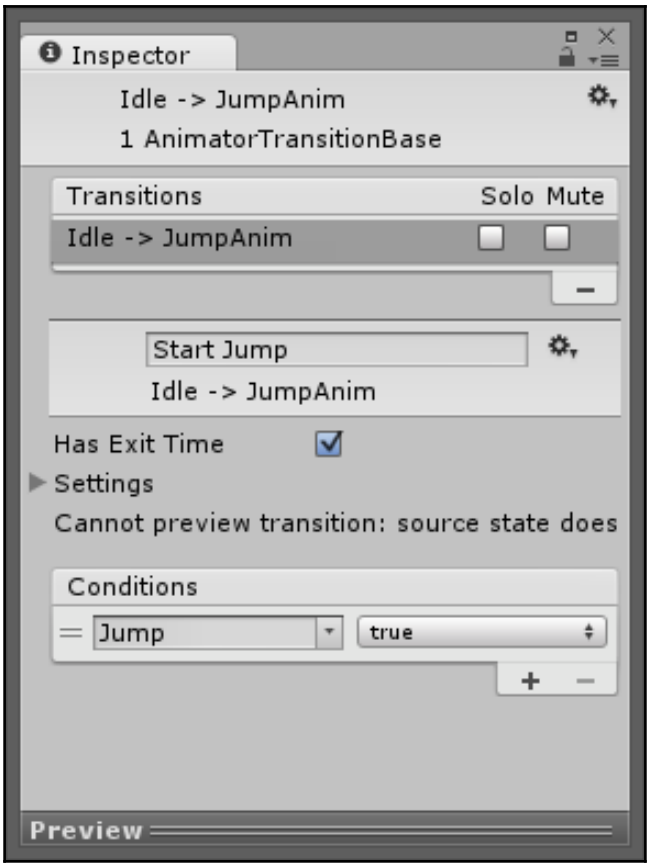
This is what the **Animator** window should look like right now:

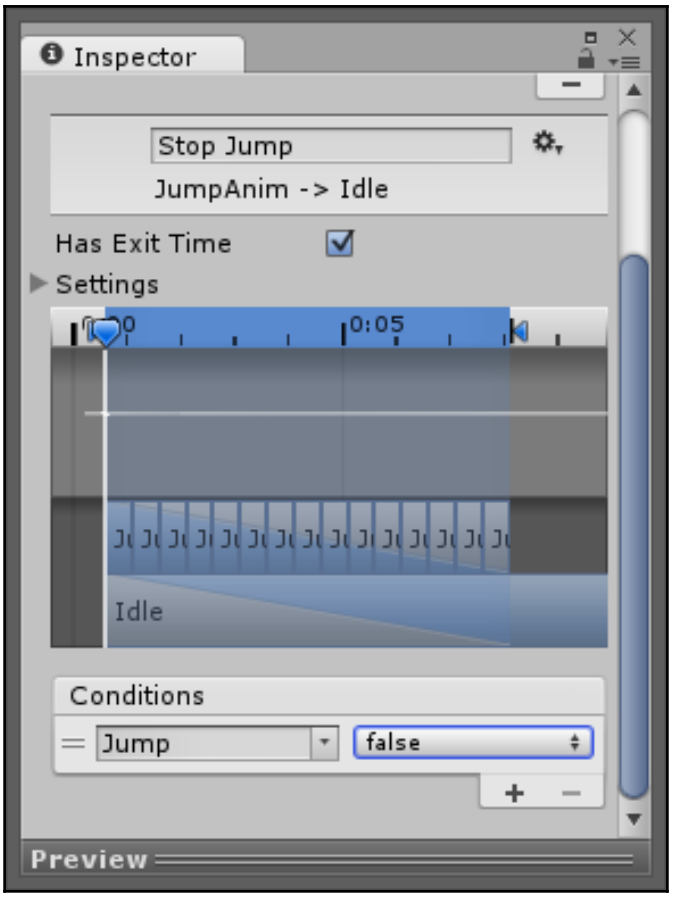


After adding the new `Jump` parameter we can configure the two new transitions for `JumpAnim`:

- For the first transition, name it `Start Jump`, add a condition, and choose the `Jump` parameter with its value as `true`
- For the second transition, name it `Stop Jump`, add a condition, and choose the `Jump` parameter with its value as `false`

What this means is that when the `Jump` variable is set to true, our player will play the jump animation and then it will stop once the variable changes to `false`, which will be done through code.





A good edit to do in this instance is to change the **Has Exit Time** variable in all the transitions to `false` in order to avoid waiting for the states to finish their transitions.

The next step is to apply the walk animation to the player movement and prepare the jump controls. In the project panel, double-click on the script we created in the previous chapter, `PlayerMovement`, inside the `Scripts` folder to open it.

We need to add another variable to store the reference to the `Animator Component` attached to the `Player` game object:

```
//Reference to the Animator
private Animator anim;
```

In the `Start()` function, we need to get the reference to the Animator by calling the `GetComponent()` function:

```
    void Start() {
        anim = GetComponent<Animator>();
    }
```

We need to change the `Update()` function so that we can take care of the `Animator` and change its parameters accordingly. As a result, our character will be animated. Let's start to add this piece of code between the two lines of code we had before, just before the `if`-statement:

```
        float translation = Input.GetAxis("Horizontal") * speed *
Time.deltaTime;

        //Change direction if needed
        if (translation > 0) {
            transform.localScale = new Vector3(1, 1, 1);
        }
        else if (translation < 0) {
            transform.localScale = new Vector3(-1, 1, 1);
        }

        // Move along the object's x-axis within the floor bounds
// ...
```

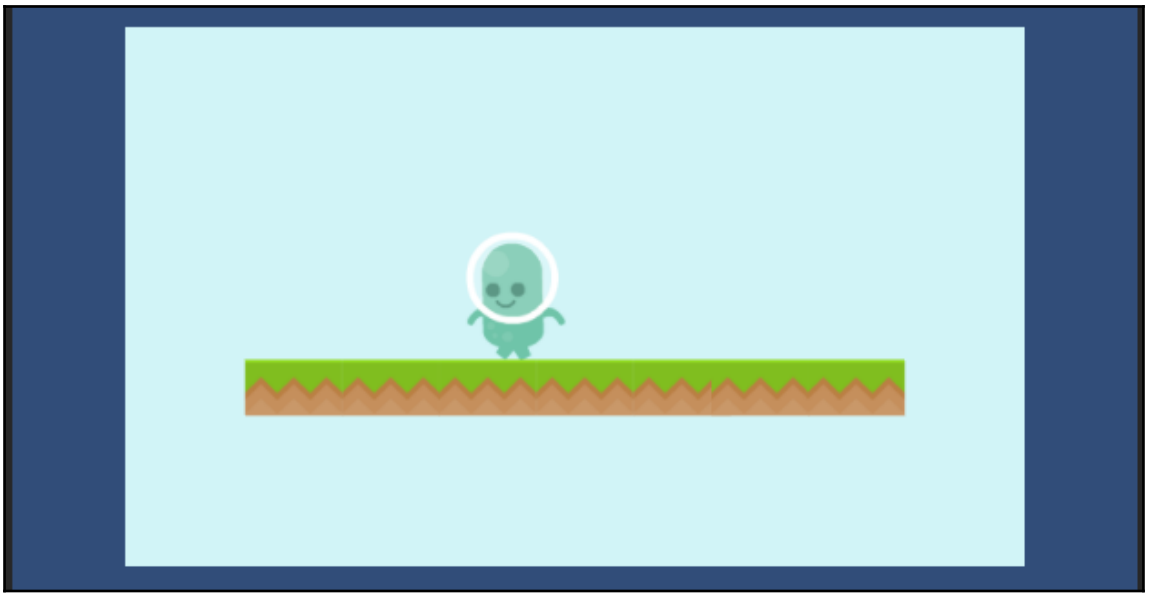
In this way, we flip the character according to its direction. Now, we need to take care of the `Animator` so it can change the animation accordingly. To do this, let's add this at the end of the `Update()` function:

```
        // Switching between Idle and Walk states in the animator
        if (translation != 0) {
            anim.SetFloat("PlayerSpeed", speed);
        }
        else {
            anim.SetFloat("PlayerSpeed", 0);
        }
```

Furthermore, we may want to switch to the `Jump` animation. For the sake of learning, we can just switch between the `Jump` animation and the walking animation every time the player presses the space bar. So, after what we have written before, let's add:

```
        // Switching between Jump and Walk animation
        if (Input.GetKeyUp(KeyCode.Space)) {
            anim.SetBool("Jump", !(anim.GetBool("Jump")));
        }
```

Save the changes and press the play button. When moving the player using the left and right arrow (alternatively *A* and *D*), you see the character playing the `Walk` animation. Also, when pressing the `Jump` (space) button, the character changes from the `Idle` sprite to the `Jump` sprite (`Animation`). Here is a screenshot of the game while playing after saving the changes:



Summary

This wraps up everything that we will cover in this chapter. So far, we have added animations to our character to be played according to the player controls. A brief summary of what we have discussed would be creating animations, animator controllers, state machines for `Animation`, and using animations in our game through the use of `Scripts`.

In the next chapter, we will make things even more realistic by using physics and applying it to our game.

3
Physics

Adding physics to any game adds realism and makes it more convincing to the player. You can still make games without the use of physics, but you will be giving up an opportunity to make your game a lot more awesome!

In this chapter, we will learn how to apply Unity's 2D physics to our game and cover some of the basics of Unity, such as:

- Getting to know colliders
- Controlling the character
- Continuing our game

2D physics

Unity's 2D physics engine is very similar to its 3D one. Almost all of the physics components have been integrated into the 2D engine with a slight difference in names (Box Collider 2D, Circle Collider 2D, Rigidbody 2D, and so on…).

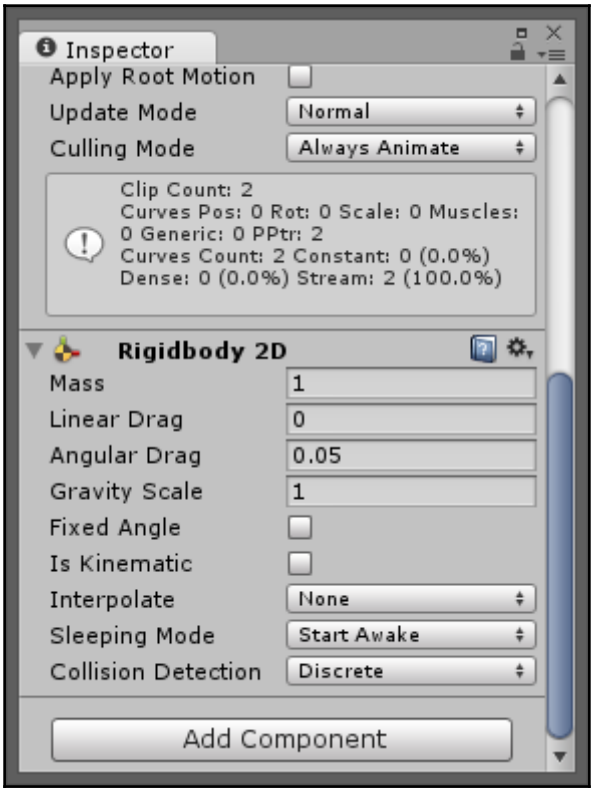
It is important to understand that 2D physics components will not interact with 3D physics components if both exist within the same scene. Even though they both share a lot of similarities, 2D physics only occur on the *X* and *Y* axis, and rotating an object using physics will only occur on the *X* axis.

If you wish to change any of the settings of the 2D physics engine, you can find all of its properties by navigating to **Edit** | **Project Settings** | **Physics 2D**.

Rigid bodies

Just like the 3D engine, the `Rigidbody2D` component controls the physical behavior of any game object it is attached to inside the scene. It also defines its physical properties, such as mass, and how gravity should affect an object.

To begin with rigid bodies, let's add this component to our `Player` object by clicking on **Add Component** | **Physics 2D** | **Rigidbody 2D**. The **Inspector** should look like the following:

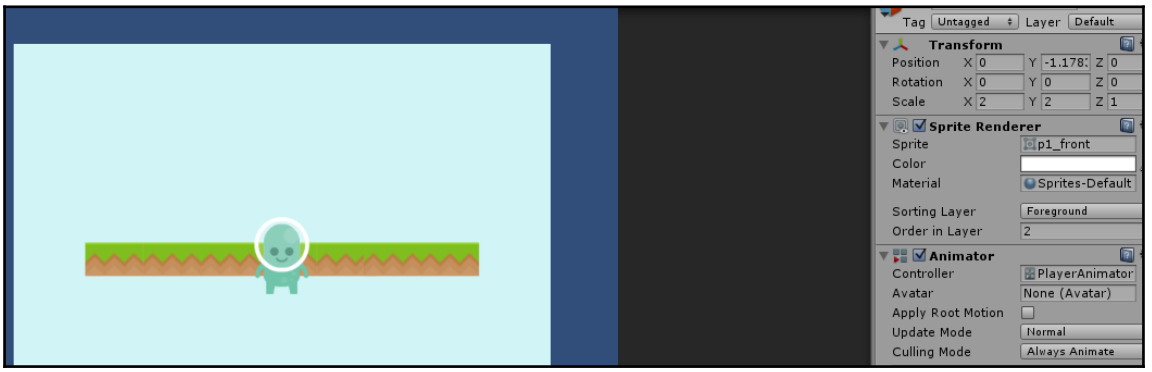


Let's see the main features of this component:

- **Mass**: Mass of the rigid body; the greater the value the greater the force that will be required to move the object.
- **Liner Drag**: This is the amount of friction that a force has to work against to make an object move.

- **Angular Drag**: This is the amount of rotational friction that a force has to work against to make an object rotate.
- **Gravity Scale**: This is the amount of gravity that affects a game object. The greater the value, the stronger the gravitational force.
- **Fixed Angle**: When marked `true`, the rigid body will continue to respond normally to the physics forces, but without rotating.
- **Is Kinematic**: When marked true, the object will not respond to any physics forces around it. This is typically used when an object needs a special type of physics behavior that can be created by a custom script.

In order to test the effects of the component, let's press the play button. At the moment, our player keeps falling down passing through every object on its way, as we can see in the following image:



To prevent this from happening, we need to start using Unity's 2D colliders!

Colliders 2D

In order to define the dimension of an object in the scene, we need to add a collider component that defines its shape and that reacts if a rigid body is attached to it.

There are four types of 2D colliders in Unity:

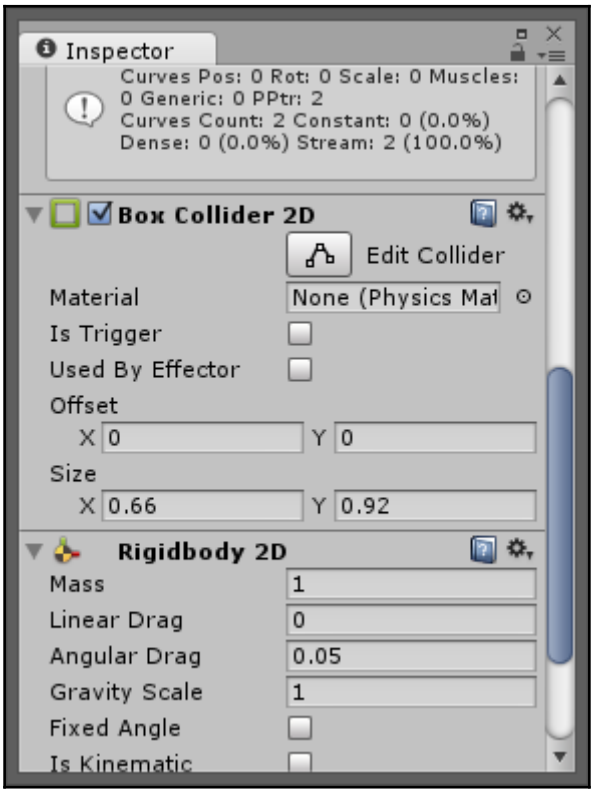
- Box Collider 2D: This type of collider works better with rectangular objects
- Circle Collider 2D: As given by its name, this collider works better with circular objects

- Polygon Collider 2D: Its shape is defined by a freeform edge made by line segments that surround the sprite
- Edge Collider 2D: It is used to define a surface without using a series of other colliders

When the first three colliders are added to a game object, they are automatically adjusted to best fit the sprite attached to the object.

Box Collider 2D

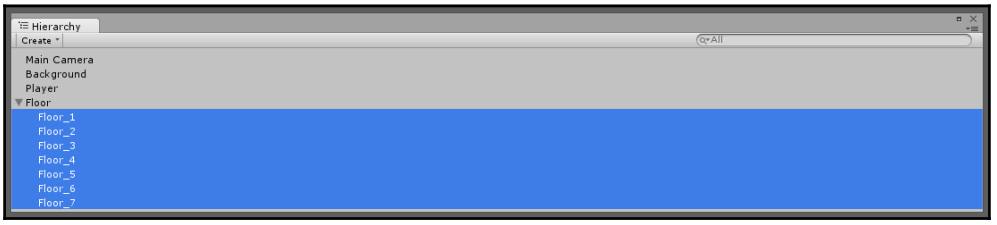
Let's try to add a box collider to the objects in our scene and see what happens. Similar to adding a rigid body, we can add a collider by clicking on **Add Component** | **Physics 2D** | **BoxCollider2D**. We can start to add this component to the `Player` object. The **Inspector** should now look as follows:



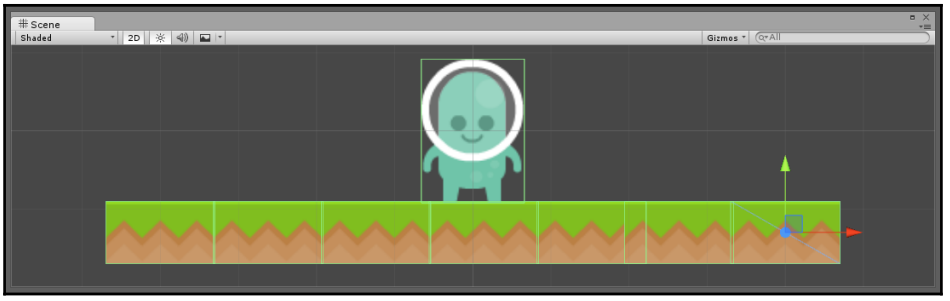
Let's explore the main features of this component:

- **Edit Collider**: This is a button that allows us to manually adjust the shape of the collider. This can also be done in the **Scene** view.
- **Material**: This is a reference to the 2D physics material that will define the object's behavior when colliding with other objects.
- **Is Trigger**: When checked, the collider will act as a trigger to fire events from the code.
- **Used By Effector**: If marked true, this collider will be used by an attached effector to define it. We will see them later in the chapter.
- **Offset**: This allows us to change the center or pivot point of the collider.
- **Size**: This allows us to change the size of the collider for each axis.

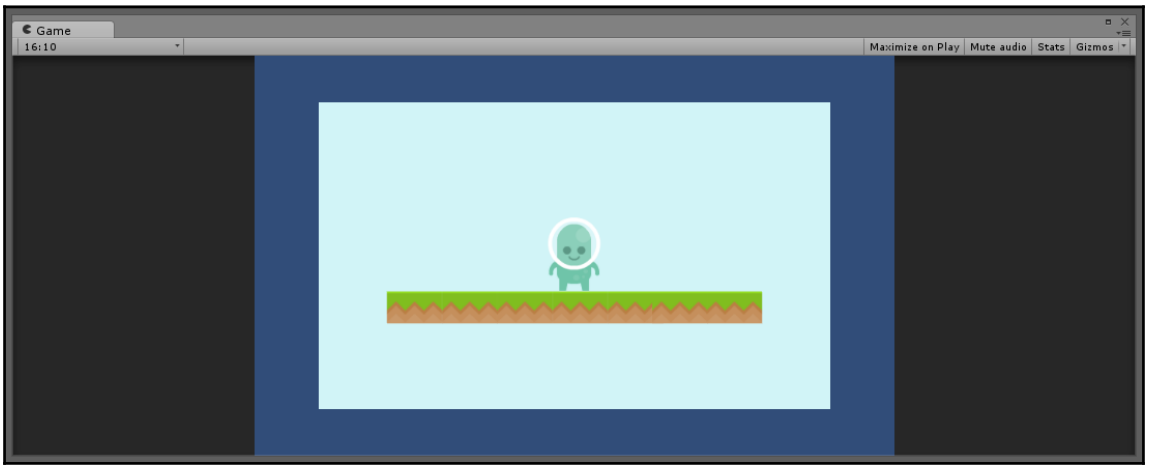
Now that the `player` object is physically defined, we should do the same for the floor tiles. We can avoid repeating this operation for each of them, by multi-editing them. We just need to select all the floor objects and then add the `BoxCollider2D` as we did before in the **Inspector**:



As a result, all of the objects in our game can interact using the Unity Physics engine. In fact, now, our player will not fall through the floor when walking on it. In the **Scene** view, the colliders look like green boxes surrounding the objects, as seen in the following image:



If we press the play button, we can see that our player doesn't fall through the floor anymore. However, he stands still in the middle of the scene without the possibility for us to control his movement. So, let's change that!



Letting the character move

In the following section, we will learn how to use the 2D controller and the physic engine of Unity to let our character move.

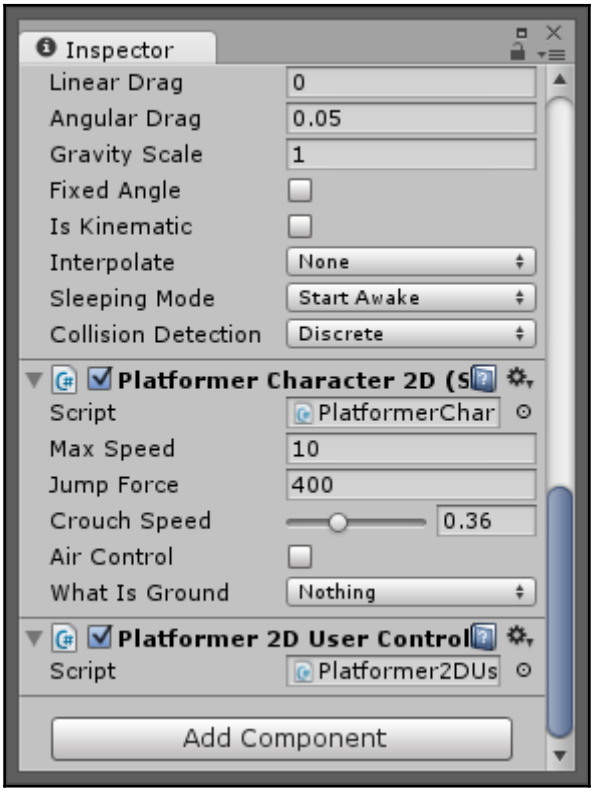
Adjusting the Platformer 2D controller

Before we start to add a new controller, we need to remove the previous one. So, select the `Player` object, and in the **Inspector**, right-click on the **PlayerMovement** script and then **Remove Component**. As a result, our script will no longer be attached to the `Player` object.

Since we are going to use the `Character Controller` inside the **Standard Assets**, we need to have it in our project. If you haven't downloaded it yet, you can do so by going in the **Asset Store**: **Window** | **Asset Store** or alternatively, *Ctrl* + *9*. If you have already downloaded them, but not imported them into this project, you can do so by by clicking on **Assets** | **Import Package** or again, using the **Asset Store**. At the end, you should have everything that you need in the `2D` folder.

Let's add the components `PlatformerCharacter2D` and `Platformer2DUserControl` to the `Player` object. You can use either the search tool in the **Project** panel and then drag and drop them, or click on **Add Component** | **Scripts** in the Inspector when the `Player` object is selected.

At the end, the `PlatformerCharacter2D` component should look as follows:



Let's see what the main features of this controller component are:

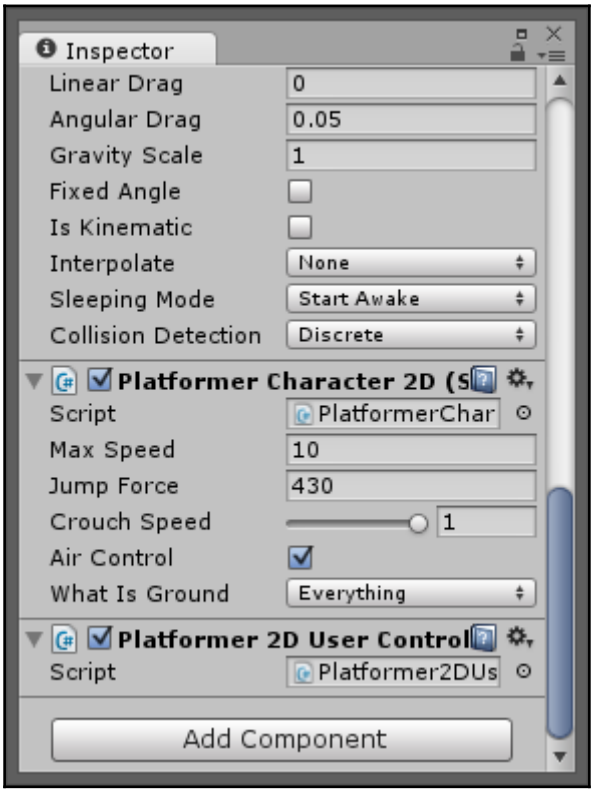
- **Max Speed**: This is the maximum speed that the player can reach when moving along the *X* axis
- **Jump Force**: This is the force that will be applied to the player's rigid body when he jumps
- **Crouch Speed**: This is the player's speed while crouching (crouching is done by pressing *Ctrl*)

- **Air Control**: If marked true, the player can also be controlled in air, when he is not being grounded
- **What Is Ground**: These are the layers that will be treated as the ground

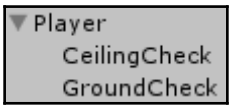
The `PlatformerCharacter2D` component makes use of the rigid body attached to the object to move the character in a Platformer fashion. It still allows us to use the other physical forces in the scene, making the character even more believable.

However, before we use it, we need to make some changes to the controller. As a result, we can achieve exactly what we want:

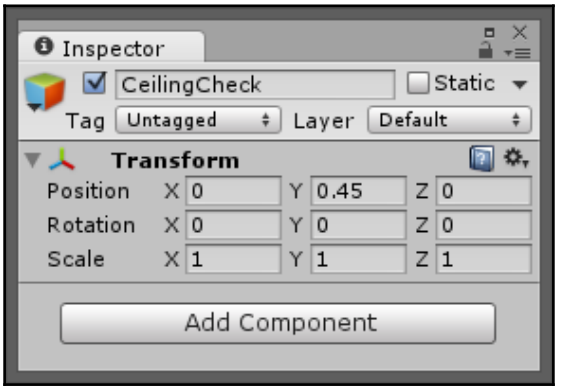
1. Let's start by changing the values in the `PlatformerCharacter2D` component to match the values shown in the following image:



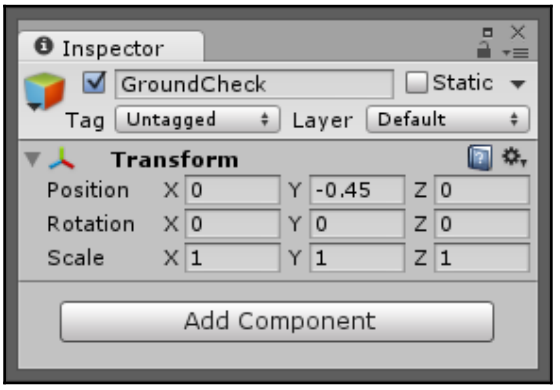
2. Now we need to define which are the top and bottom ends of the `Player` object. This is because they are used by the `PlatformerCharacter2D` component to understand the dimensions of the character. We can easily do this by adding two empty child game objects to the `Player`, as shown in the following image:



Now we also need to adjust their positions. Here are the **CeilingCheck** settings:

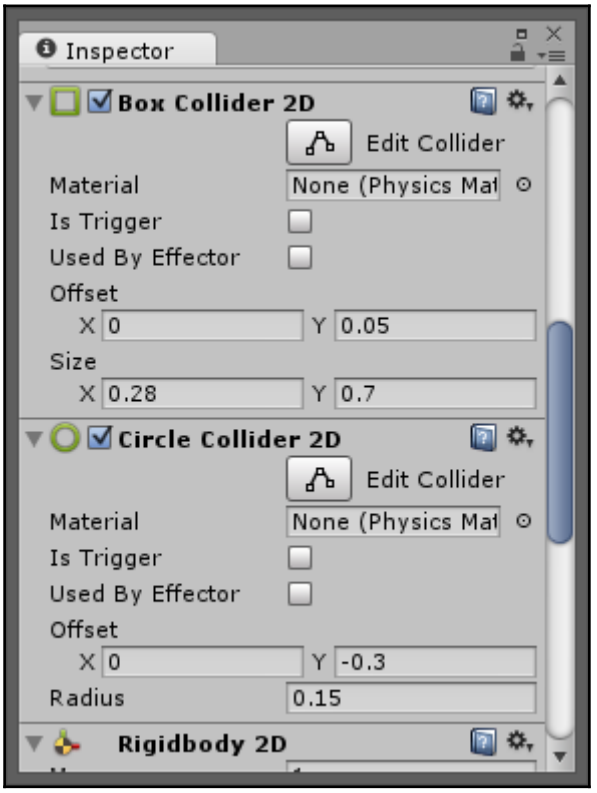


Here are the **GroundCheck** ones:

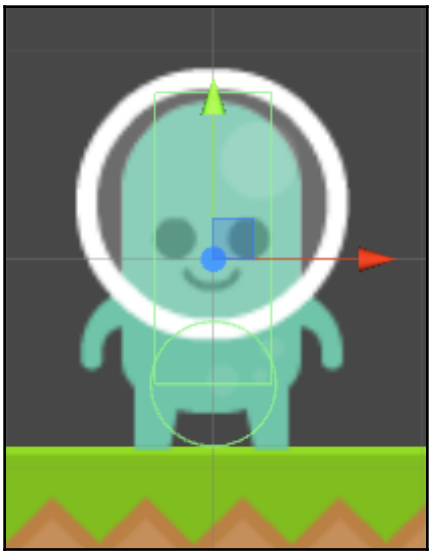


Defining a physical shape for the character

For the next step, we have to give a physical shape to our character. So, let's add another collider: `CircleCollider2D`. Now, we need to adjust the values on both the colliders that are attached to the player, as shown in the following image:



This is how the colliders will look in the **Scene** view after the changes:



Finally, in the `RigidBody2D` component, we need to change **Fixed Angle** to **True**. As a result, our character will not rotate. Also, in the **Interpolate** variable, choose the `Interpolate` method in order to smooth the character movement.

Improving the Animator

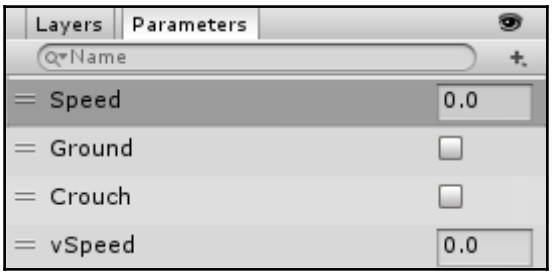
In the **Animator** component, choose `Animate Physics` inside the **Update Mode**. We use this option to keep the animator in sync with the `FixedUpdate` calls in the code. As a result, our character is animated properly.

In the next step, we have to change the Animator parameters in order to match the variables in the `PlatformerCharacter2D` class. Select the `Player` object and open the **Animator** window by double-clicking on it. Now, we need to apply the following changes:

1. Rename the parameter `PlayerSpeed` to `Speed`. This is the speed of the player.
2. Rename the parameter `Jump` to `Ground`. This stores information about whether the character is grounded or not.

3. Add a new parameter of type `Boolean` and name it `Crouch`. This stores if the player is crouched or not.
4. Add a new parameter of type `float` and name it `vSpeed`. This is the vertical speed.

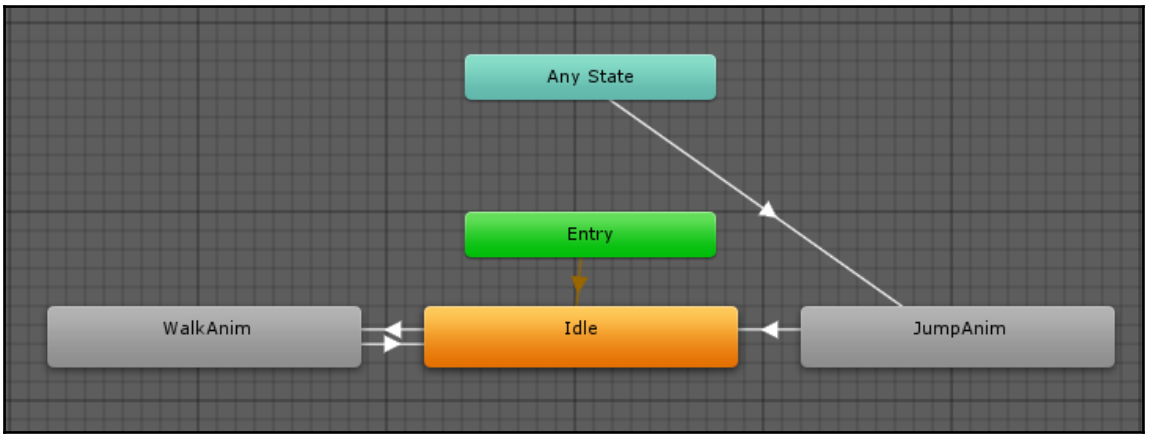
So, in the end, we should have the following parameters set in the **Parameters** tab:



Now that we have set the parameters, we can use them in our state machine. So, let's start to modify it by following these steps:

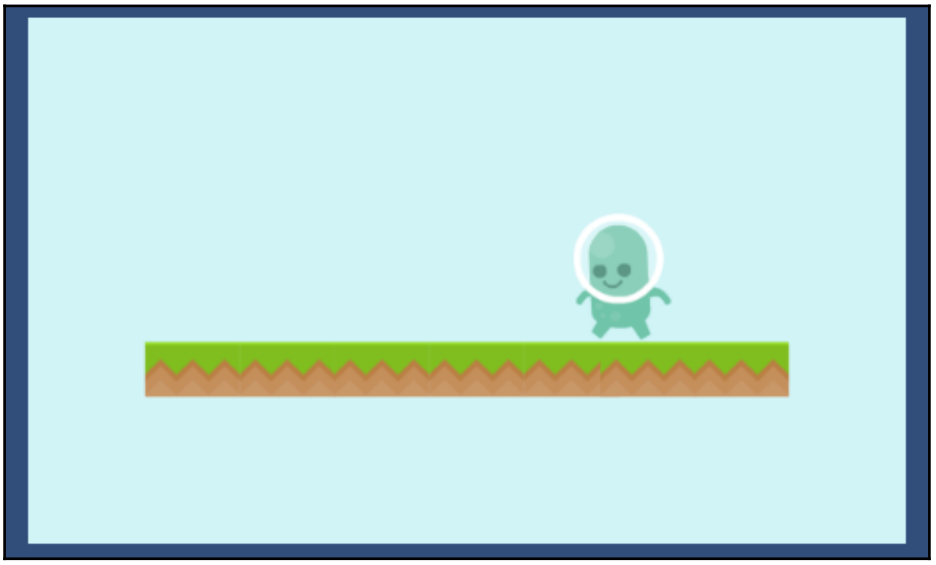
1. Delete the transition from the `Idle` state to the `Jump` state.
2. Add a new transition from `Any State` to `Jump` and name it `StartJump`. Then, add a new condition that uses the `Ground` parameter with the value `False`.
3. In the transition from `Jump` to `Idle`, change the **Transition Duration** to 0, so to minimize the time between the two states. Also, make sure that the condition uses the `Ground` parameter with the value `True`.
4. In the transition from `Idle` to `Walk`, the condition should use the **Speed** parameter with a value greater than `0.1`, and not the old parameter we have erased.
5. Finally, in the transition from `Walk` to `Idle`, again the condition has to use the **Speed** parameter with a value less than `0.1`.

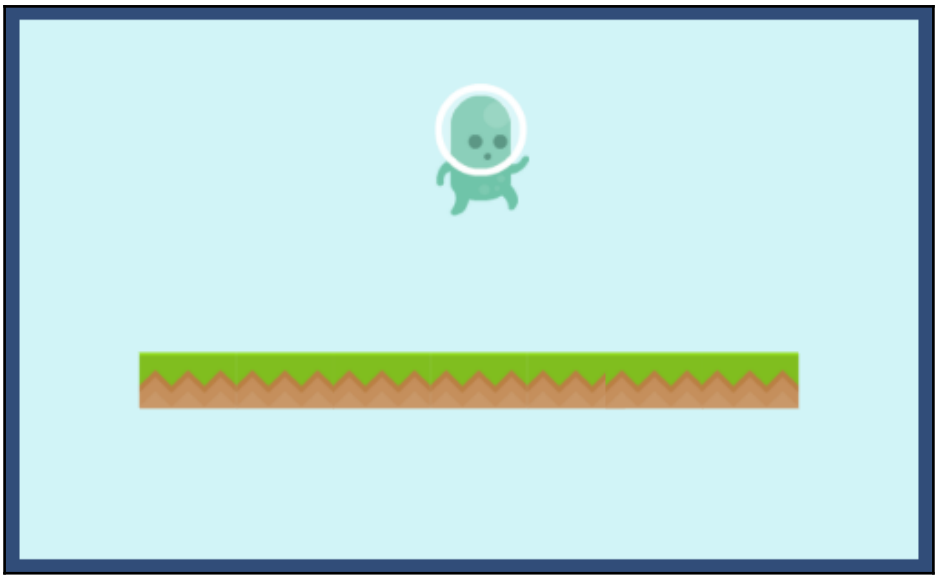
After we have made the above changes, the Animator screen should look as follows:



Testing the character movement

As the last step, we can press the play button and see what we have achieved so far. As you move the character, you can see that he moves along the *X* axis and he can jump!



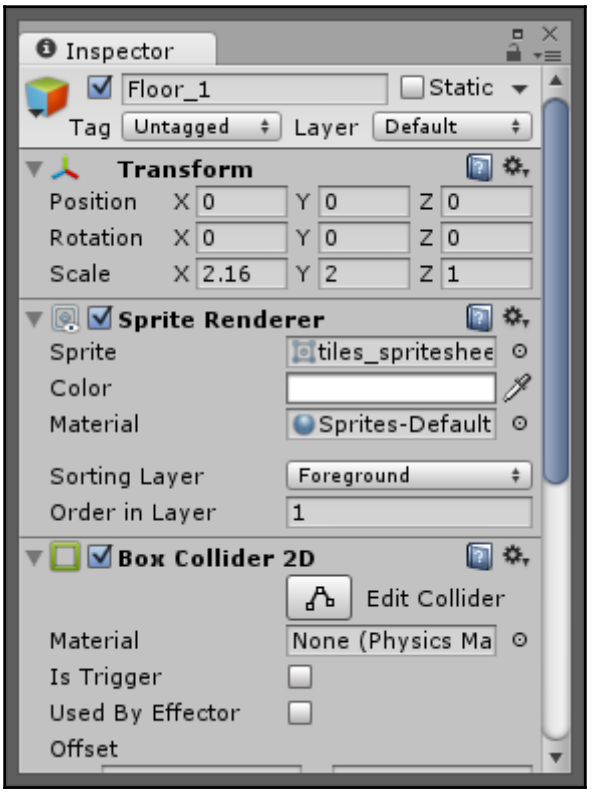


Building a cool level

Now that we have the character fully animated and controllable, it's time to build a cool level to let him move in. In order to organize the level, we can follow these steps:

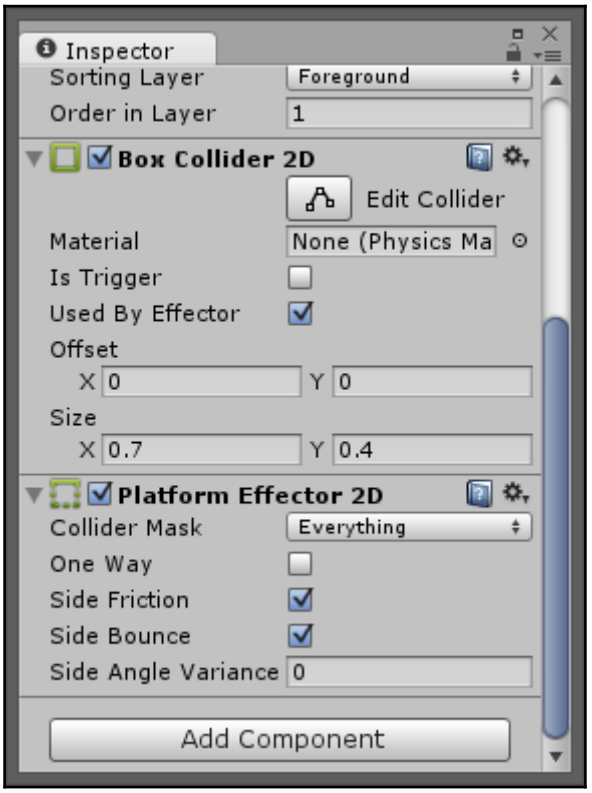
1. Delete both `Floor_1` and `Floor` game objects (since we start from scratch).
2. Create two new objects, `Level` and `Floors_1`. Reset their position (all zeros) and scale (all ones).
3. Create a `Floor_1` game object from the tile we want to use. Furthermore, set its scale to `(2.15,2,1)`. Finally, add a Box Collider 2D.
4. Parent them to each other in the following order: `Floor` | `Floors_1` | `Floor_1`, as shown in the following image:

This is what the **Inspector** should look like:



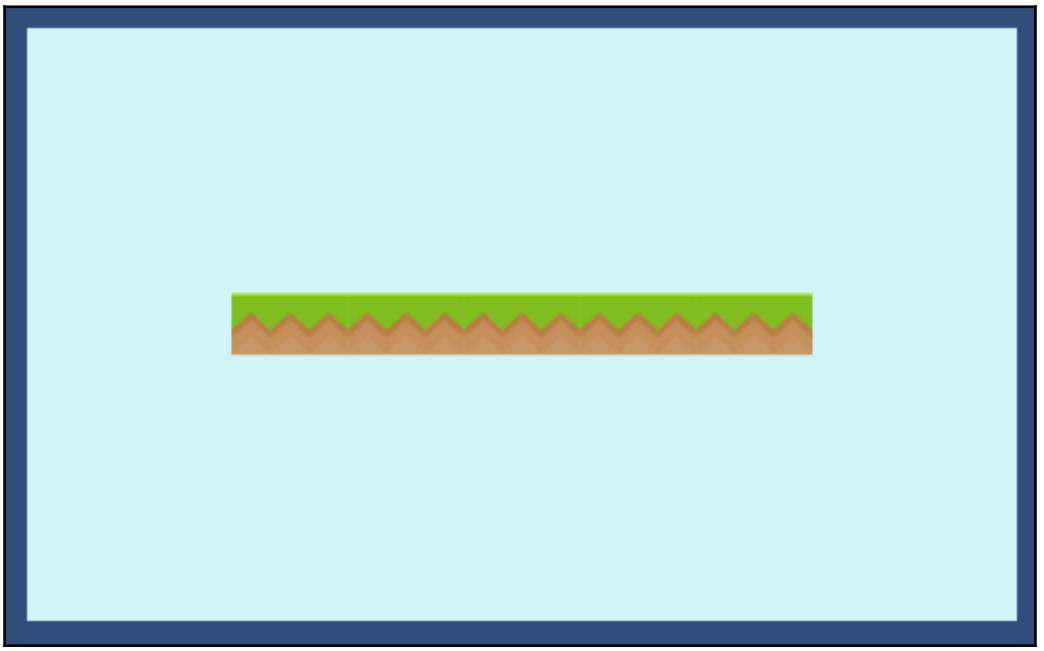
5. Add the component `PlatformEffector2D` to the object `Floor_1`. To easily find it, you can use the search tool or navigate to **Component** | **Physics 2D**.
6. Check the **Used By Effector** variable in the collider to `True`.

7. Change the `PlatformEffector2D` component to match the following image:

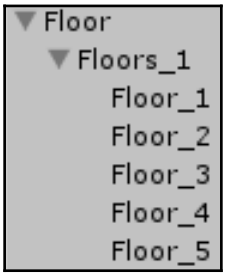


8. In the Box Collider 2D component of `Floor_1`, use **Slippery** as the physics material. It can be found in the **Standard Assets**.

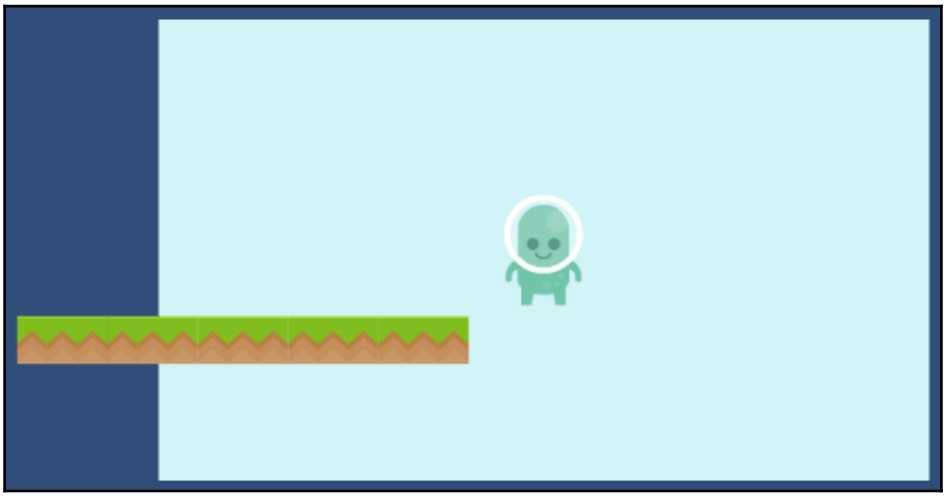
9. Duplicate the child floor object until you achieve the following appearance (leave a distance of `1.5` between each block on the *X* axis):



10. This is the appearance in the **Scene** view.

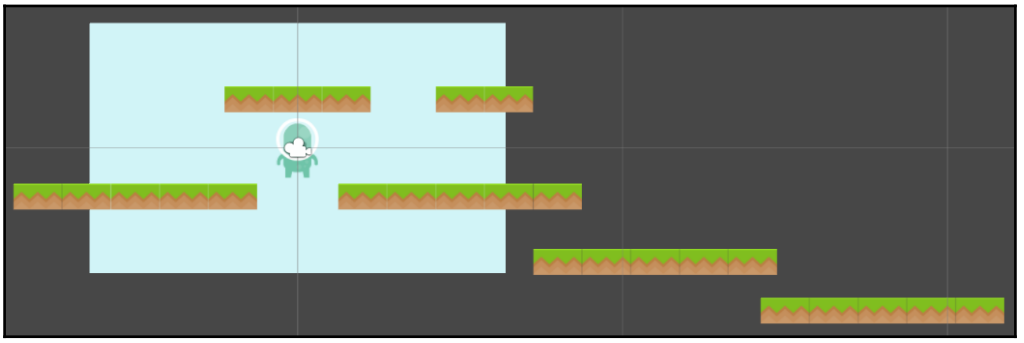


11. Change the position of the object `Floor_1` to `(-5,-1.5,0)`.



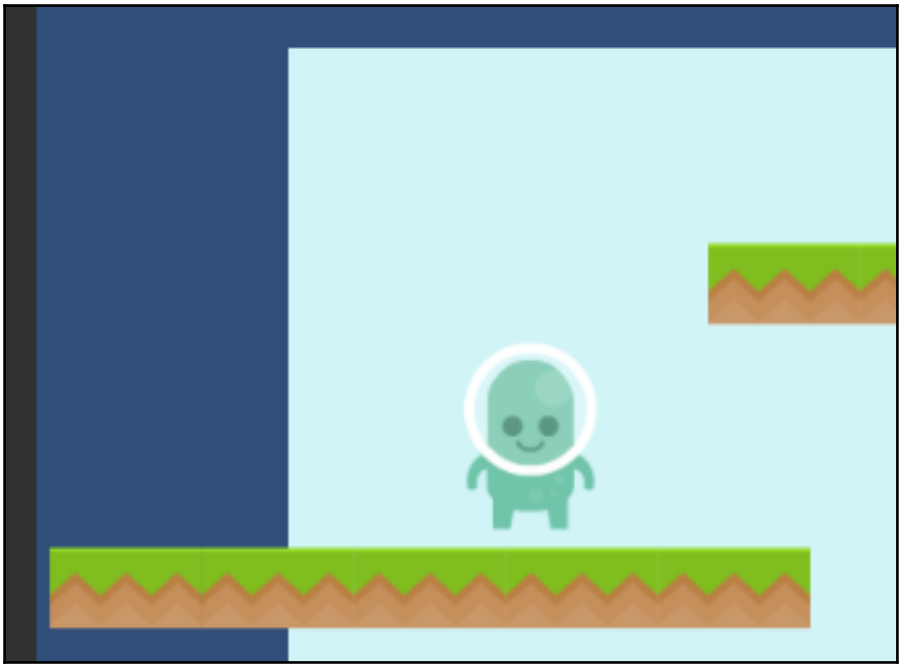
12. Keep duplicating the floor objects until you achieve the following result:

To quickly duplicate game objects, you can use `prefabs`. They are real time savers.



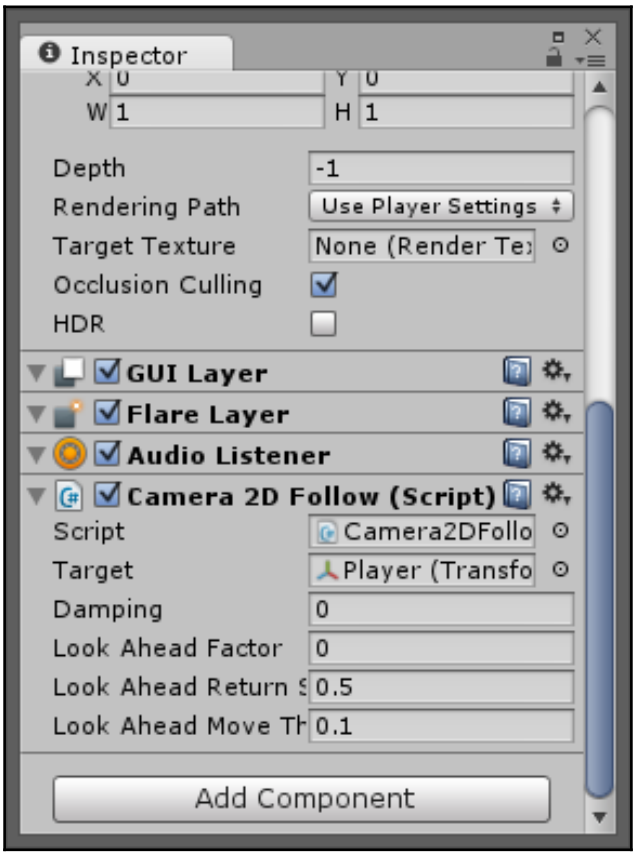
Final result achieved by duplicating (or using Prefabs). How it appears in the Project panel.

13. Change the position of the `player` object to the following `(-4,0,0)` and make sure that its tag is set to `Player`:

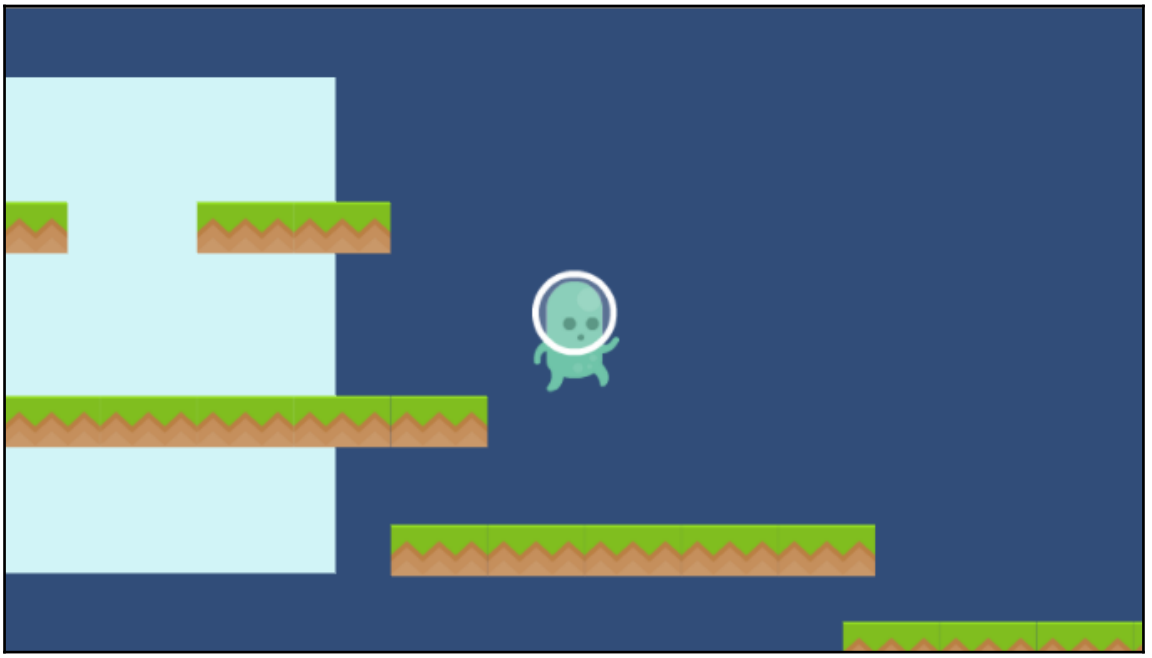


14. Select the `Camera` object and add the `Camera2DFollow` component. To add the component, use the search tool or navigate to **Component** | **Scripts**.

15. Inside the `Camera2DFollow` component, set the target to the player object, then set both **Damping** and **Look Ahead Factor** to 0:



Now, let's try to play this level and see what we have so far!

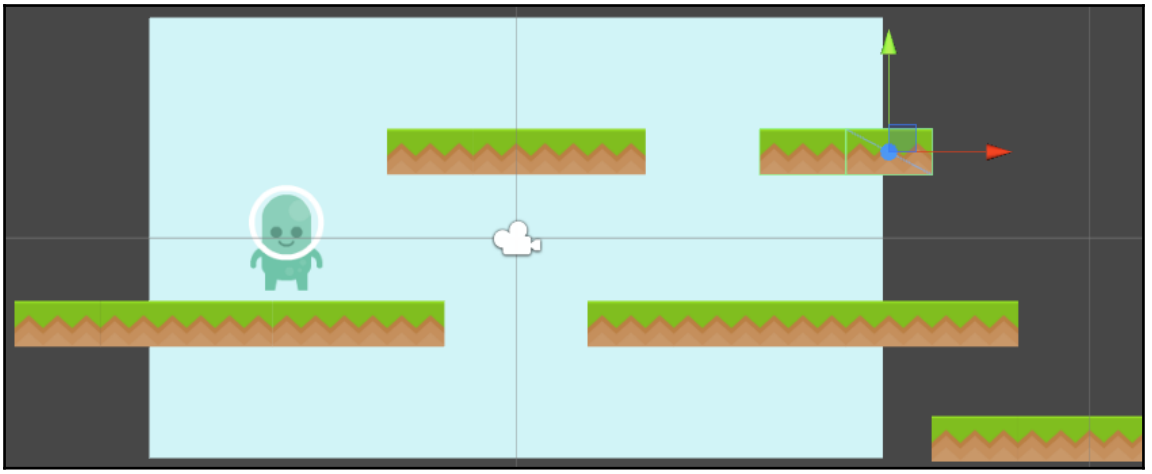


As you can see, we now have a nice Platformer level. But, before going further, we need to understand the changes that we have made to our scene. They are explained as follows:

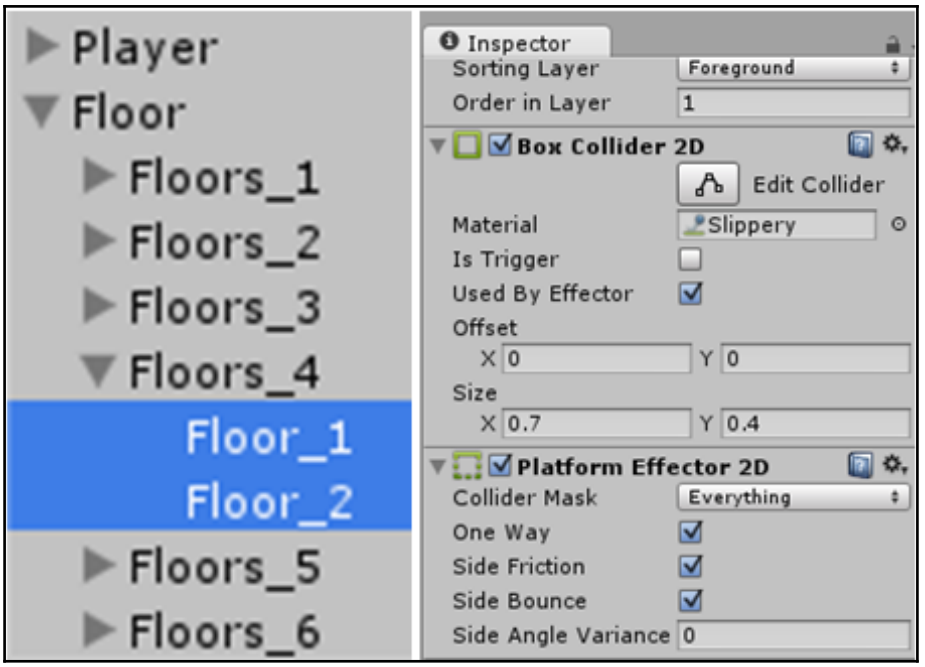
- `PlatformEffector2D`: This built-in component, when added to an object with a collider, can apply various platform behaviors such as the `One Way Collision` which enables the player to go through a platform when jumping upwards but doesn't let him pass through when falling down.
- `Camera2DFollow`: This script controls the camera to track the player when he moves. It can be found in the `2D` package in the `StandardAssets` folder.

Now the level is okay, but we can still add a few touches to spice things up! For instance, you can try the following:

1. Under the `Floors_4` object, change the `OneWay` variable in its children `PlatformEffector2D` component to `True`:

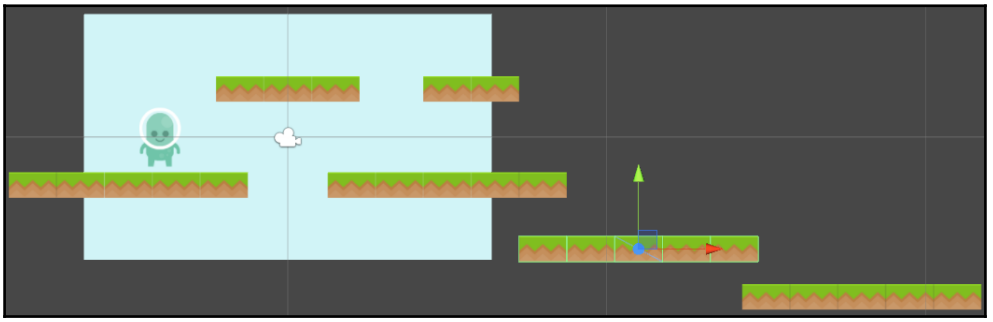


Here is where it is located in the Scene View.

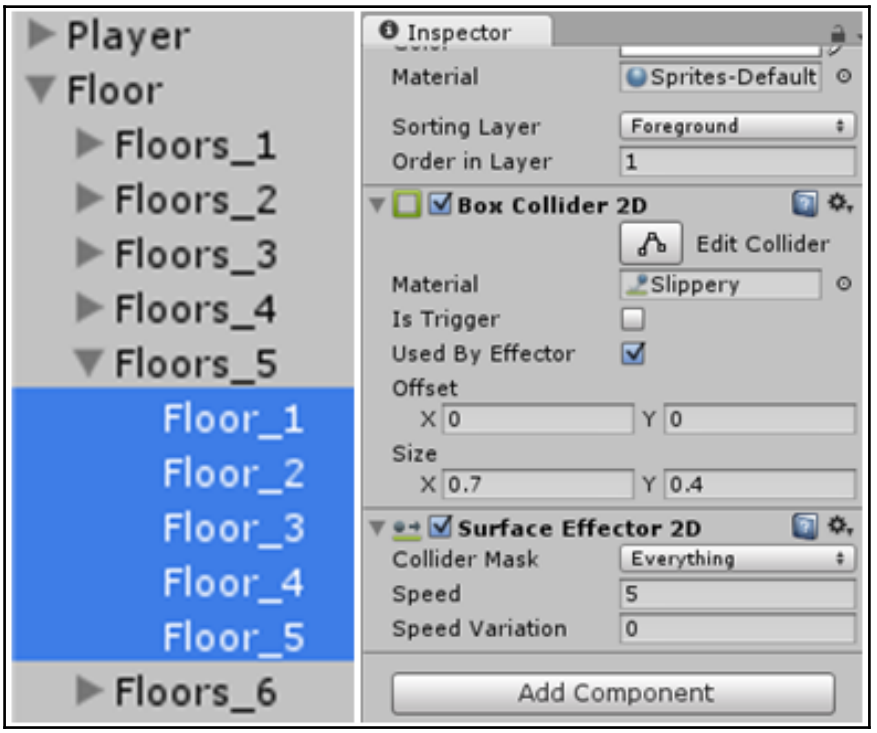


And here is how the components appear in the Inspector.

2. Under the `Floors_4` object, remove the component`PlatformEffector2D` from its children.
3. Under the `Floors_4` object, add the component `SurfaceEffector2D` to its children and set the `Speed` variable to `5`. To add this component, use the search tool or navigate to **Component** | **Physics 2D**. Here it is in the **Scene** view:

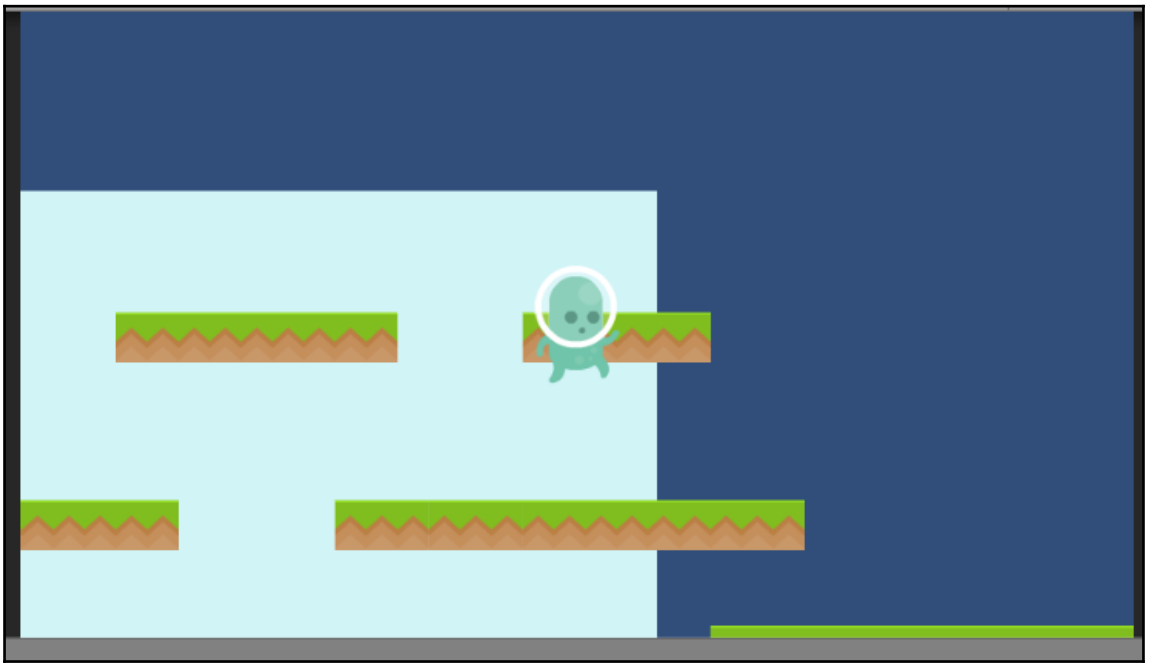


Here it is in the **Scene** view:



The `SurfaceEffector2D` component applies a force along the surface of the collider, which in turn moves the physical bodies on top of it accordingly.

Let's see what we have accomplished so far in this chapter! So, make sure to save the scene first and then hit play:



That's it! Now that we have changed the player movement to respond to the physical forces in the scene, which added more realism to our game. You can play around and build different scenarios so you can be confident with the concepts explained in this chapter.

Summary

In this chapter, we covered 2D Physics Engine and the use of 2D colliders and the platform controller. After that, we applied what we learned to our game. In the next chapter, get ready, because we are going to finish this game!

4
Level Design

Great games are often games that contain beautiful environments. However, creating a nice looking level is not an easy task to accomplish, even in 2D.

In this chapter, we will learn how to create a proper level. The following is what we will cover:

- Level design with tiled images
- Approaching UI
- Implementing game logic
- Scripting and placing enemies in the level

Tiled for 2D level design

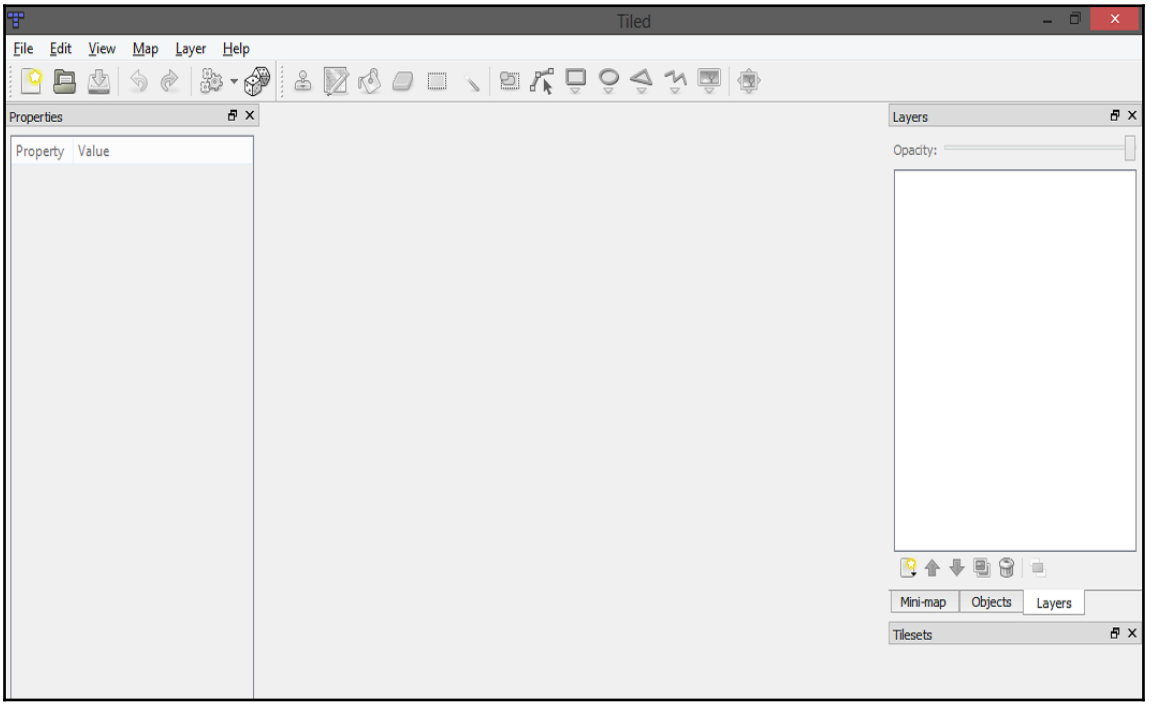
In this chapter, we are going to finish the game that we started in the previous chapters. To begin, let's start by designing an even cooler level!

Before we start creating our level, it's good practice to design our level on paper, even if it's some sketches of different ideas.

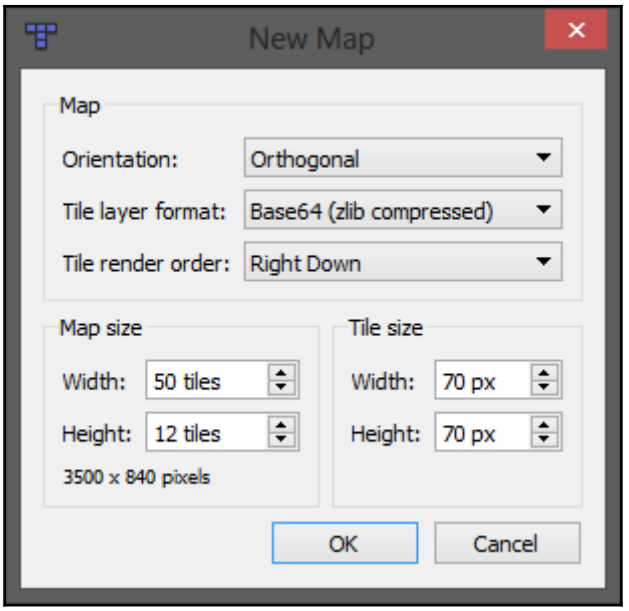
We will be using a third-party tool named **Tiled** for this purpose. Tiled is a free 2D map editor that will save you a lot of time working on your levels. It is a tool that makes it much easier to create a 2D level, instead of doing it within Unity by duplicating game objects. Here is the download link for it: `http://www.mapeditor.org/download.html`.

Lastly, in order to use what you will create in Tiled, you also need to download another program called **Tiled2Unity**. It is a utility that allows us to import levels that have been created in Tiled, as prefabs into Unity. Here is the link to download Tiled2Unity: `http://www.seanba.com/tiled2unity`.

After installing both programs, we can start by opening Tiled. This is what it looks like when it is opened:

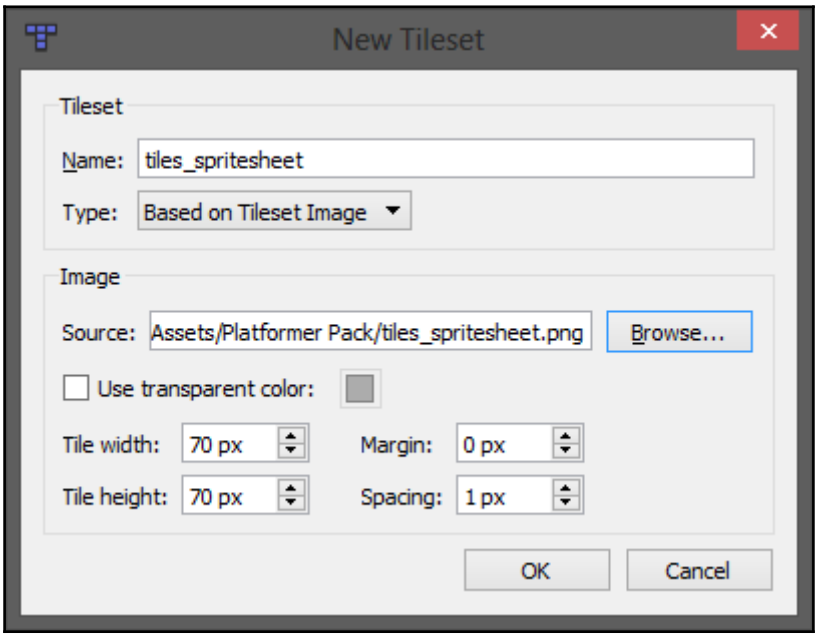


Click on the **New** button in the top-left corner. A box will appear on screen, where we can set the new map properties, as shown in the following image:

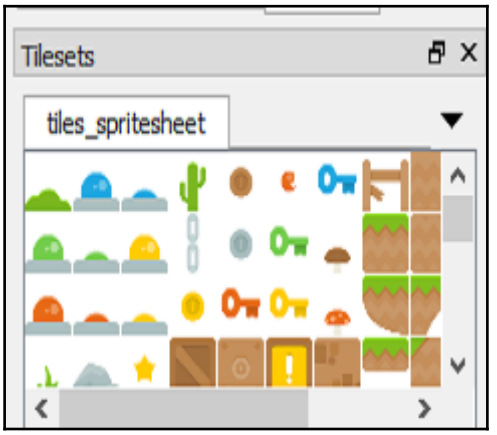


From the **Tilesets** panel in the bottom-right corner, click on **New Tileset**:

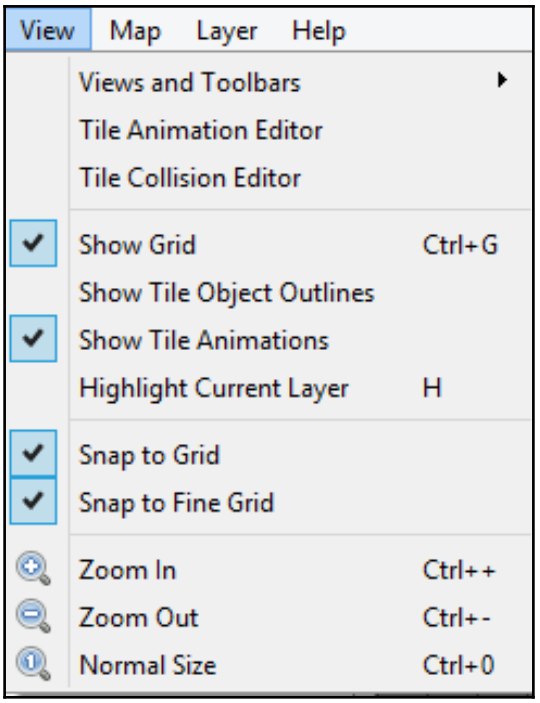
Link the source to the Tiles sprite sheet we have in our project `PlatformerPack | tiles_spritesheet.png` and configure the rest, as shown in the following image:



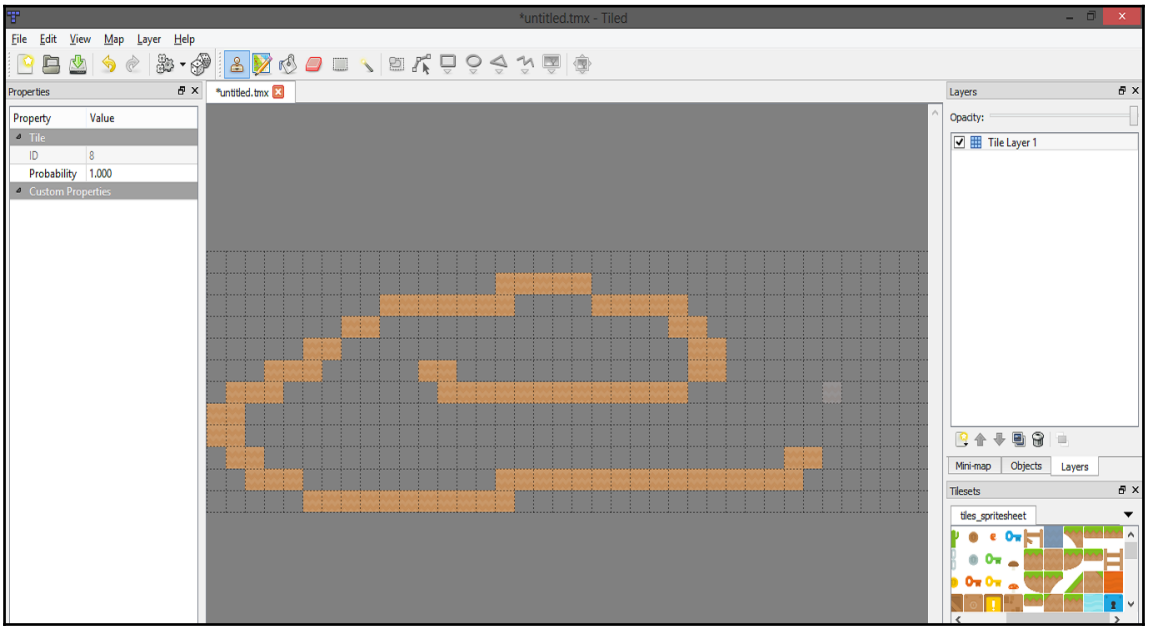
A new map will be created with our tiles being available for selection in the **Tilesets** panel:



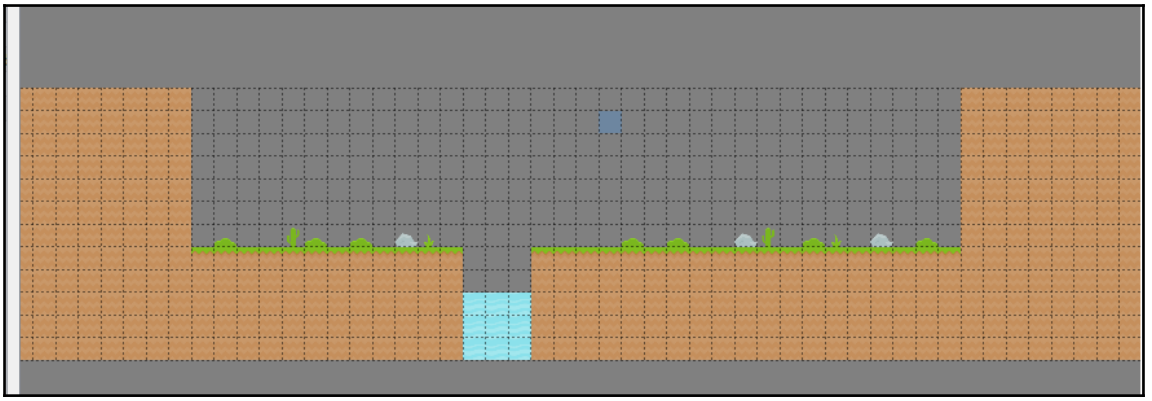
Before we start drawing, a good rule of thumb is to select **Snap to Grid** and **Snap to Fine Grid** in the **View** menu:



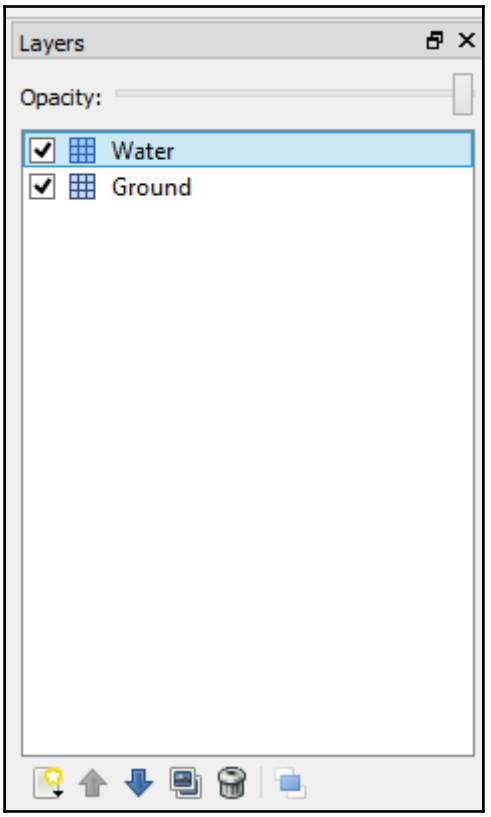
Click on any of the tiles and left-click and drag your mouse inside the map. Here, you will see that the tile is being placed in each grid you pass over:



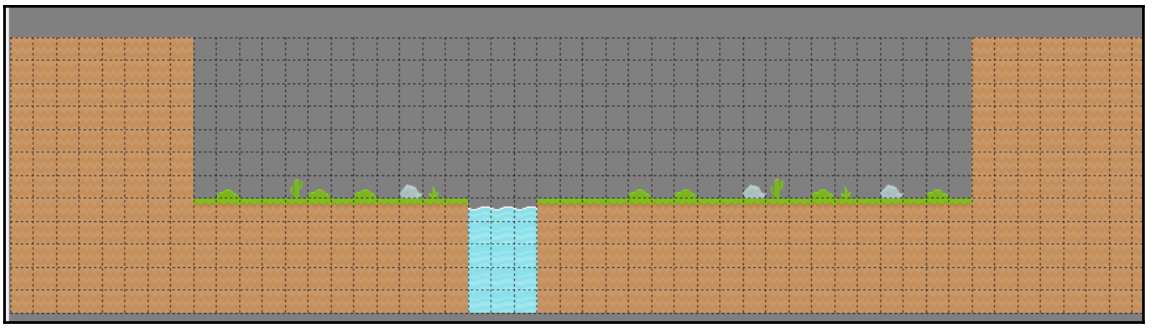
Now, let's try to make something a little more pleasing to the eye than the preceding image. An example can be using a mixture of the tiles that we've got:



While painting your way through the level, you may have noticed the **Layers** panel and that the current layer we have been working on is called `TileLayer1`. Let's call it `Ground` and then add another layer called `Water`:

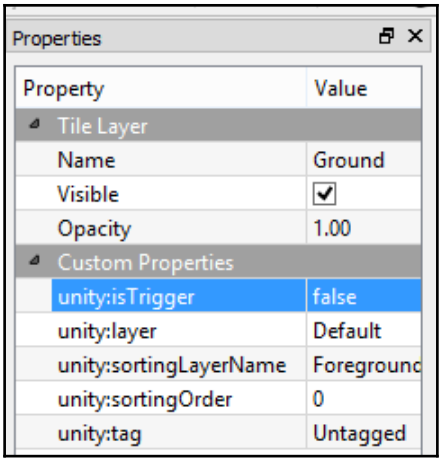


Layers allow you to improve the organization of your level. All the work we've done so far has been put on the `Ground` layer, which is good. But now we need to add some water. Therefore, select the `Water` layer and add some water tiles. Make sure to leave the top three tiles empty; we will use them next. Now, to finish the water area of our map off, add the wave tiles to the top in the empty space above the original water area, as shown in the following image:



Each layer has its own properties, and the cool thing is that when we import the map using Tiled2Unity, it will apply each layer's properties to the soon to be in-game objects. This can save a lot of time later on.

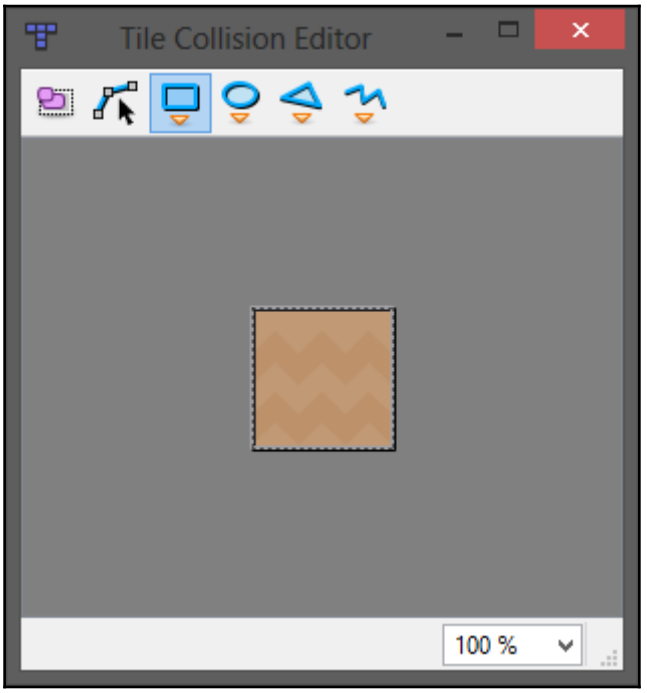
Select the `Ground` layer and navigate to the **Properties** panel on the left-hand side of the screen and add the following **Custom Properties**:



Now, let's repeat these steps again for the the `Water` layer, but change the **unity:isTrigger** property to `true` instead of `false`. In fact, we will use this trigger later in the game to check if the player falls into the water gap.

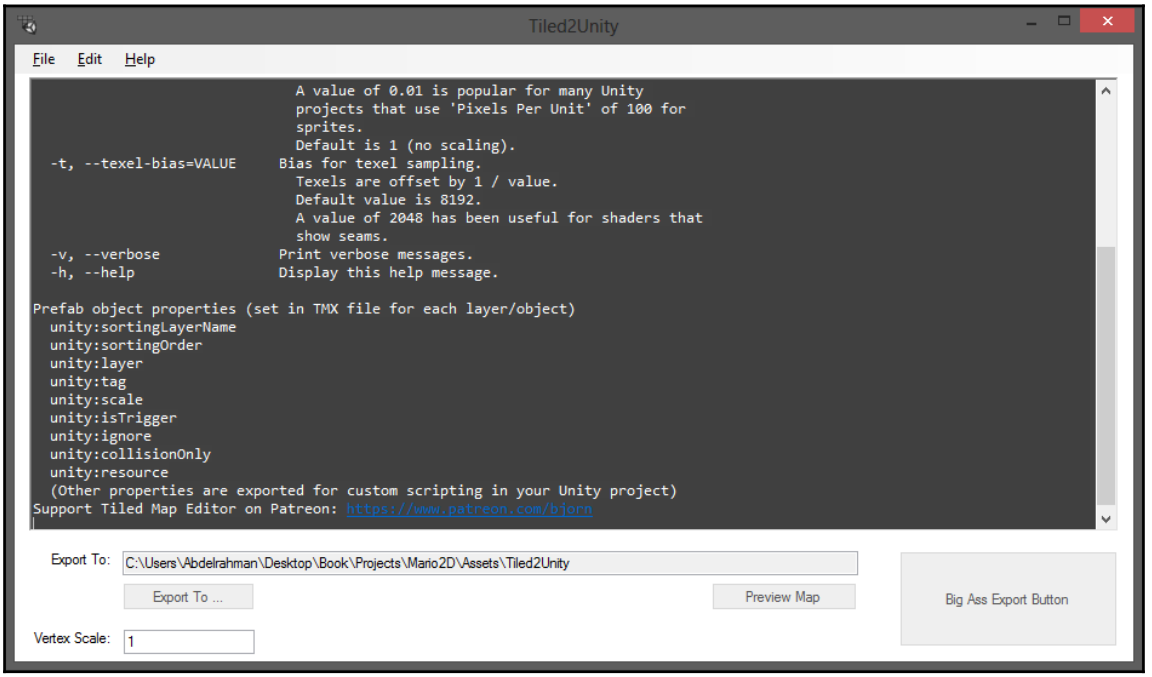
The next step is to add some colliders to our tiles in the map. To add a collider to a tile, you need to select that tile from the **Tilesets** panel and navigate to **View** | **Tile Collision Editor**.

After opening the **Tile Collision Editor**, use the rectangle tool to fully enclose our tile, as shown in the following image:



Repeat the previous steps for all of the tiles we have added into the map, except for any decorative tiles such as grass or rocks. When you are done, make sure to save the file. In our case, we will name it `Platformer`. Before going any further, make sure that our Unity project is opened!

In order to import what we have done so far in Tiled to Unity, we will need to use the `Tiled2Unity` tool that we installed earlier. After you have opened the program, you will see a window like the one in the following image:

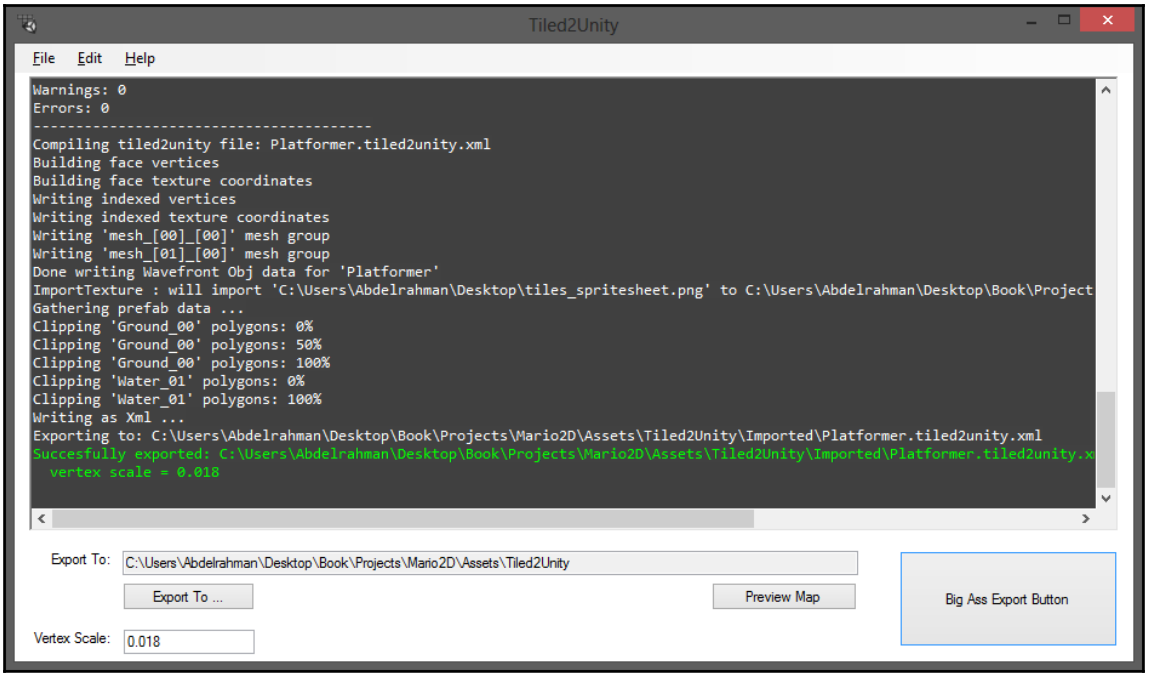


As you can see, this tool is pretty straight-forward. Click on **Help** | **Import Unity Package To Project** and inside Unity, click on **Import**.

What this will do is import the plugin into Unity so that we will be able to use the files that come next in our scene. Next, go to **File** | **Open Tiled File** and select the saved file `Platformer.tmx`.

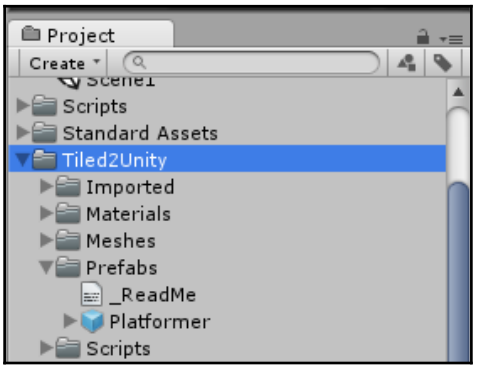
Next, click on the **ExportTo** button and make sure it is set to the `Tiled2Unity.export` file inside the folder `Assets/Tiled2Unity`. Finally, set the **Vertex Scale** to `0.018` and then click on **Export**.

The files will be imported to Unity and the program will display the stats for the conversion, as seen in the following image:

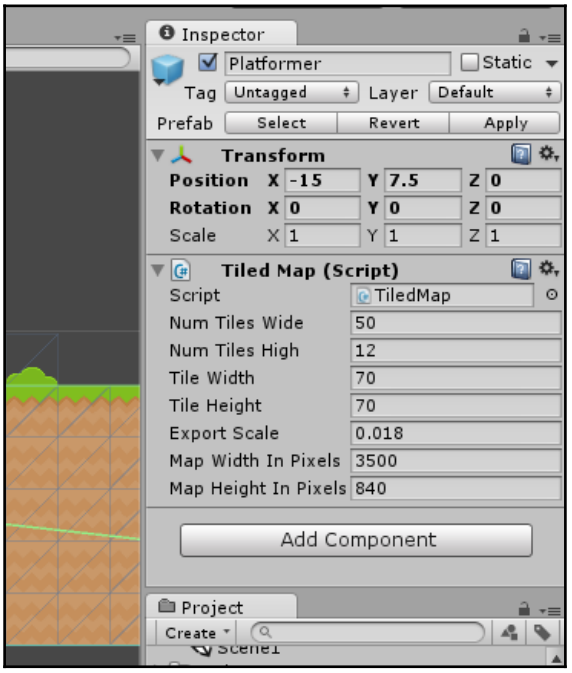


You can close **Tiled** and **Tiled2Unity,** as we do not need them, and head back to Unity.

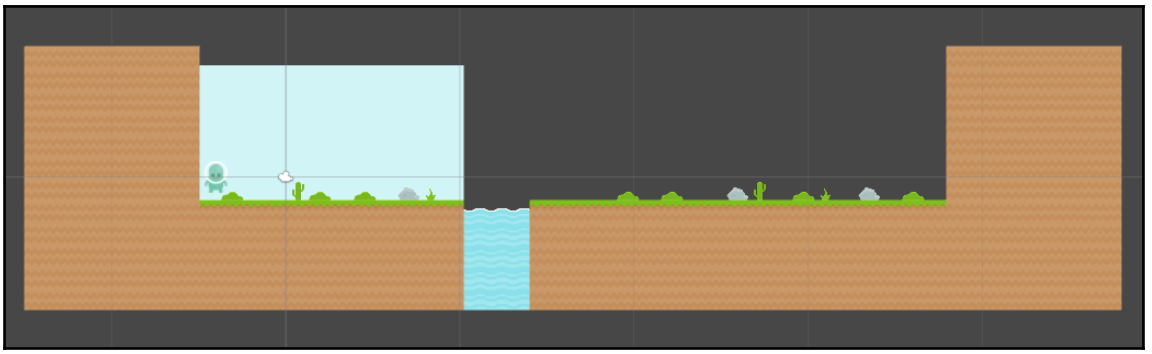
A new folder has been created inside our project called `Tiled2Unity`. Inside it, our map is set as a ready-to-use prefab in the `Prefabs` folder under the same name we saved it as. In this case, `Platformer`:



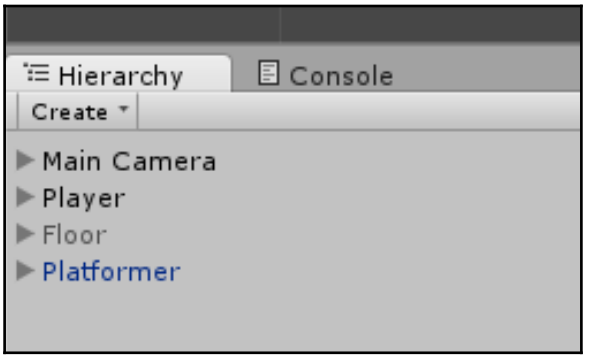
Disable the `Floor` object then drag and drop the `Platformer` prefab into the scene. Modify its transform position, as shown in the following image:



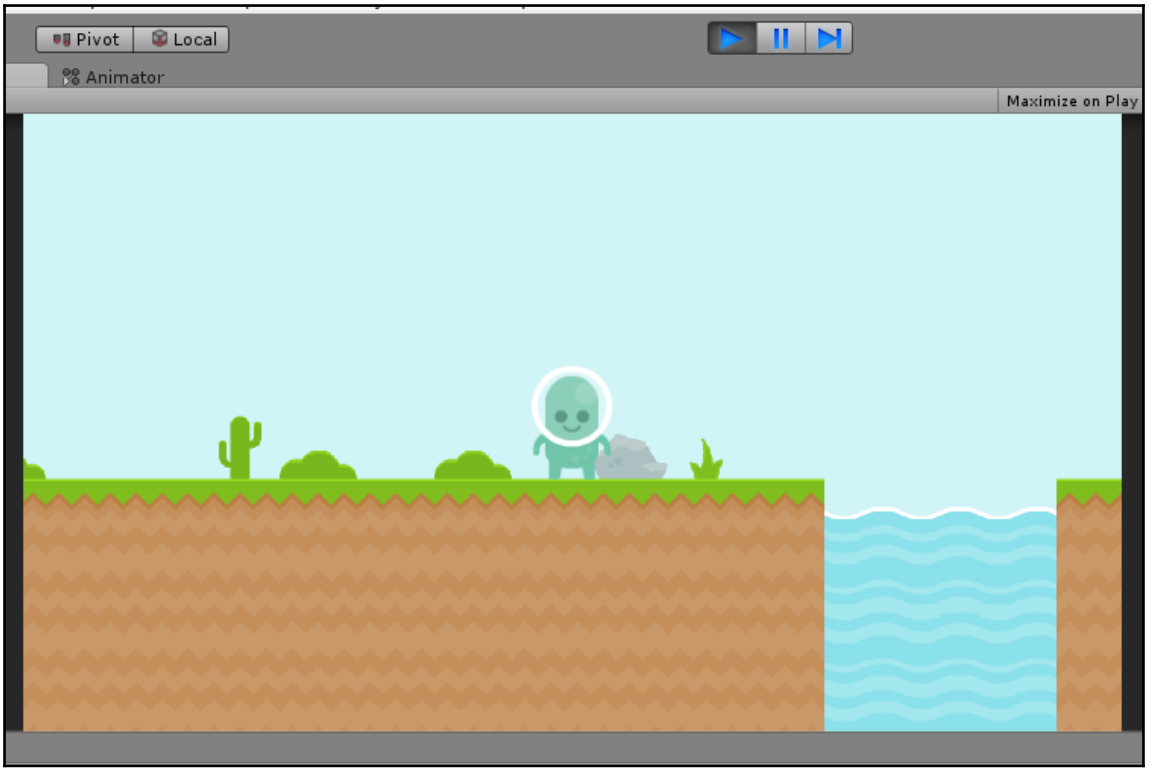
Next, modify the scale of the `Background` object to `(5,3,1)` and then parent it to the `Camera` object, so that the background will follow our character as well. Now the scene should look as follows:



Here, we can have an idea of how to organize everything in the **Project** panel:

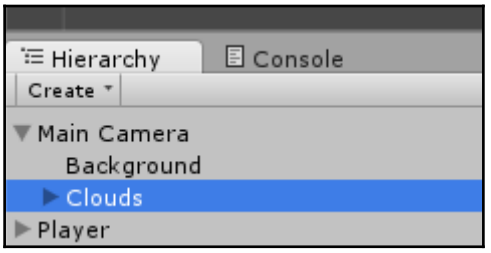


Press the play button to see what we have accomplished so far!

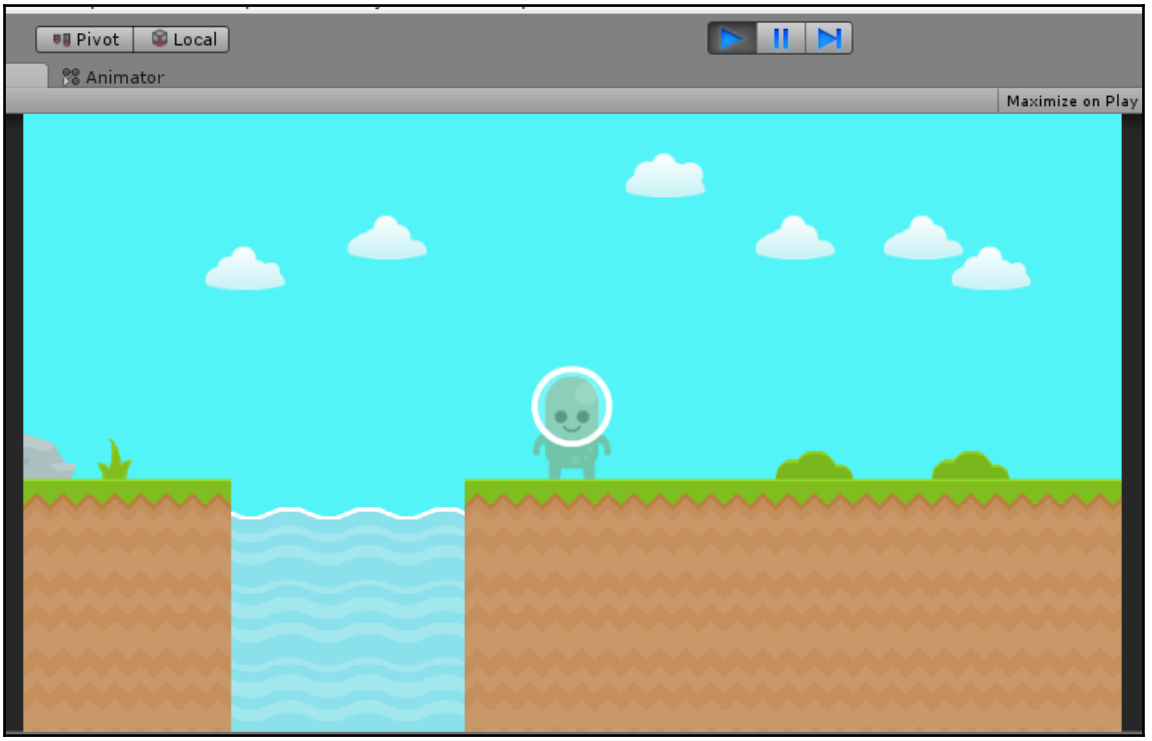


Our scene looks much better, but we can still improve it.

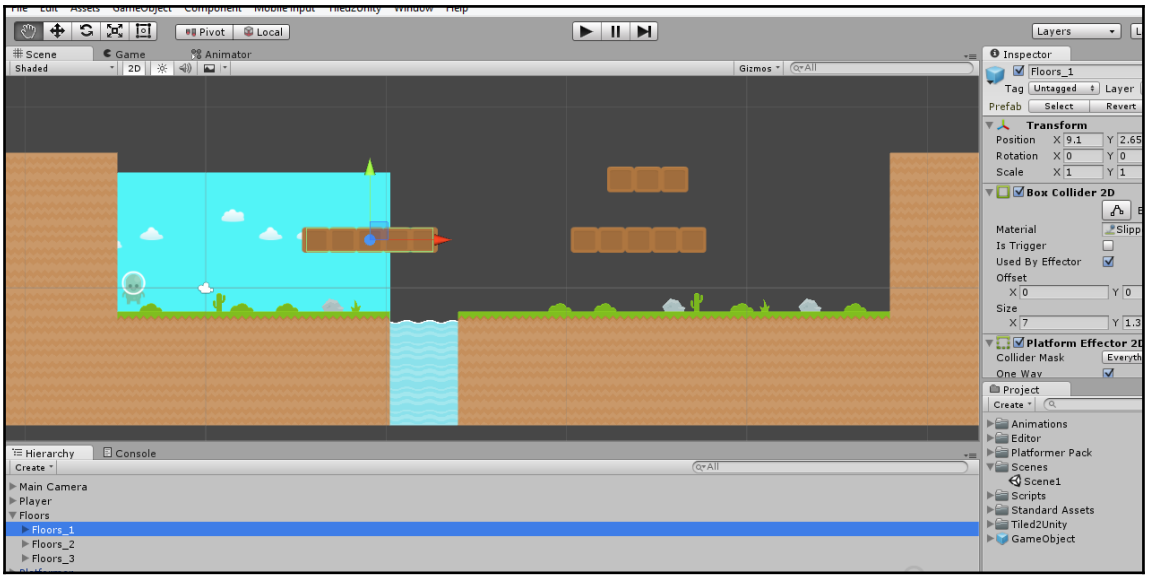
Let's start by adding some clouds to the **Background** object under **Main Camera**:



This is how the scene should look afterwards:



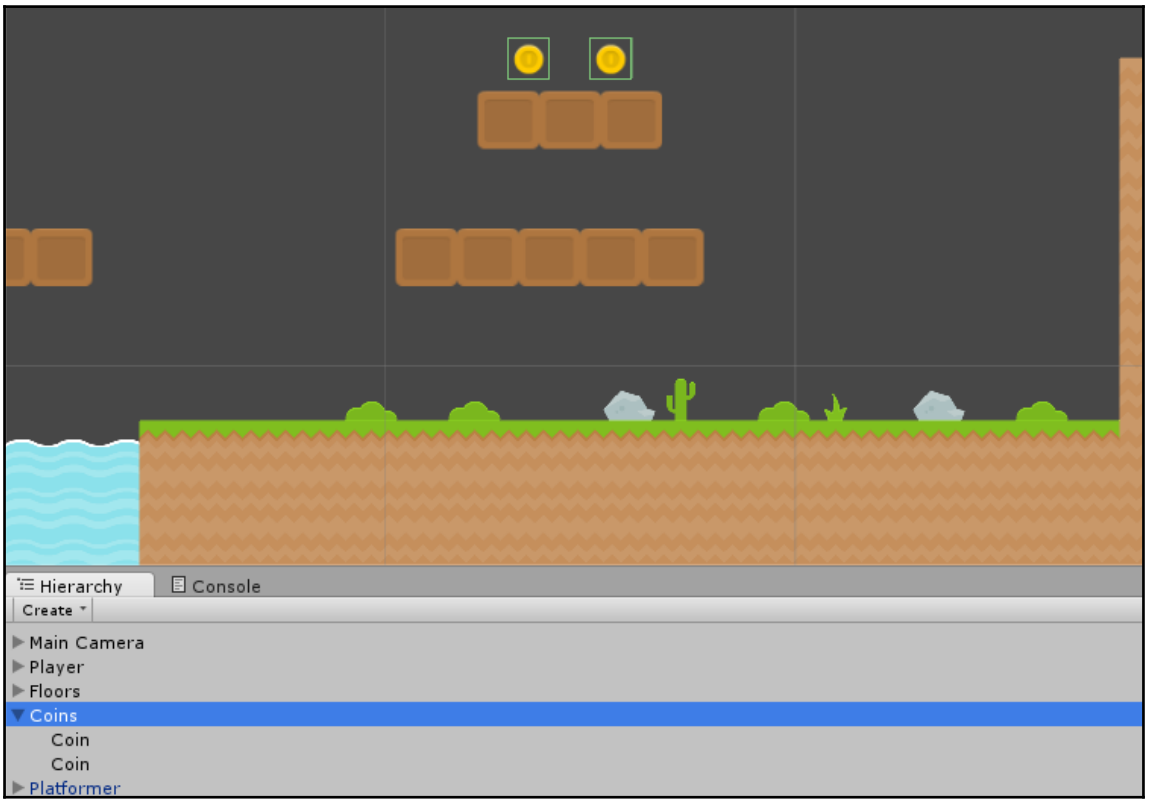
Remove the old `Floor_#` objects and add some new floor tiles. Then, assign colliders to them and platform effectors. Finally, parent them to the `Floors` game object, as shown in the following image:



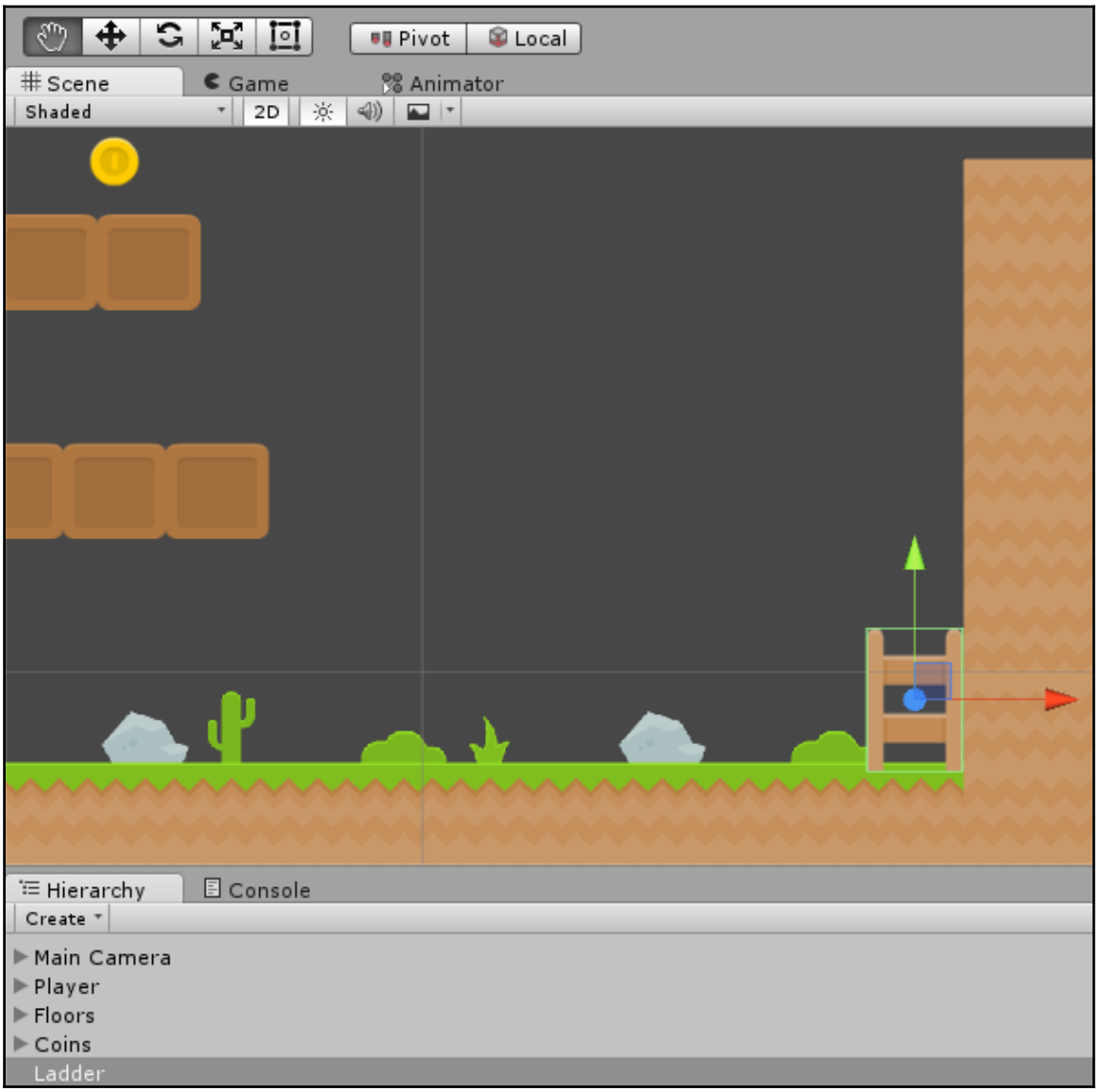
You will notice that we assigned the collider and the effector to each group of floor tiles instead of the tiles directly. This is something that I recommend, as it will be easier to organize and edit along the way, especially when your scene starts getting bigger. Also, make sure that the **One Way Boolean** inside the **Platform Effectors** is `true`, as it will be better for our intended gameplay.

www.allitebooks.com

Let's add some coins to the scene just above the top floor so that the player can increase his score by collecting each one. We will configure their behavior later, but for now, don't forget to add a collider to each coin and set the **IsTrigger** property to `true`:



Now, let's add an exit area in our scene so that the player can finish the level and end the game. Grab a ladder sprite and add it at the end of the scene and then attach a collider to it with the **IsTrigger** option turned on:



We should group our scene items now so that everything will be kept neat and clean for future changes. Create a new game object named `SceneObjects`. We have finally finished designing our scene! The next step is to implement the User Interface.

Approaching UI

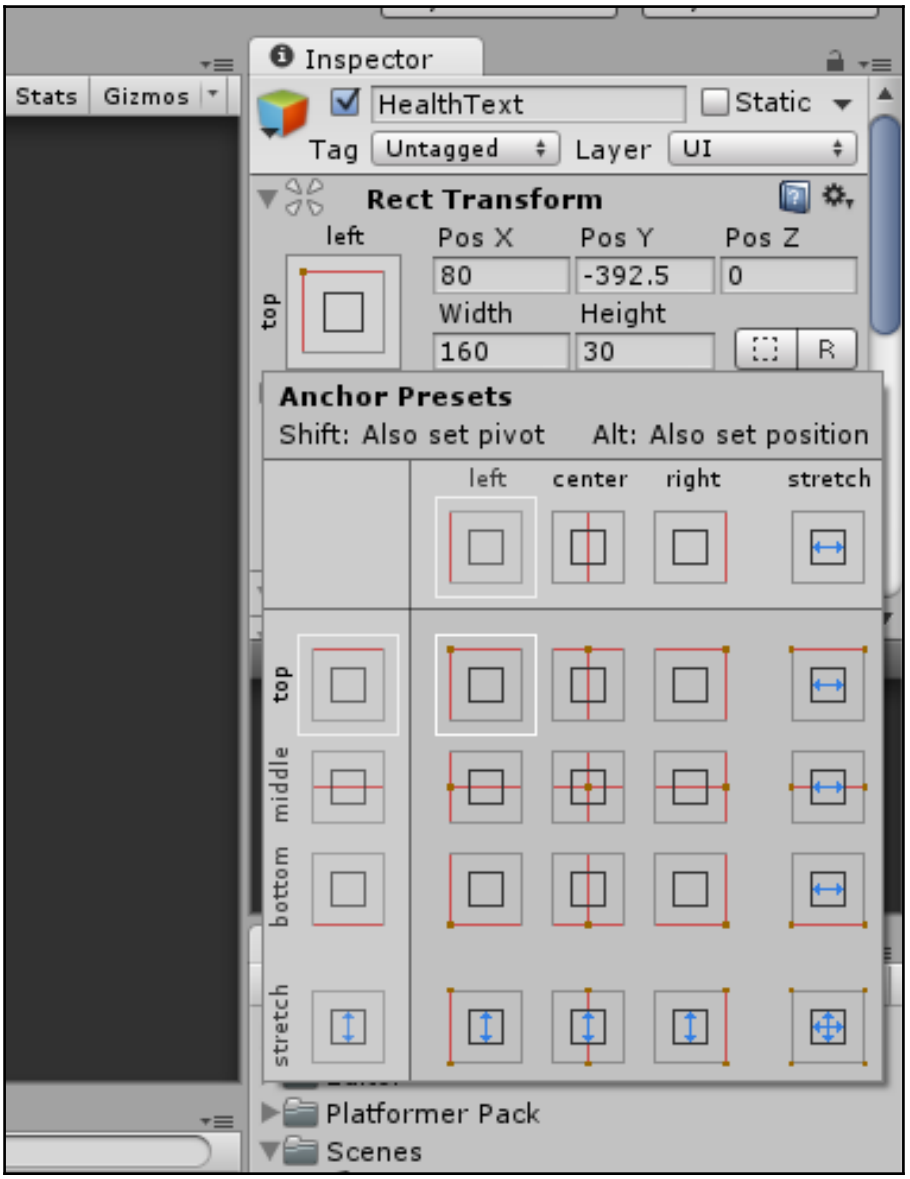
We will learn about some more advanced features of **User Interfaces** (**UI**) in `Chapter 8`, *User Interface for the Tower Defense Game*. However, I recommend that you read a specific book about UI, if you want to master it. For instance, *Unity UI Cookbook*, *Packt Publishing*.

To make our level a little more enjoyable, we should add some UI elements to indicate the player's health and score. So let's do that!

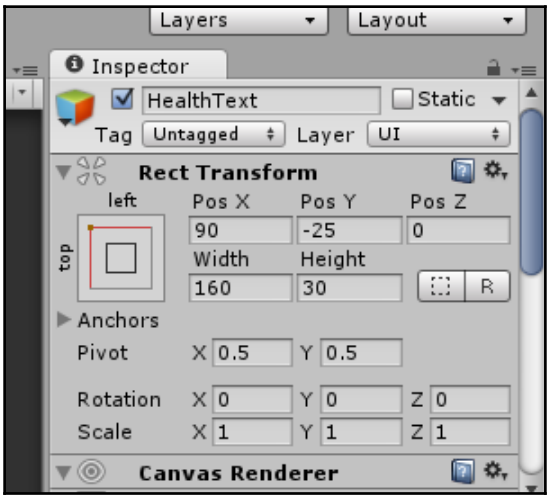
In order to add some UI elements, we need a canvas. Usually, every time we create a UI element for the first time in the scene, a canvas is created as well. However, you can also create one by right-clicking in the **Hierarchy** panel and then clicking on **UI** | **Canvas**. A new object is added to our scene, with the name `Canvas`.

A canvas is an area that all UI elements must be children of. Let's start by adding some UI text under it. We can add it by right-clicking on the **Hierarchy** panel and then **UI** | **Text**. As a result, a new object named `Text` under `Canvas` has been created.

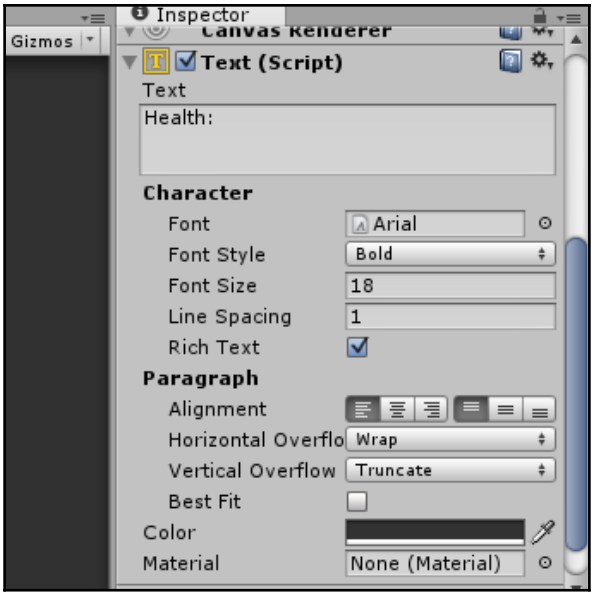
First, let's rename our `Text` object to `HealthText` and then adjust its position so that it is positioned at the top-left of the game view. We can achieve this by choosing the top-left anchor:



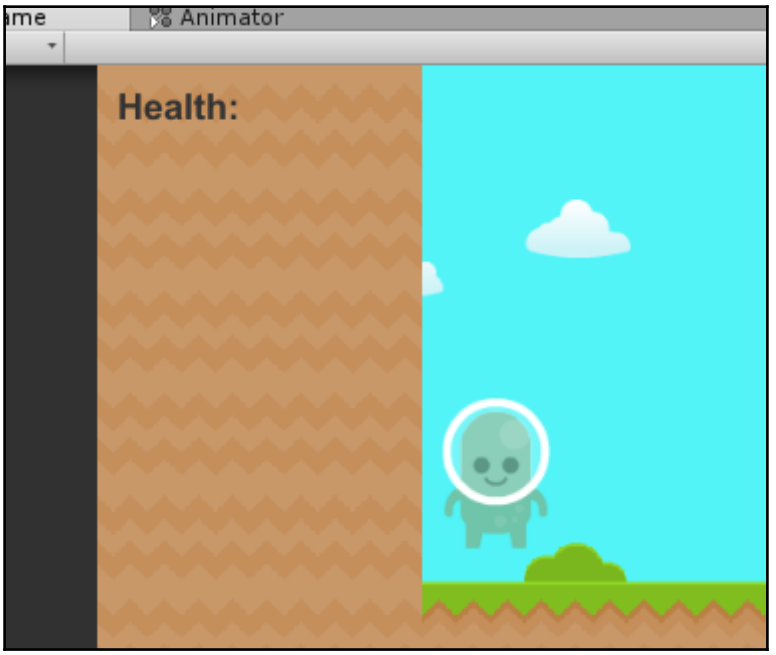
Next, adjust the **Rect Transform** accordingly, as shown in the following image:



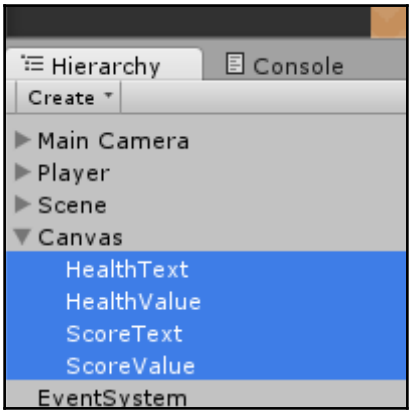
Next, change the `Text` component attached to the `HealthText` object to match the properties, as shown in the following image:



As a result, we will have this in our **Game** view:



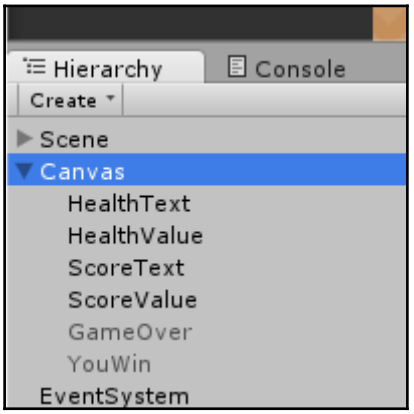
We need to add three more text objects, which are all parented to the canvas, as shown in the following image:



The **HealthText** and **ScoreText** texts should be **Health:** and **Score:** respectively; as for **HealthValue** and **ScoreValue** texts, they should be `2` and `0`, respectively. Note that the last two texts will be referenced later, in a script, so that they can be changed in-game to show the player stats.

As a result, our **Game** view should look as follows:

Let's add two more texts to our canvas, named `GameOver` and `YouWin`, which will be shown to the player when one of the two events occur. Don't forget to disable both objects so that they don't appear while we are playing the level! You can disable them by unchecking the box, located next to their name in the **Inspector**. We will enable them again, in-game, when they are needed:



For the **Game Over** screen, we could create something like this:

Whereas, for the **Winning** screen, we could have the following:

Game handler

Now that we have all the elements in our game, we need a script that makes our character intractable with the other elements in the level, and that takes care of the level logic.

To do this, create a new C# script and name it `GameHandler` under the folder `Scripts`. Then, attach the script to our `Player` object. Double-click on the script in order to open it.

First, we need to define a lot of variables. Let's start with two to keep track of the player's score and health:

```
public float health = 2;
public float score = 0;
```

Now we need a variable to check if the game is over:

```
public bool gameover = false;
```

Finally, we need the reference to our UI elements:

```
public UnityEngine.UI.Text healthUI;
public UnityEngine.UI.Text ScoreUI;
public GameObject gameOverUI;
public GameObject youWinUI;
```

The next step is to implement the logic behind the collision of the character with an object. This can be done by using a function called `OnTriggerEnter2D()`. Inside this function, we can check which kind of object the player has crossed. If it is a coin, then the score is updated and the coin is destroyed. If it is water, then the player dies. If it is the end of the level, we need to show the winning screen:

```
void OnTriggerEnter2D(Collider2D c) {
    if (c.name == "Coin") {
        AddScore();
        Destroy(c.gameObject);
    }
    else if (c.tag == "Water") {
        health = 0;
        healthUI.text = health.ToString();
        gameOverUI.SetActive(true);
        StopGame();
    }
    else if (c.tag == "Ending") {
        youWinUI.SetActive(true);
        StopGame();
    }
}
```

Now we need a function to subtract the health of the player and update the UI as well:

```
public void SubtractHealth() {
    health -= 1;
    healthUI.text = health.ToString();
    if (health == 0) {
        gameOverUI.SetActive(true);
        StopGame();
    }
}
```

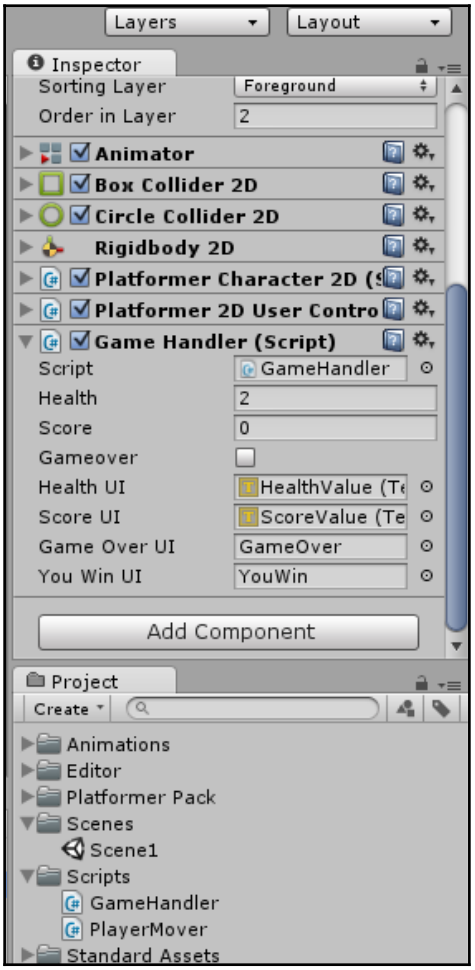
Another function is needed to increase the score of the player:

```
public void AddScore() {
    score += 10;
    ScoreUI.text = score.ToString();
}
```

Finally, the following function is for when the game is over:

```
public void StopGame() {
    gameover = true;
    gameObject.SetActive(false);
}
```

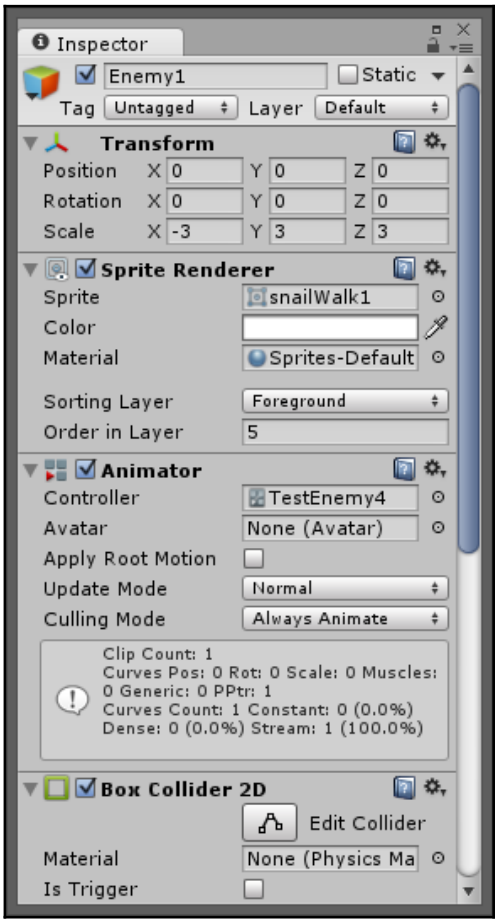
Save the changes and head back to the scene. Make sure that the object `Collision` in **Scene | Platformer | Water_01** is tagged as `Water` and the `ladder` object is tagged as `Eding`. Finally, assign all the references in `GameHandler` with the respective objects, as shown in the following image:

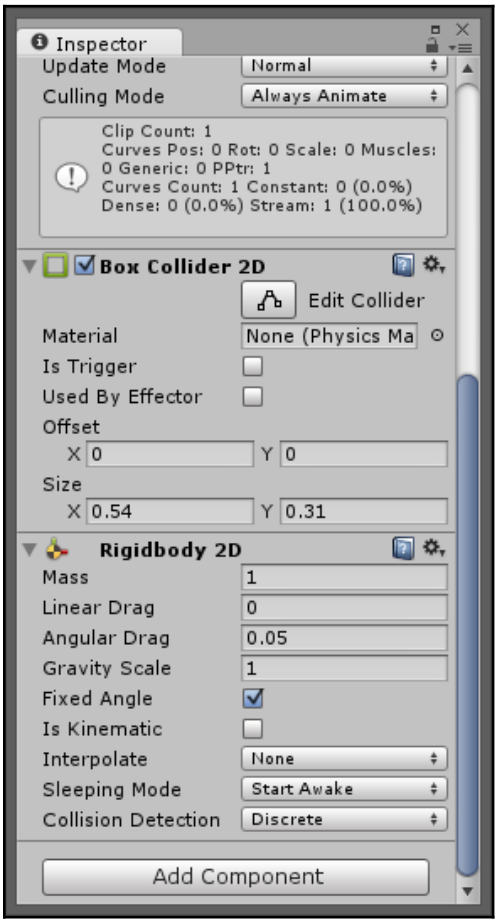


Adding enemies

To make our level more exciting, we need to add some enemies to the level. We can use the sprites under **Platformer Pack** | **Enemies** and start by creating the snail enemy.

Just like how we animated our player, we will do almost the same for the enemies; drag and drop both `snailWalk1` and `snailWalk2` into the scene to create the animation. Then, rename the object to `Enemy1`, scale the object to `(-3, 3, 3)`, so that it fits the dimensions of our game level, and then place it on the first platform of the level. Finally, add a box collider that fully encloses the sprite and a rigid body with its **FixedAngle** variable set to `true`. As a result, the **Inspector** should look like the following:





Before going forward with the enemies and their code, we first need to create some obstacles to place in their way. In fact, our enemies will change direction every time they collide with an obstacle.

We can achieve this by dragging `boxAlt` from the **Project** panel under **Platformer Pack** | **Tiles** inside the scene. Then, change the tag to `Obstacle` and add a collider to the object with the **IsTrigger** variable set to `true`. Also, name the object `ObstacleUp_1` and duplicate it. The duplicated object should be `ObstacleUp_2`.

To quickly reuse obstacles, you can create and use prefabs too.

Finally, parent them to **Scene** | **Floors** and place them along the path of our snail enemy, as shown in the following image:

Now, we need to script the behavior of the enemy. So, let's create a new C# script and name it `EnemyScript` inside the folder `Scripts`. Then, attach the script to the `snail` object. Double-click on the script to open it.

We first need to add some variables to store the speed of the enemy, its velocity vector, and scale:

```
public float speed = 1;
Vector2 curVelocity;
Vector3 curScale;
```

In the `Start()` function, we need to initialize its velocity in the rigid body:

```
void Start () {
  //Set initial direction and speed
  GetComponent<Rigidbody2D>().velocity = new Vector2(-1 * speed, 0);
}
```

In the `Update()` function, instead, we need to check if the enemy stops, and make it resume its walking:

```
    void Update() {
        //get the current velocity
        curVelocity = GetComponent<Rigidbody2D>().velocity;
        //Resume walking if the enemy stops
        if (curVelocity.x == 0) {
            transform.position = new Vector2(transform.position.x,
transform.position.y + 0.01f);
            GetComponent<Rigidbody2D>().velocity = new Vector2(curScale.x >
0 ? -1 : 1 * speed, 0);
        }
    }
```

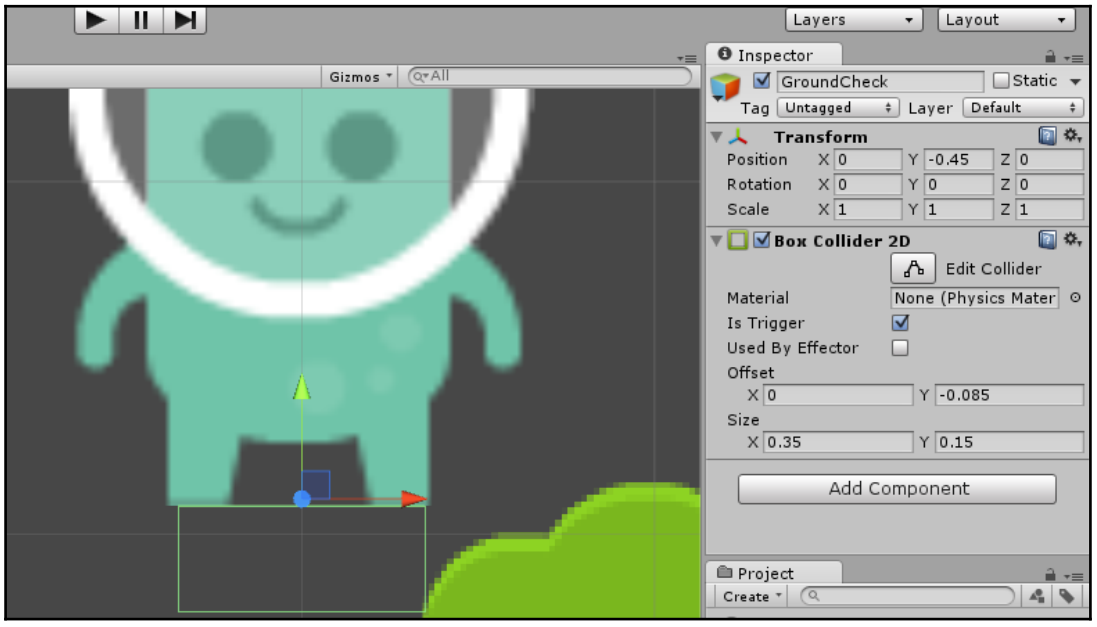
Then, we need to create the `OnTriggerEnter2D()` function to detect if the enemy touched an obstacle, so it needs to change direction, or if the player killed it by jumping on it:

```
    void OnTriggerEnter2D(Collider2D c) {
        if (c.tag == "Obstacle") {
            GetComponent<Rigidbody2D>().velocity = new Vector2(-1 *
curVelocity.x, 0);
            curScale = transform.localScale;
            curScale.x *= -1;
            transform.localScale = curScale;
        }
        else if (c.name == "GroundCheck") {
            print("Killed By Jump!");
            Destroy(gameObject);
        }
    }
```

Similarly, we need to do it if the enemy collides with a collider, and again, we need to check if it is an obstacle or the player. However, in this case, the player's collider is not `GroundCheck`, therefore, in this case, it is the enemy that subtracts a life to the player before they die:

```
    void OnCollisionEnter2D(Collision2D c) {
        if (c.collider.tag == "Obstacle") {
            GetComponent<Rigidbody2D>().velocity = new Vector2(-1 *
curVelocity.x, 0);
            curScale = transform.localScale;
            curScale.x *= -1;
            transform.localScale = curScale;
        }
        else if (c.collider.tag == "Player") {
            c.transform.GetComponent<GameHandler>().SubtractHealth();
            Destroy(gameObject);
        }
    }
```

Save the changes and head back to the scene. In the `GroundCheck` object under `Player`, add a collider and set its **IsTrigger** variable to `true`. Then, center it directly under the player, as shown in the following image:



In this way, we can detect if the player has jumped on the enemy and killed it, or if they touched it and have been hurt.

We can finally test the scene and see what we have accomplished so far:

As you can see, a lot of features have been added to the game, which is great! However, we still need to add another couple of enemies. Since the enemy logic is the same for all of them, we can choose different enemies, instead of always using the snail. Some good options could be the fly, the fish, or the slime.

After we have added three more enemies into the scene, we must add some obstacles at the end of their path in order to keep them in place, just like we did with the snail. Therefore, we can add five obstacles on the ground. However, instead of using them as triggers, we can leave them as normal colliders. As a result, the player will be able to properly interact with them:

Let's see what we accomplished so far in this chapter. Don't forget to save the scene before every test!

Here, in the preceding image, the game is running.

The preceding image is when the player has reached the end and won.

Summary

In this chapter, we covered level design for 2D games and used Tiled to achieve this. Then, we touched on some concepts related to Unity's UI, which we will see in more detail in `Chapter 8`, *User Interface for the Tower Defense Game*. Finally, we implemented the game logic and have scripted enemies, which we added in the level.

In the next chapter, we will head towards creating our second game!

5
Creating Our Own RPG

For our second game, we will work on creating a two-dimensional RPG. We will start working on the game, and begin paving the way for the next chapters.

In this chapter, we will walk through the creation of the game base; the following is what we will go through:

- Role-Playing Games
- Getting the project ready
- Designing the level
- Adding our player

Role-Playing Games

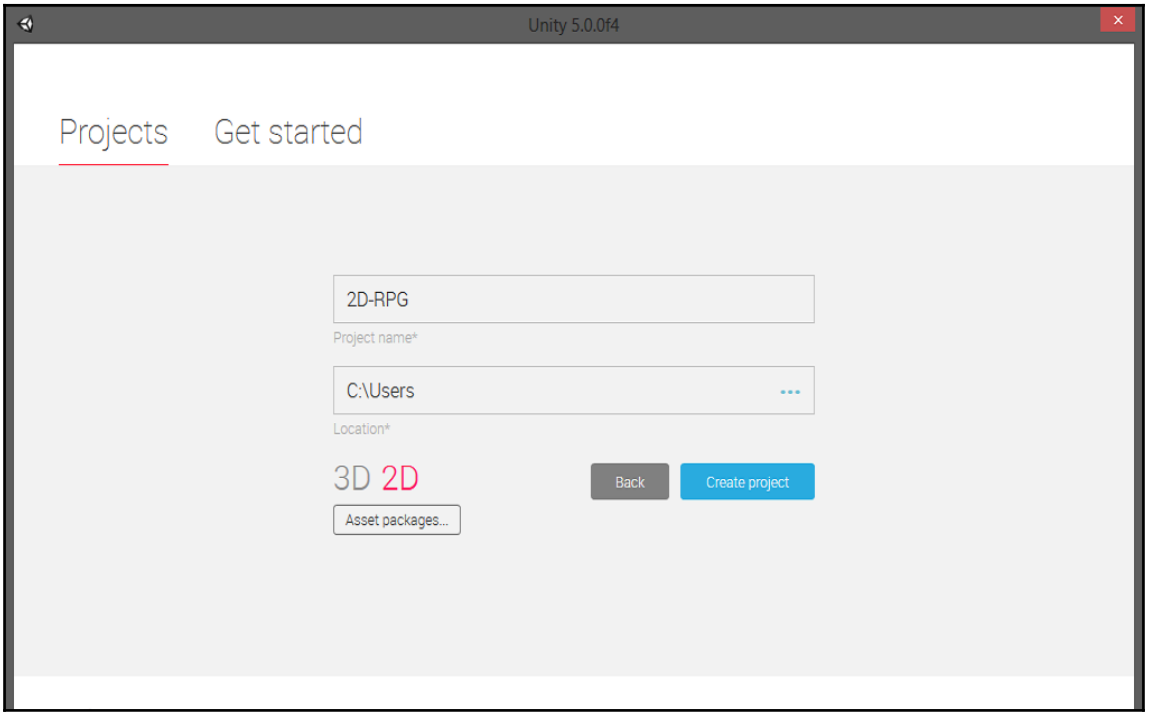
In previous chapters, we worked on our platformer game while tinkering with the Unity 2D engine. Now, we will continue doing the same while creating our own **Role-Playing Games** (**RPG**).

RPGs can always be fun to create and play; that's one of the reasons as to why they are so popular, regardless of whether they are 2D or 3D. One of the main differences between a 2D platform and 2D RPGs is that the second one has the top view rather than the side one. This has a big impact on both designing maps and programming the physics and movement of our characters.

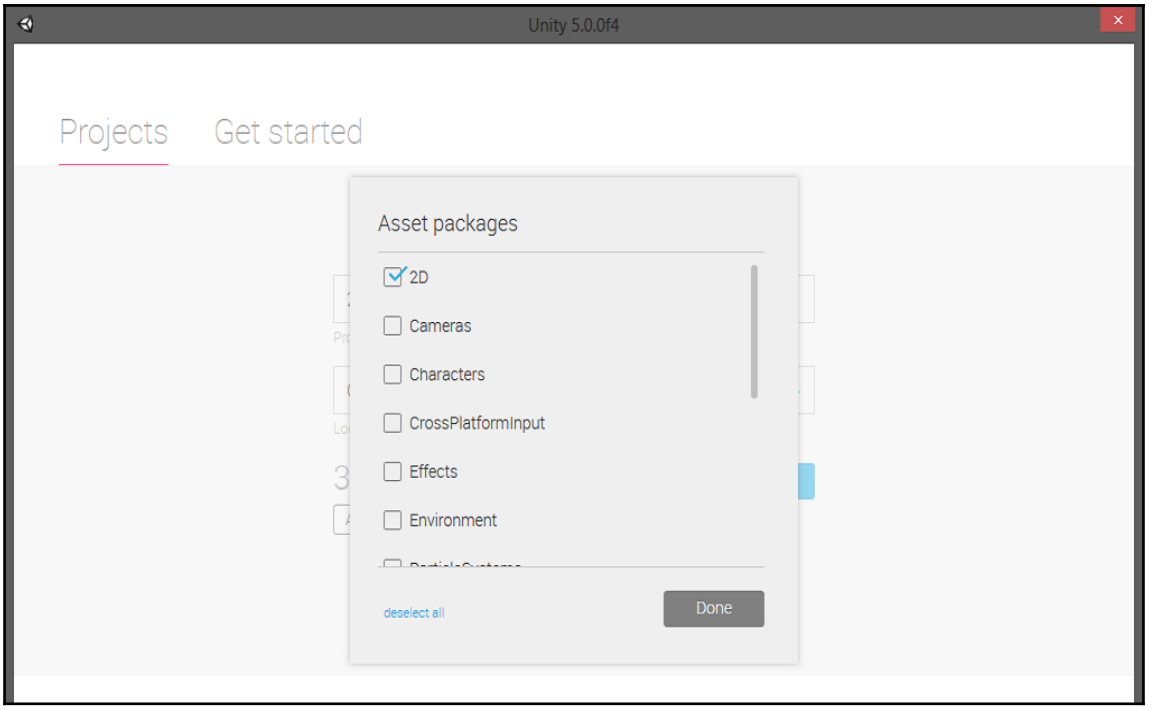
Before starting to create our RPG, we need to get our assets ready that we will use in the game.

Getting ready

Let's open up Unity and create a new project, as shown in the following image:

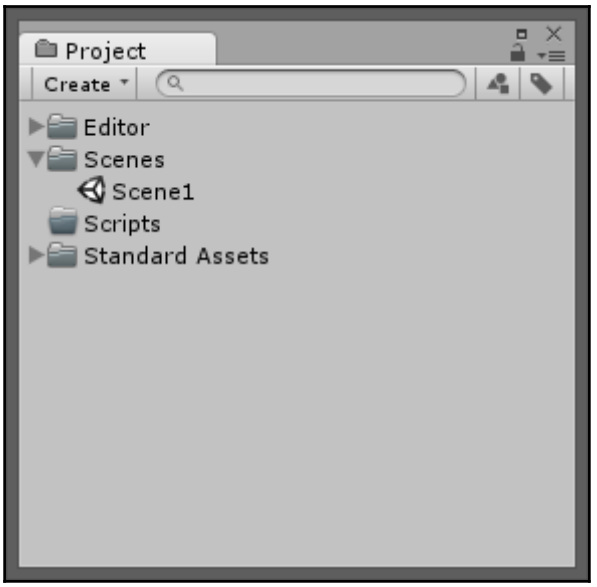


Make sure that the 2D mode is selected, click on **Asset packages…**, and select **2D**, as shown in the following image:



Keep in mind that the `Standard Assets` need to be downloaded from the Asset Store, as we did in *Chapter 1*, *Sprites*.

Press the **Done** button, and as a result the project is created. In the **Project** panel, create two new folders named `Scripts` and `Scenes`. Inside `Scenes` save an empty scene and name it `Scene1`, as shown in the following image:

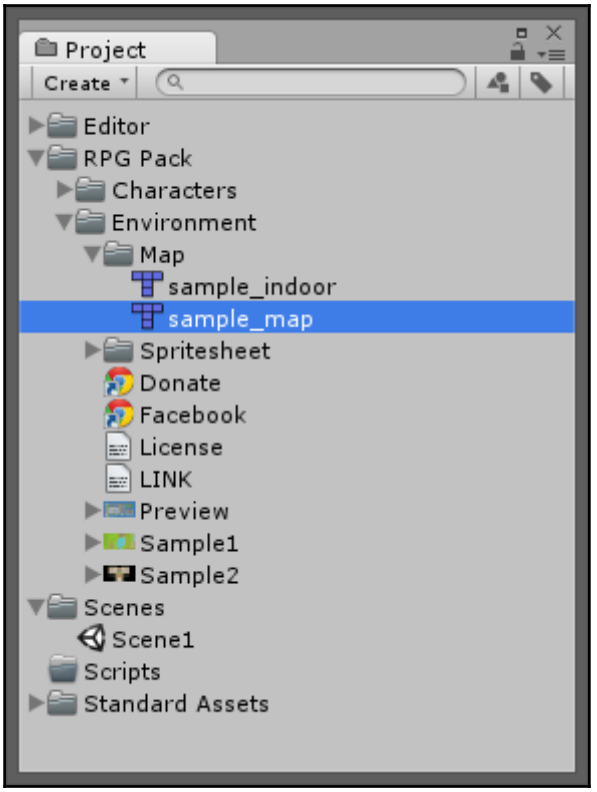


One last thing to do is to import our assets. As we did in *Chapter 1*, *Stripes*, we can use the free assets from the `http://kenney.nl` website (just remember that we will also need software to decompress). In particular, we need the RPG packs. You can find them at the following link: `http://kenney.nl/assets?s=rpg`.

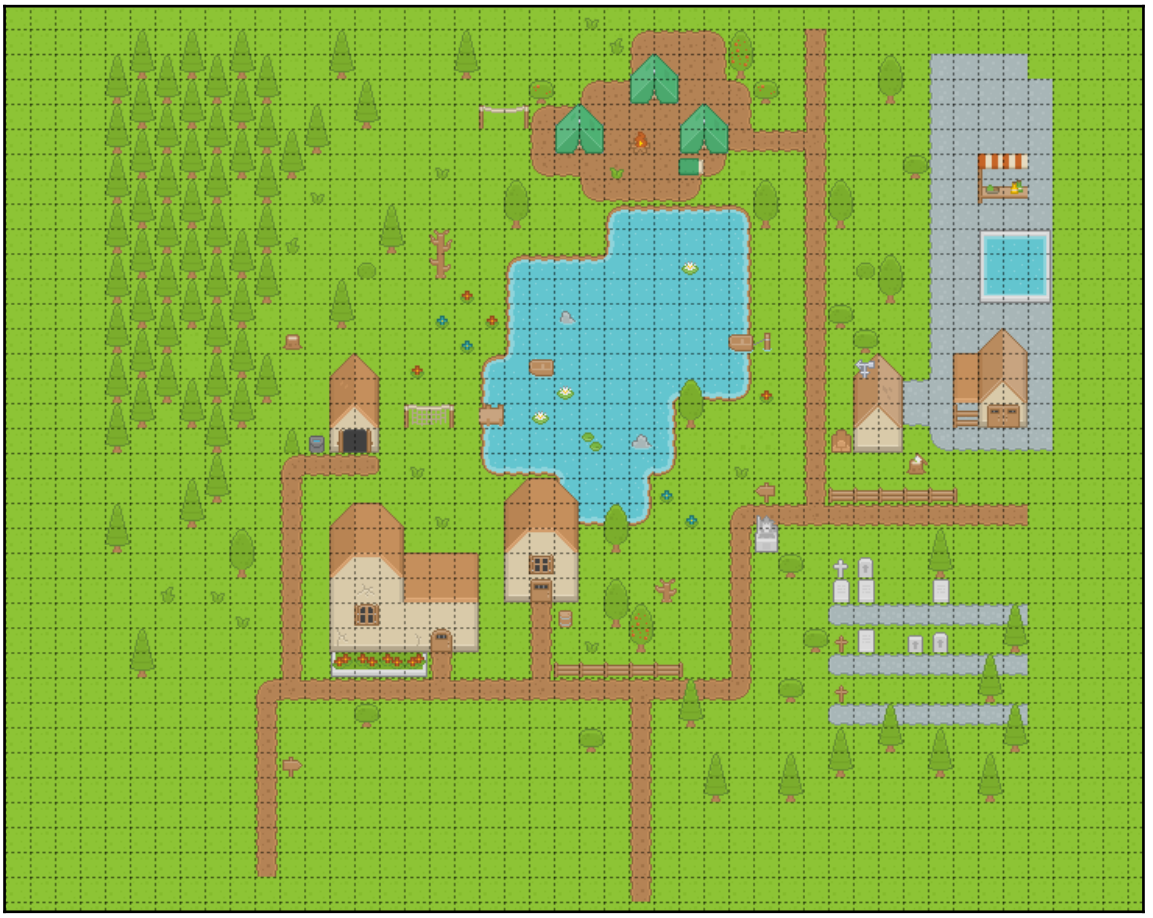
Of course, don't forget to import them into Unity before continuing on with the chapter. Furthermore, you can also reorder them, so that they are easier to find later. For instance, you can start to place all the packages into one folder called `RPGPack`. Then, divide this folder into sub-folders such as `Characters` and `Environment`. You will see this done throughout the rest of the chapter.

Importing the level

In order to create our scene, we will need to play around with our new assets in `Tiled`, but luckily, a lot has already been set up for us. Open `roguelike-pack/Map/sample_map` in `Tiled`. If you missed the previous chapter where we discussed `Tiled`, you should at least read that section before you continue. In fact, it is another software, separate from Unity:

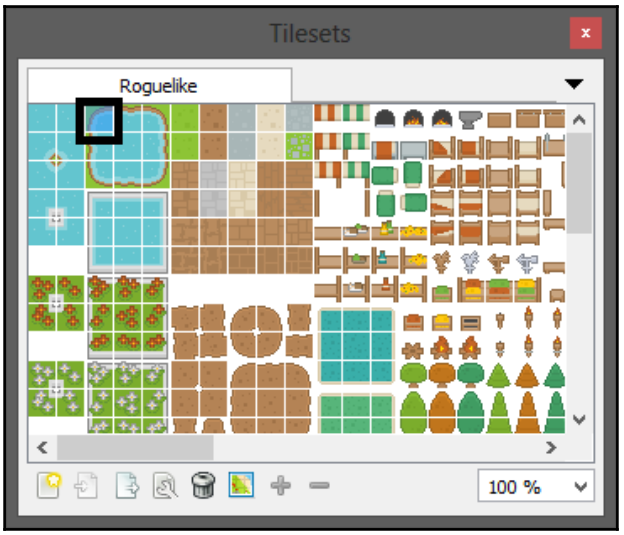


As you can see, the scene is almost complete. We will not need to spend much time to make it game-ready:

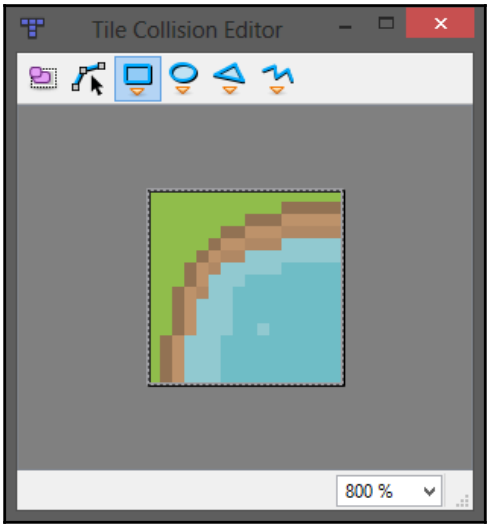


Now, we need to set up the colliders for our scene and it's best to do it from here. This task can be tiring and boring, but it's definitely worth it.

Let's start by selecting the lake's top-left sprite inside the tileset:

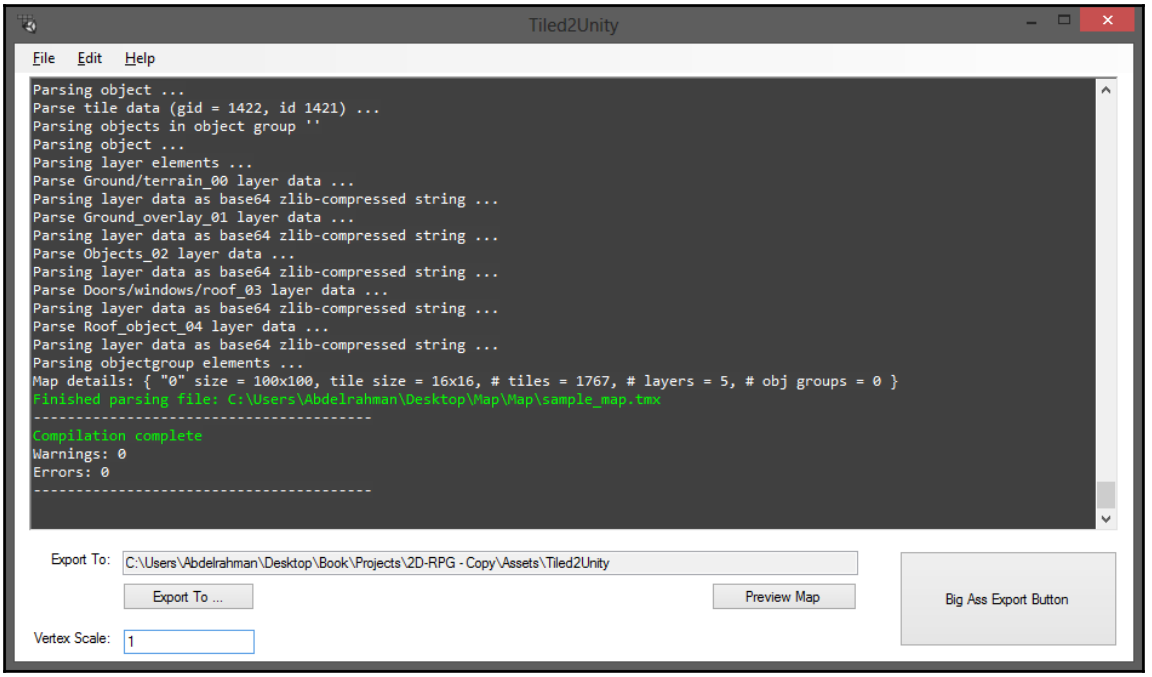


Now go to **View** | **Tile Collision Editor** and enclose it with the rectangular tool so that it looks as follows:

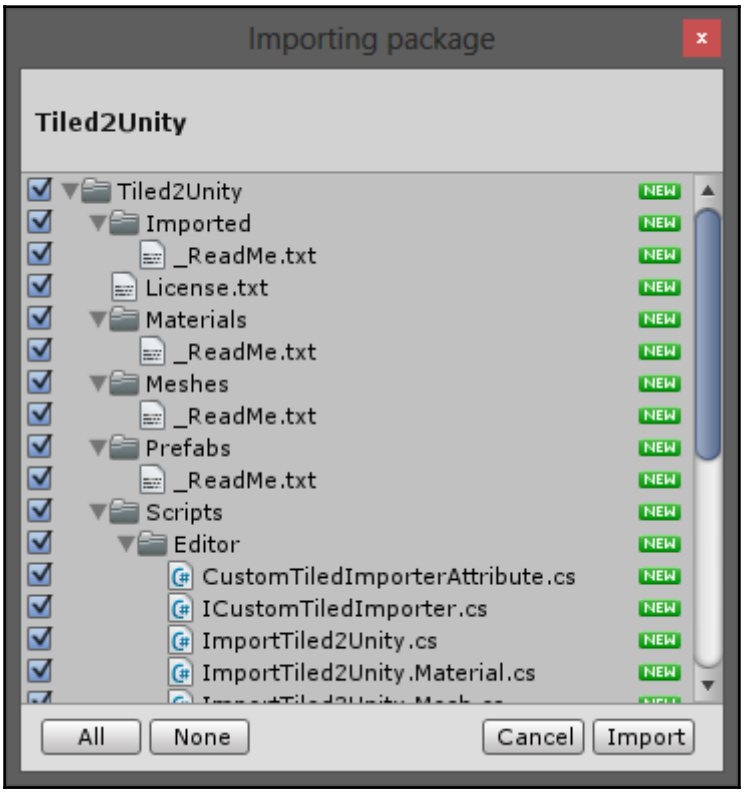


Unfortunately, we need to do the same to every sprite object that should be a collider to the player inside the game; these include: tree trunks, houses, roofs, tents, crosses, and so on…

After all the **Colliders** have been properly set up, save the map and open the file from `Tiled2Unity`, as shown in the following image:

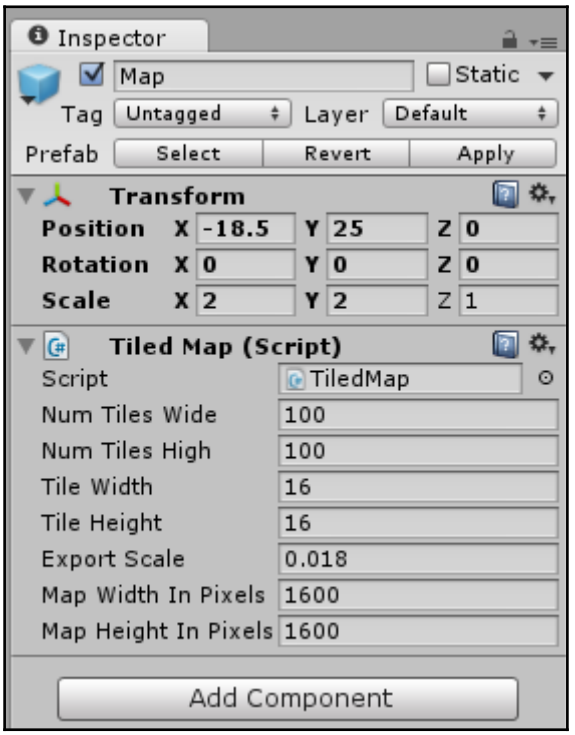


Go to: **Help** | **Import Unity Package To Project** then click on **Import** inside Unity:



You can make sure the colliders are properly set up by clicking on **Preview**. Adjust the scale to `0.018` and make sure that **Export To** is pointing at the `Tiled2Unity.export` file inside our project at `Assets/Tiled2Unity`. Finally, press **Export** to add the map to our project.

After the map has been imported, drag the `sample_map` prefab from **Tiled2Unity** | **Prefabs** to our scene and rename it `Map`. Furthermore, set its **Rect Transform**, as follows:



Now, we need to rename the camera object from `MainCamera` to `Camera` and set the camera size to `3.5`. The scene should look like this now:

Slicing the sprites for our hero

Now that the scene is ready to go, we need to create our `Player` object. However, before doing so, we need to slice our **Spritesheet**.

Open the file `roguelikeChar_transparent`, which you can find inside the `roguelike-pack/Map/Spritesheet` folder. As you can see, this file includes everything that can be used by the character in our game:

The image contains seven sections of sprites, which are:

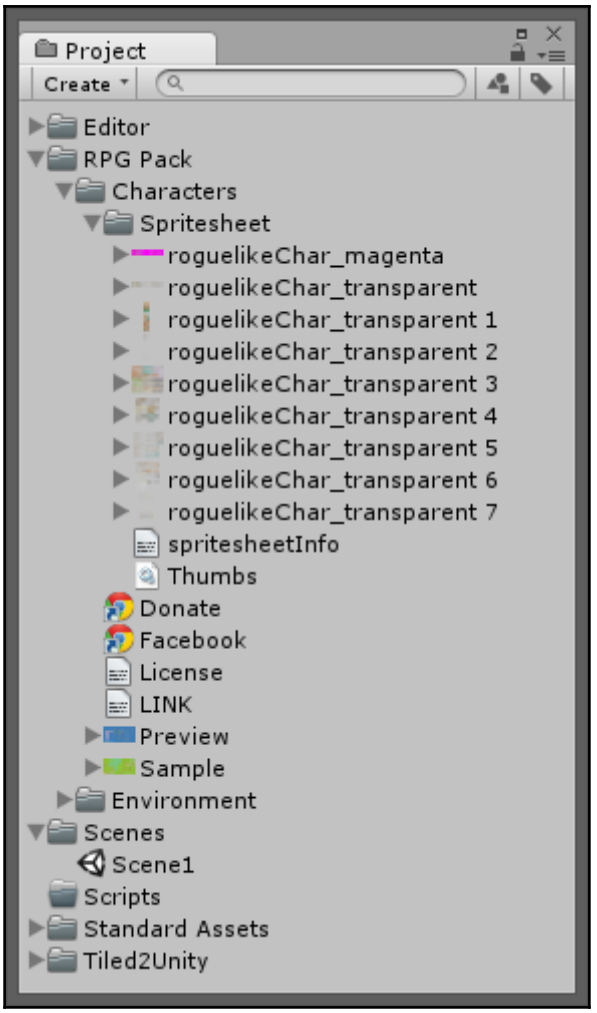
- Bodies (undressed and some dressed examples)
- Bottoms
- Tops
- Hair
- Helmets
- Shields
- Weapons

To smoothly slice the sprites in Unity's **Sprite Editor**, we will need to manually divide the image to seven files using your favorite image editor. By doing this, each file will only contain one of the sections in a nicely enclosed manner.

For future reference, the image files should be named as follows:

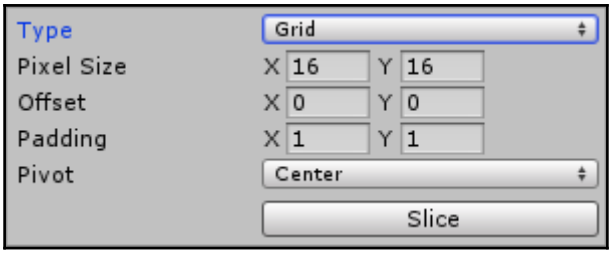
- Bodies: `roguelikeChar_transparent1`
- Bottoms: `roguelikeChar_transparent2`
- Tops: `roguelikeChar_transparent3`
- Hair: `roguelikeChar_transparent6`
- Helmets: `roguelikeChar_transparent7`
- Shields: `roguelikeChar_transparent4`
- Weapons: `roguelikeChar_transparent5`

By importing them into Unity again, we will have the **Project** panel, as shown in the following image:

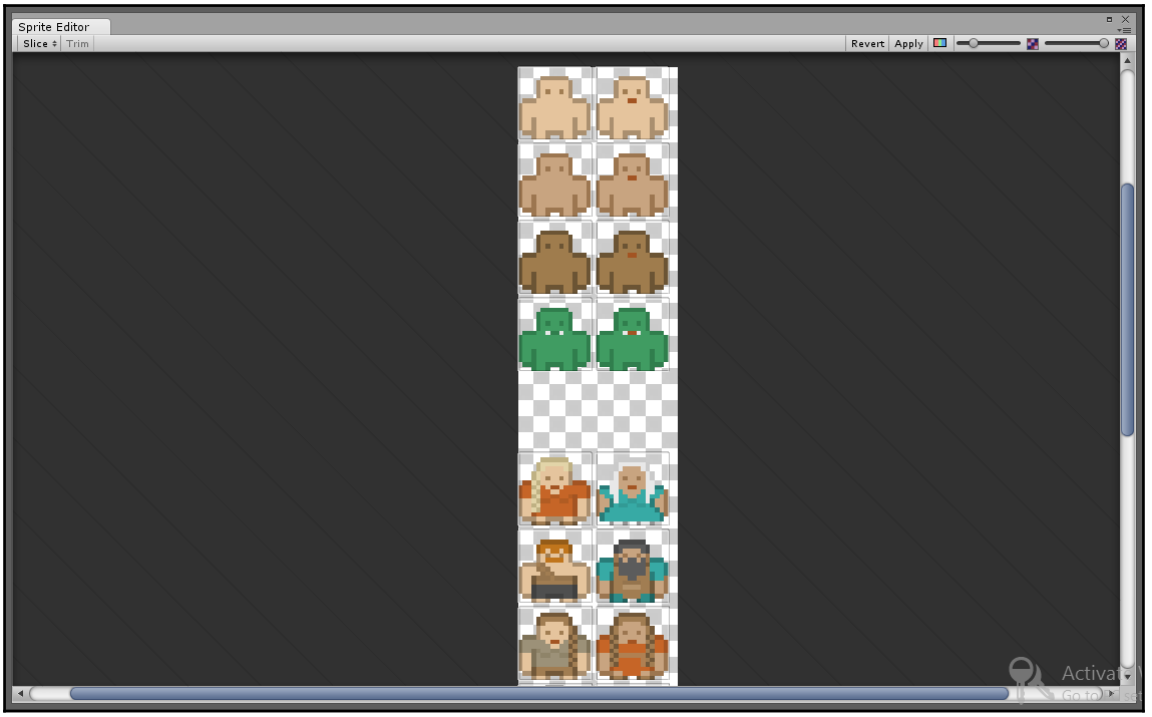


Now that the files are ready, let's start slicing the sprites. In the **Project** panel, choose `roguelikeChar_transparent1` and then from the **Inspector**, change the sprite mode to `Multiple`. Finally, click on **Sprite Editor**.

In the **Sprite Editor**, open the **Slice** menu. Set the properties as shown in the following image, and press **Slice**:



The spritesheet should now look as follows (you can erase the two empty sprites):

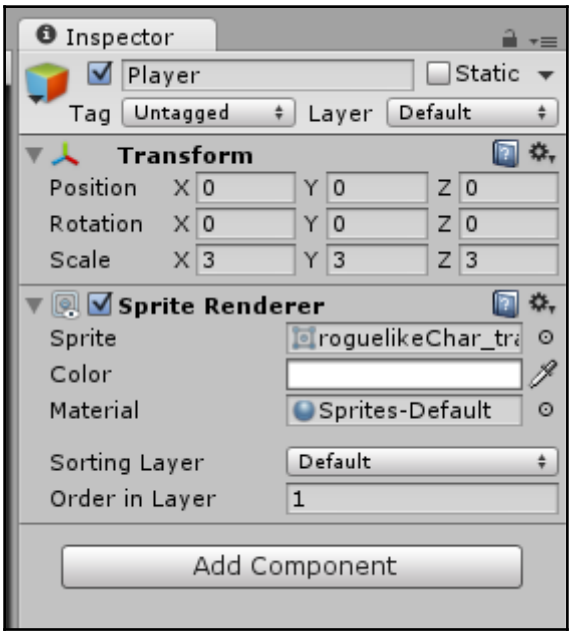


We need to repeat the same process for the other six spritesheets.

After the necessary changes have been made, we can create our `player` object.

Creating our hero

Let's navigate to the `roguelikeChar_transparent1` spritesheet and drag its first sprite `roguelikeChar_transparent1_0` into our scene. Rename the object to `Player`, set the **Order in Layer** in the **Sprite Renderer** component to `10`, and then modify the **Rect Transform**, as shown in the following image:



Next, parent the `Camera` object inside the Player object so that it will follow the player in the game. The scene should look as follows:

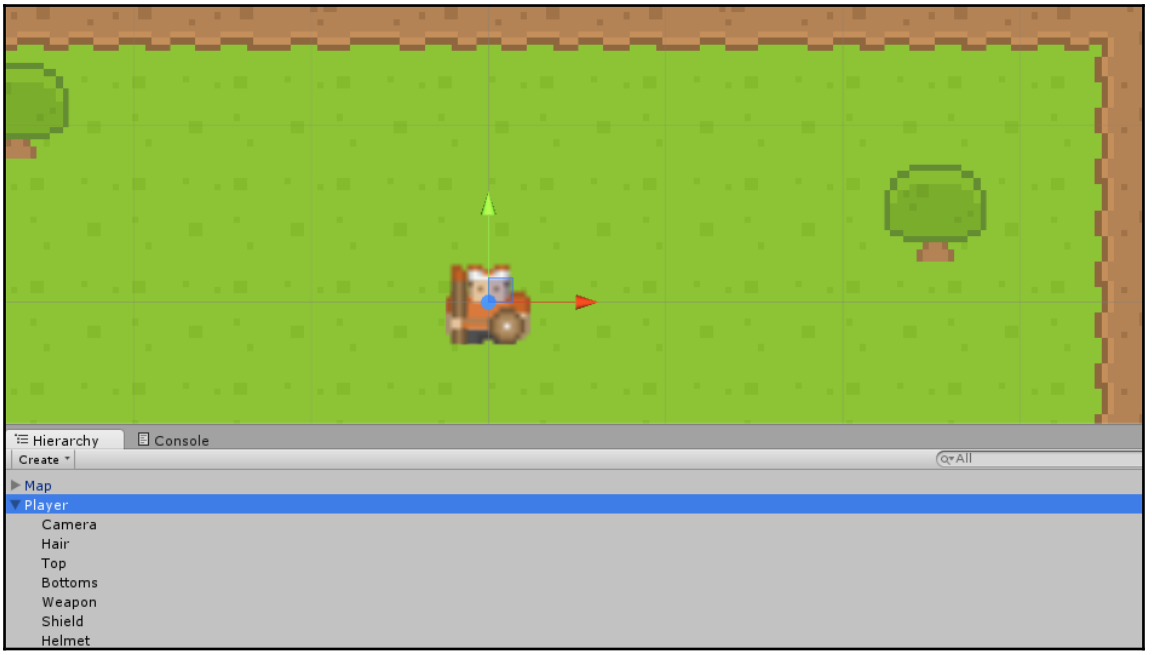
Dressing up our hero

To make our character more interesting, let's add some items to the player. Drag the following sprites from the **Project** panel onto the `Player` object:

- `roguelikeChar_transparent6_0`, rename it to `Hair` and set the **Order in Layer** variable to `11`
- `roguelikeChar_transparent3_0`, rename it to `Top` and set the **Order in Layer** variable to `12`
- `roguelikeChar_transparent2_0`, rename it to `Bottoms` and set the **Order in Layer** variable to `11`
- `roguelikeChar_transparent5_0`, rename it to `Weapon` and set the **Order in Layer** variable to `13`

- `roguelikeChar_transparent4_0`, rename it to `Shield` and set the **Order in Layer** variable to `14`
- `roguelikeChar_transparent7_8`, rename it to `Helmet` and set the **Order in Layer** variable to `12`

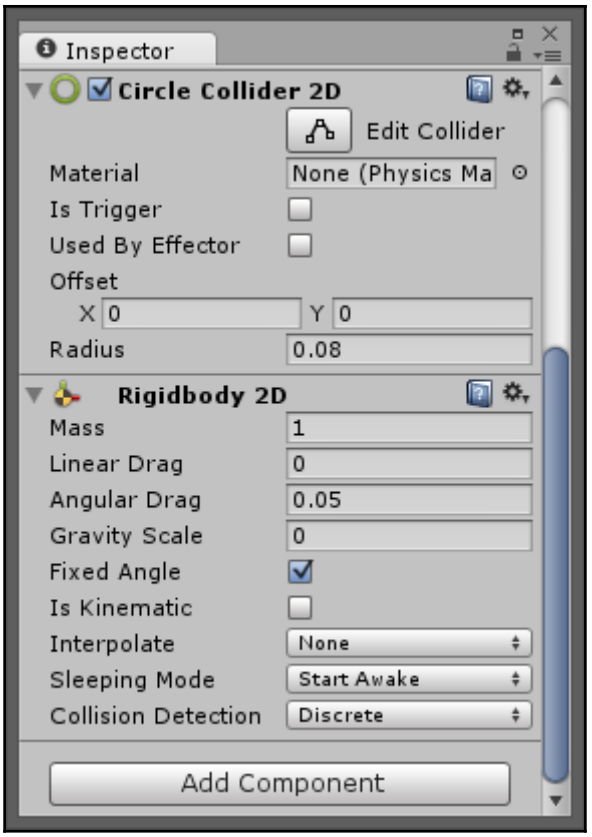
Reset the `Rect Transform` for all of the preceding objects, and as a result, our hero should be cool, as below:



Giving the power of movement to our hero

Our hero won't move unless we do something about it. We can start by adding a **Circle Collider 2D** to the player object and then a **Rigid Body 2D** too.

There are two main things that we are going to change in the rigid body. The first is setting the **Gravity Scale** to 0. As a result, our hero will not go downward in the map. Then, we will set the **Fixed Angle** variable to `true`. By doing this, our hero won't rotate when he or she collides with something. You can see the properties set in the following image:



Our hero is now ready to deal with the physics engine of Unity. However, some extra custom code is still needed to move him or her according to the player's input. Let's create a new C# script and call it `HeroMovement` inside the folder `Scripts`. Then, attach the script to our hero. As usual, double-click on the script to open it.

The first thing that we need to do is to add two variables. One is needed to store the speed of our hero and another one to get the reference to his `RigidBody2D`:

```
public float speed = 4.0f;
Rigidbody2D playerRigidBody2D;
```

In the `Start()` function, we can get the reference to the `RigidBody2D` by calling the `GetComponent()` function, as follows:

```
void Start() {
    // Get the RigidBody component
    playerRigidBody2D = GetComponent<Rigidbody2D>();
}
```

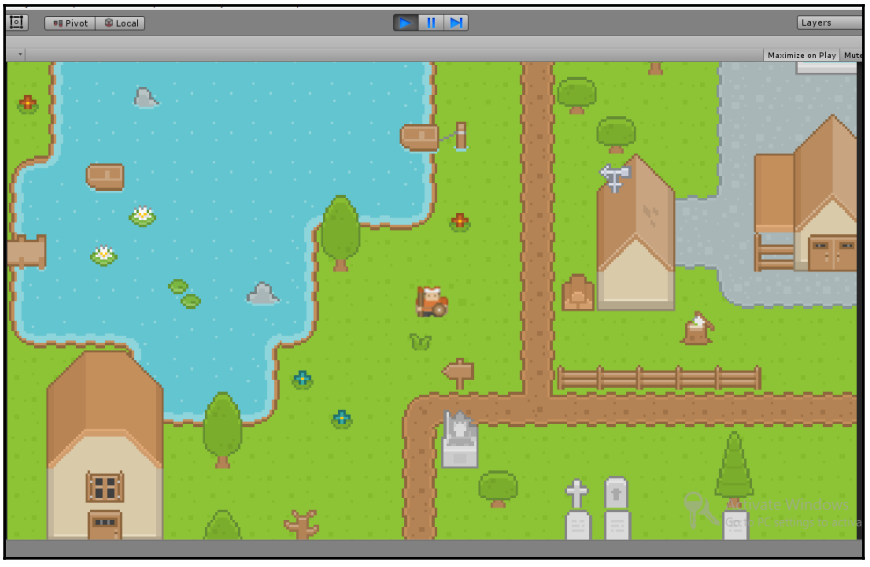
Finally, in the `Update()` function, similar to what we did in *Chapter 1*, *Sprites*, we need to give a velocity to his `RigidBody` based on the player input. However, in this case, we are going to use both axes, the vertical and the horizontal:

```
void Update() {
    float movePlayerX = Input.GetAxis("Horizontal");
    float movePlayerY = Input.GetAxis("Vertical");
    playerRigidBody2D.velocity = new Vector2(movePlayerX * speed,
movePlayerY * speed);
    }
```

Save the changes and head back to Unity.

We can test the scene and see what we have accomplished so far by pressing the play button:

Once we have pressed play, we can move our hero:



Take your time to explore the map with your hero, to test that all the colliders are properly set.

As you can see, now we can move the player while it is reacting to the level physics. However, our hero looks a little static, since it doesn't animate, and we would like to change that. In the next section, we can give him or her a little bit of life from inside the script itself.

Animating the hero

We are now going to create a script that moves the weapon and the shield items upward then in a downward fashion to simulate a simple breathing look. Let's create another C# script and name it `Breather` inside the `Scripts` folder. Next, attach the script to both the `weapon` and `shield` object under the player. Double-click on the script to open it.

First we need to add a couple of variables to store the local position to make our hero breath:

```
public Vector2 position1;
public Vector2 position2;
```

Then, a third variable stores the time between the two positions defined before:

```
public float waitTime;
```

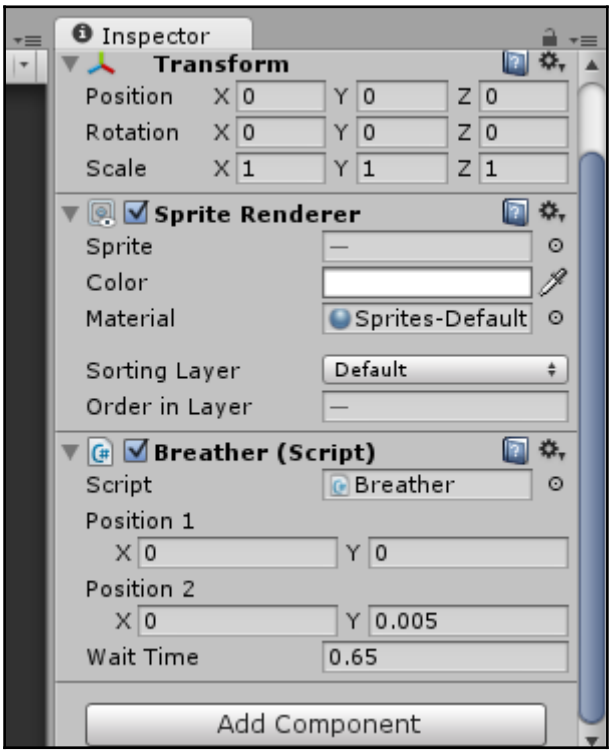
In the `Start()` function, we start the `coroutine` we are going to implement:

```
void Start() {
    StartCoroutine(Mover());
}
```

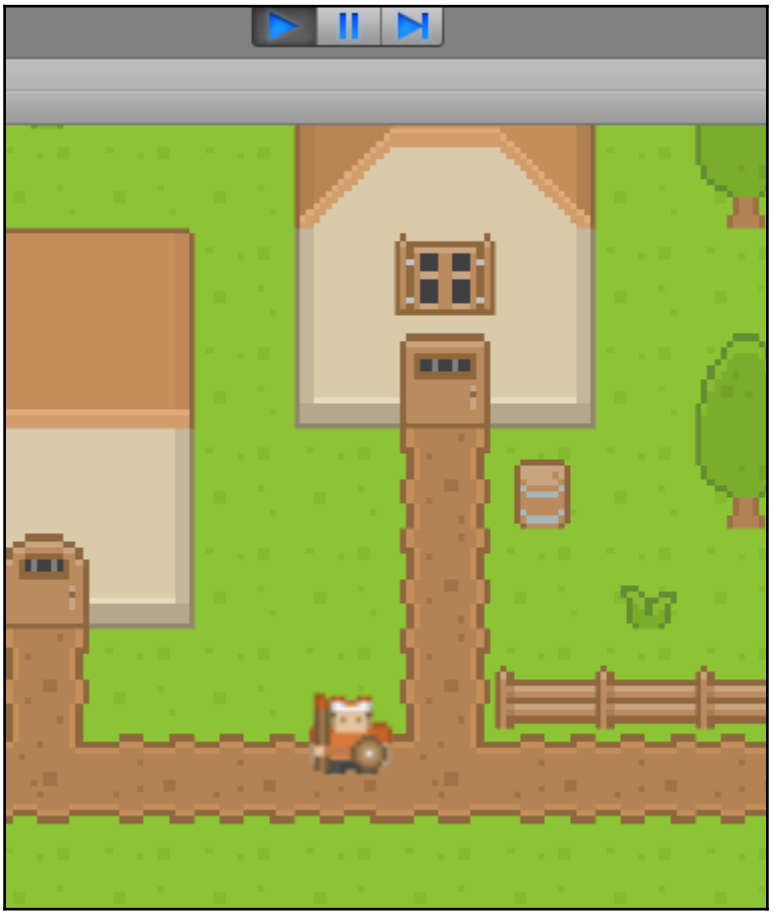
Finally, in our `coroutine`, we start at a random moment and then there is a main loop that changes between the two positions:

```
IEnumerator Mover() {
    yield return new WaitForSeconds(Random.Range(0, 10) / 10);
    while (true) {
        transform.localPosition = position1;
        yield return new WaitForSeconds(waitTime);
        transform.localPosition = position2;
        yield return new WaitForSeconds(waitTime);
    }
}
```

Save the changes and head back to the scene. Modify the `breather` component in both objects, `Weapon` and `Shield`, to look as shown in the following image:



We can now test the scene and see how the character looks as it simulates a simple breathing animation:



Once we have pressed play, we can see our hero breathing:

The effect of breathing is achieved by moving the Weapon and the Shield.

Summary

In this chapter, we started the RPG project, imported the game level, and added our hero to the map. After that, we tweaked the hero's equipment in order to make the character look a bit more hero-like. Lastly, we gave our hero life by creating a breathing animation.

In the next chapter, we will be adding more features to our game, in particular other characters!

6
AI and Pathfinding

A good RPG requires some enemies and, therefore, their AI. In this chapter, we will create an enemy script in order to understand the basics of AI in RPGs.

In particular, we will learn about pathfinding and how this can be used to create our enemy character. The following is what we will go through:

- Pathfinding
- The AStar Algorithm
- Using pathfinding for enemies

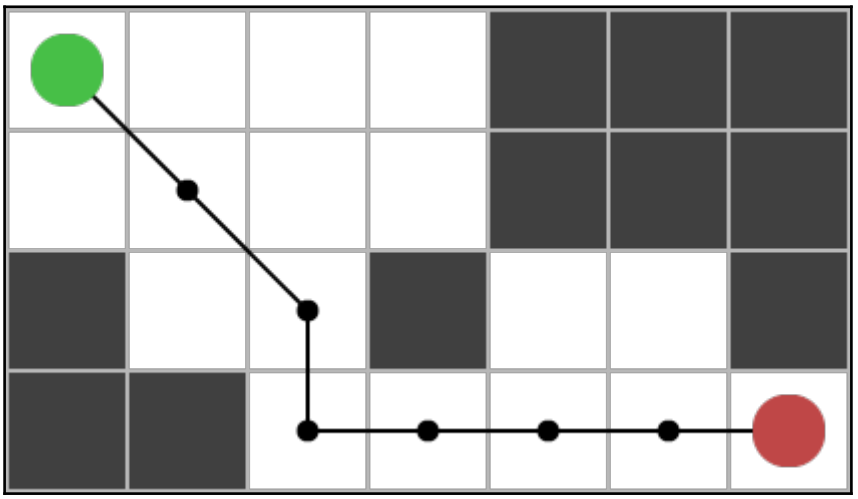
In the previous chapter, we worked on our player and designed the game level. Now we need to learn how to add enemy characters to our game.

Pathfinding

Pathfinding is a fundamental part of making video games. Over the past few years, it has become an even more important part of video games. The main purpose of pathfinding is to navigate a certain game object (AI character) around the scene by finding paths to overcome any obstacle in the way.

There are many algorithms used to compute a proper path, but the most common is the AStar Algorithm (also referred to as *A**), which accomplishes exactly that, with efficiency.

In the following example, you can see a calculated path starting from the green circle, which connects it to the red circle while navigating around the obstacles in the scene. This is mainly what pathfinding is all about:



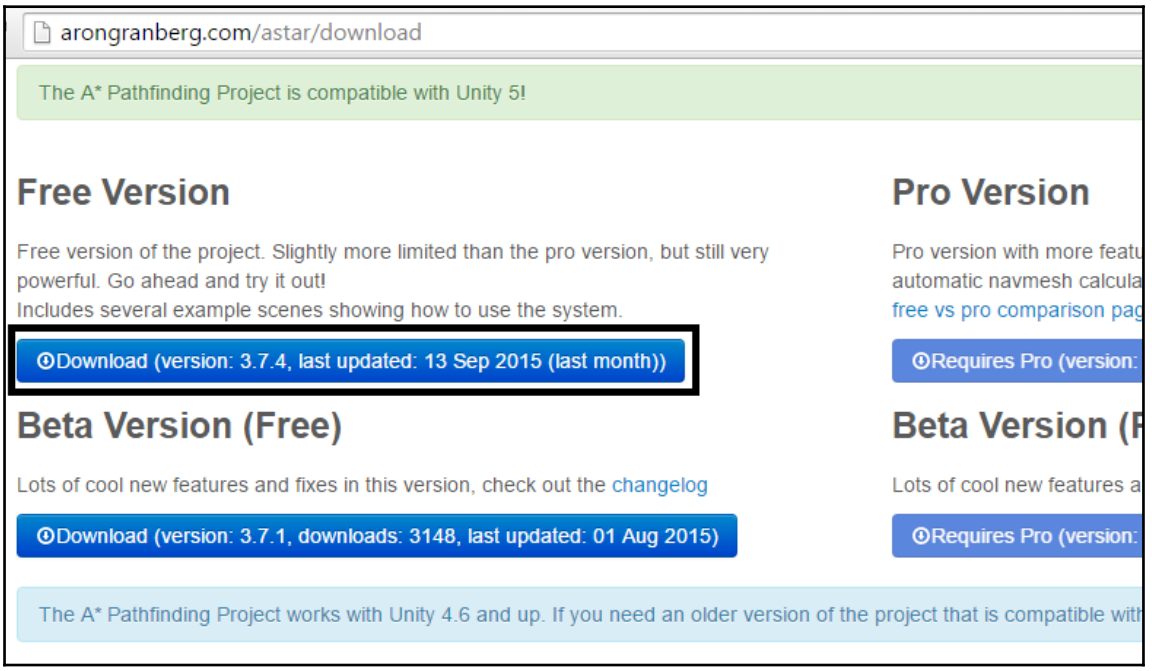
AStar Algorithm in Unity

The AStar Algorithm is famous for being one of the most reliable pathfinding algorithms out there. It is mainly used to find a proper traversable path between two points, as shown in the preceding figure.

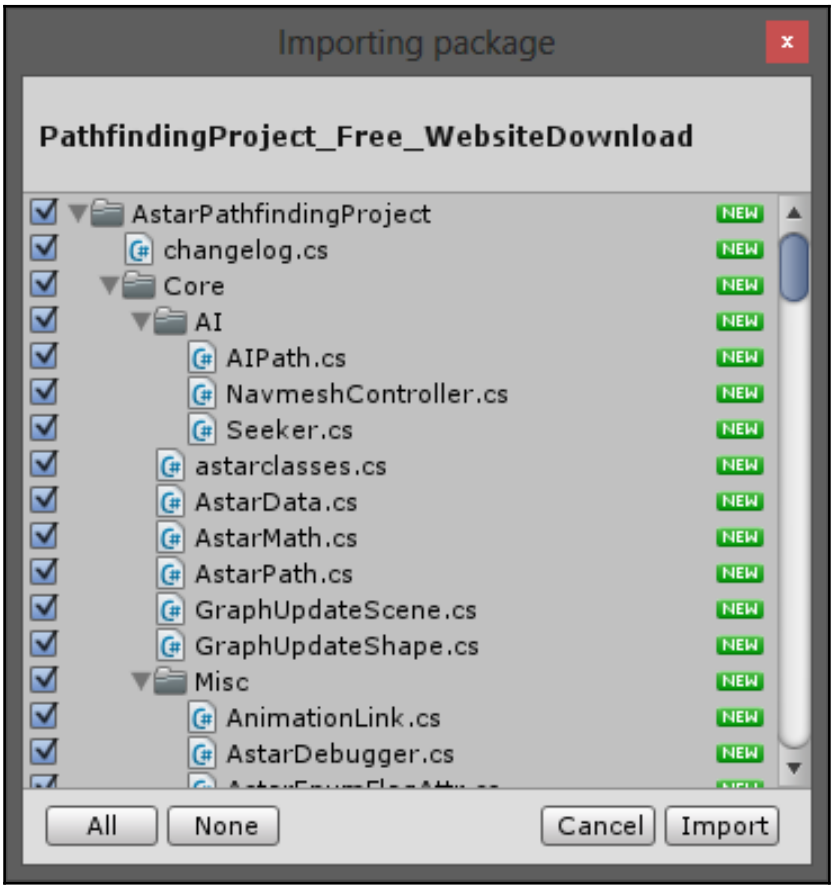
We won't be diving into the specifics of the AStar Algorithm, but we will use it in our game to plan the paths of AI characters.

A tool for Unity

Luckily, a great tool to make the process of implementing AStar in Unity already exists, and it is also freely available. It can be downloaded here: `http://arongranberg.com/astar/download`. I suggest downloading the latest stable release and avoid any beta version for any bugs that it may have, as shown in the following image:



After extracting the downloaded archive, double-click on the package in order to import the necessary files inside Unity. The following screen should appear, and then click on **Import**:



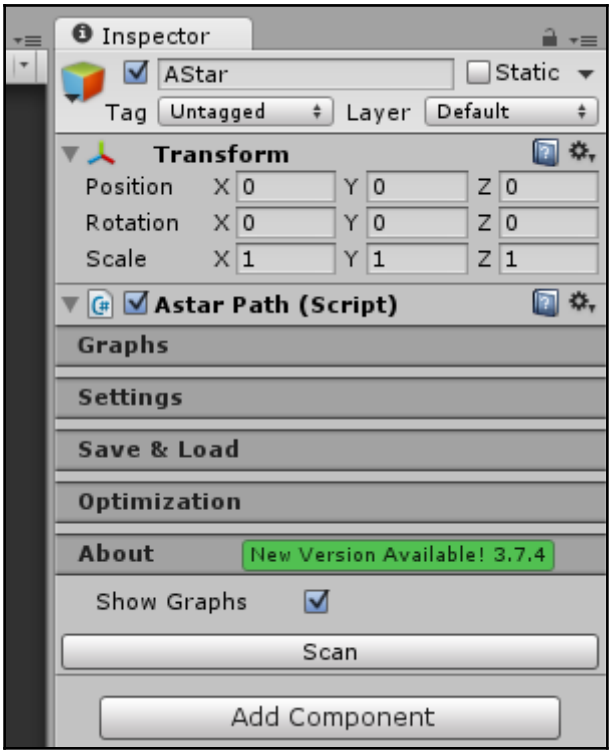
Afterwards, a new folder will be created, named `AstarPathfindingProject`, containing all of the files we are going to use for the pathfinding.

If you feel like learning more about this tool, there is very good documentation at the following link: `http://arongranberg.com/astar/docs`.

Setting up the tool

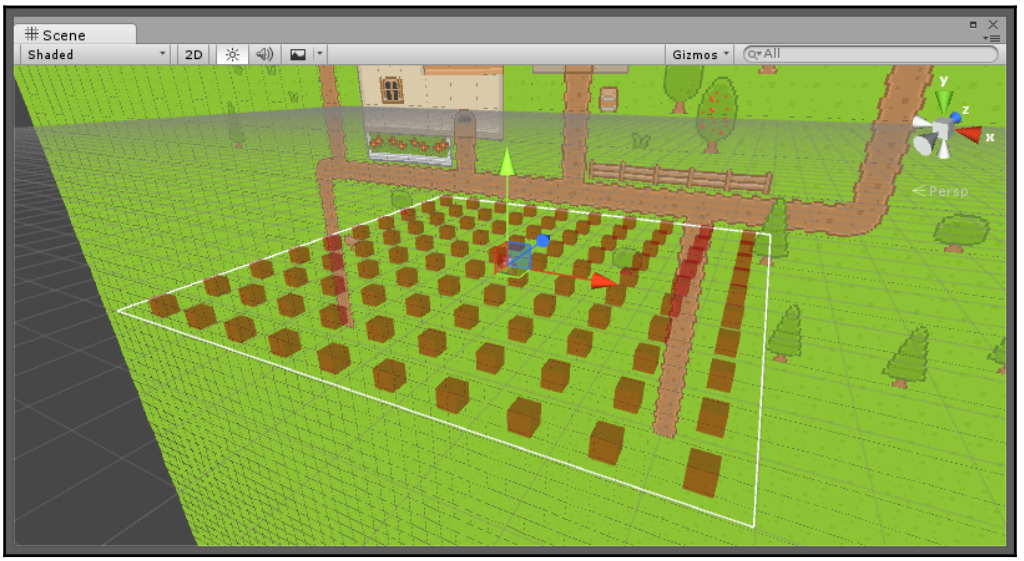
Now, in order to start using this tool in our game, we need to create an empty game object called `AStar` and add the `Pathfinder` component. This can be achieved by clicking on **Add Component** | **Pathfinding** | **Pathfinder**.

The `AStar` object should look like this in the **Inspector** window:

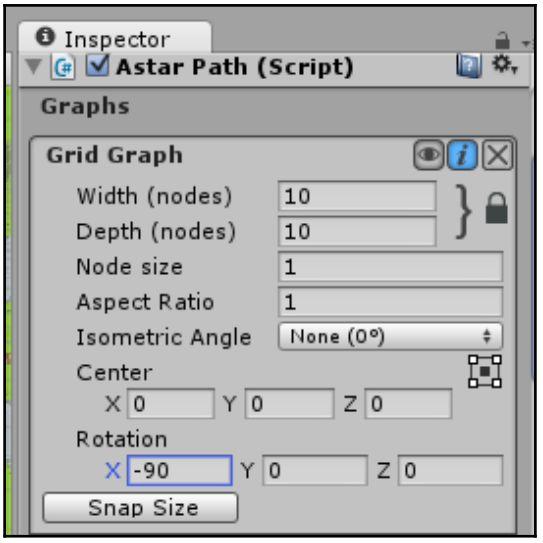


One of the first steps in pathfinding is marking *path ways* and any obstacles in the scene, so that the AI character will know where it's okay to move and where it's not. In order to accomplish this, we need to create a grid graph.

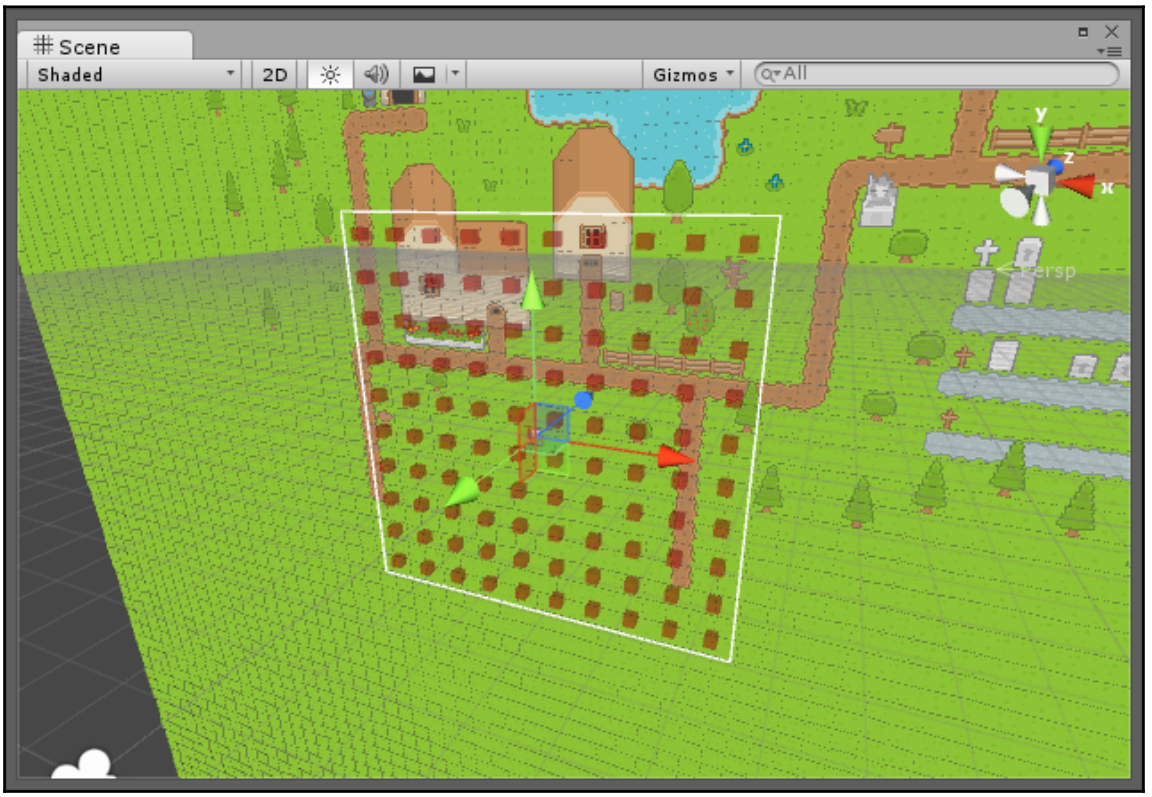
To create a grid graph, first navigate to the scene view and return to 3D mode. Next, click on **Graphs** under the **Astar Path** component and choose **Grid Graph**, as shown in the following image:



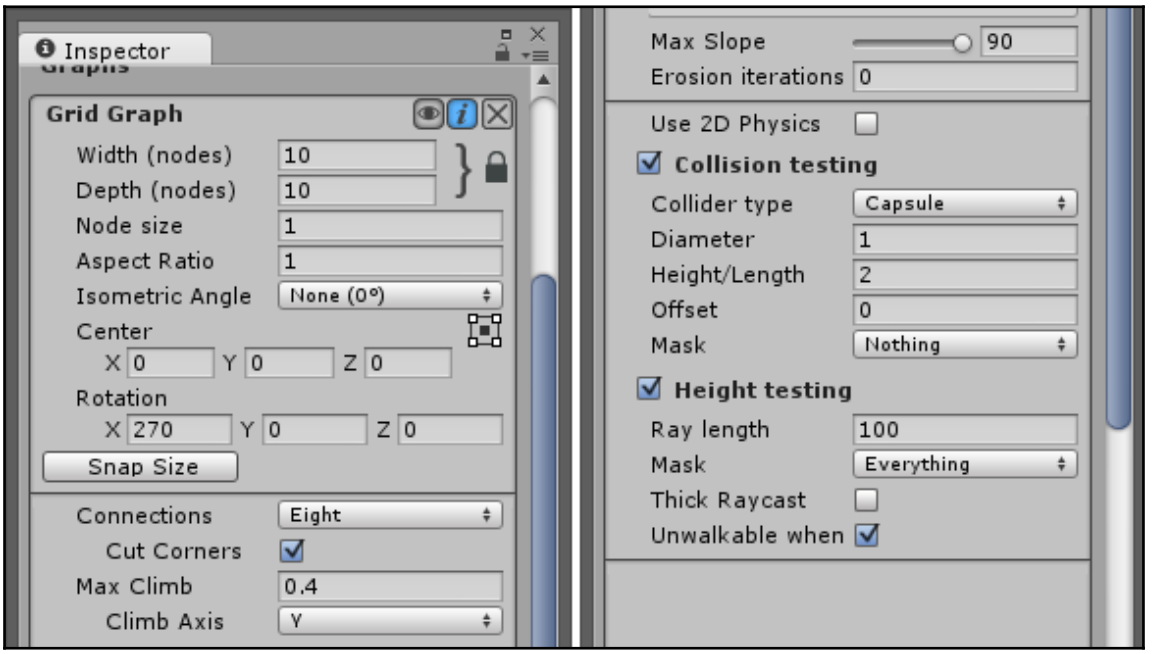
As you can see, a grid has been generated with nodes covering it. These nodes will be used to mark objects that will interfere with the player path and to generate paths around them. However, first we need to rotate the grid since it's on the wrong axis:



As shown in the preceding screenshot, inside the **Inspector**, we should change the grid graph's rotation around the *X*-axis to `-90`, since they describe the same rotation. Then, click on **Scan** at the bottom of the component. This, is the final result in our **Scene** view:



In fact, you will find that the graph is now aligned properly with our scene. Now, let's get a better look at our graph properties in the following screenshot, followed by an explanation of each parameter and which values to set for our scene:

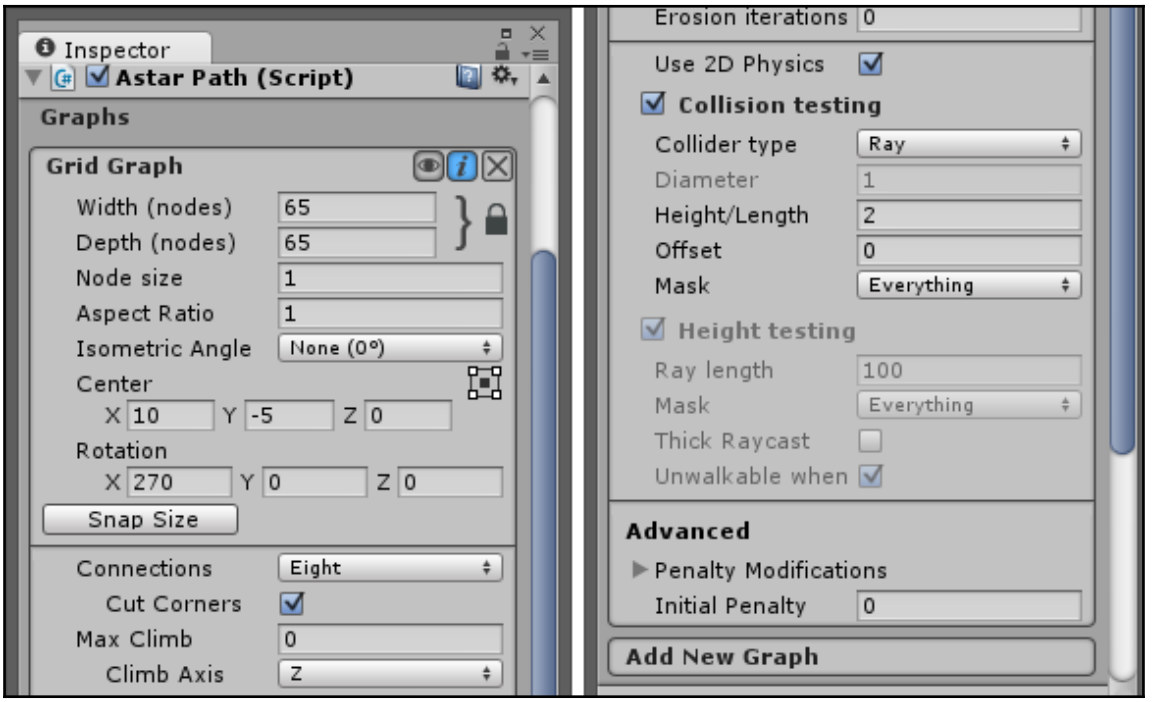


In the preceding screenshot, the parameter are as follows:

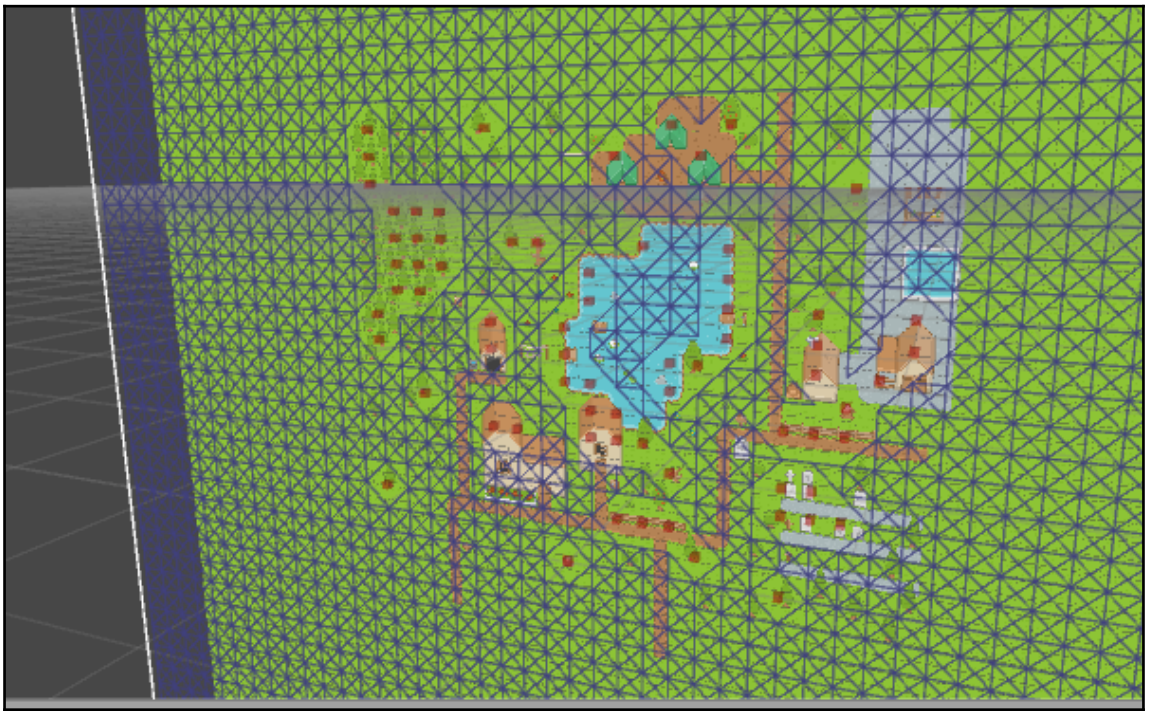
- **Width**: The number of nodes defining the grid's width. We can set it to `65`.
- **Depth**: The number of nodes defining the grid's height. Set it to `65` as well.
- **Center**: The center point of the grid, which should be set at (`10,-5,0`).
- **Rotation**: Defines the rotation of the grid, which we already set to (`270,0,0`).
- **Connections**: Number of connections between each node and its neighbor nodes. Leave this value at `8`, so that the AI character can also move diagonally.

- **Max Climb**: Determines the ability of the character to climb certain objects. However, since our game is in 2D, we do not need it. So, change the value to 0 and the **Climb Axis** to `Z`.
- **Use 2D Physics**: A flag to determine whether the components should use the 2D physics or not. Of course, set it to true.
- **Collider type**: Defines which kind of collision should be used. Since we are in 2D mode, the `Ray` collider will be sufficient.
- **Mask**: Determines which kind of object should be taken into account. Change it to **Everything**, which means that any object with proper colliders attached to it will be treated as an obstacle.

After you have made all the above changes, click on **Scan** and don't forget to save the scene. At the end, the component should look like the one in the following image:



You can double-check with the previous image to make sure that all the values are properly set:



Now, if you look closely at the grid, you should be able to see that the nodes are surrounding the obstacles in the scene by marking them for the path, which is exactly what we wanted.

Using pathfinding for enemies

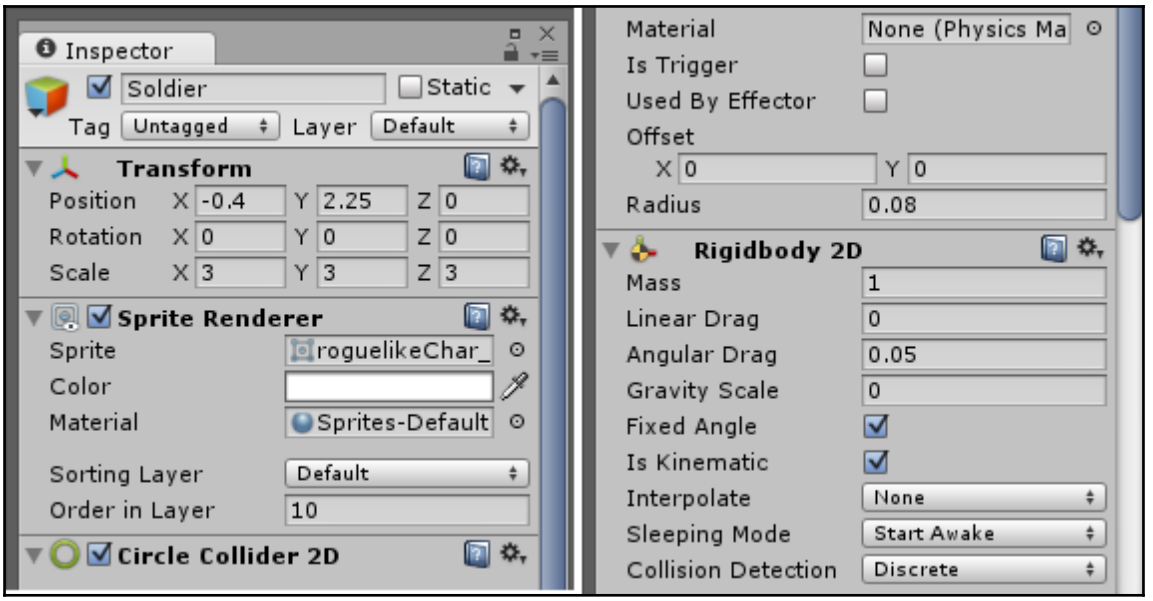
Since our scene is AI ready, we can start adding AI characters in order to test what we have done so far.

Shaping our soldier

First, drag the sprite of the enemy character into the scene. For instance, you can use `roguelikeChar_transparent 1_21`, which resembles a soldier figure and can be found in the `RPG Pack/Characters/Spritesheet/roguelikeChar_transparent 1` folder. You can see it in the following image:

This is the sprite that we will use in this chapter. Once we are in the scene, rename the `gameObject` to `Soldier`, and set its position to (`-0.4,2.25,0`) and its scale to (`3,3,3`).

Then, add a circle collider and a rigid body to our soldier by setting the same variable values that we already set in our `player` object. The only difference is that we need to set the rigid body's `IsKinematic` value to true, since we will be moving the character by a script that doesn't require any input from the player, unlike the player `gameObject`. You can see the final settings in the following image:



Every brave soldier needs a weapon. Therefore, let's make a copy of the weapon and the shield objects inside the `player` object and parent these copies to our soldier. Then, be sure to reset the position of the two objects to `(0,0,0)`, because they probably still hold their old positions.

You can change the weapon and armor sprites to better suit our soldier, as you can see in the following screenshot:

In order to start using the paths created on the grid by the pathfinding tool, we need to attach the `Seeker` component to our soldier. Click on **Add Component** | **Pathfinding** | **Seeker**.

Giving intelligence to the soldier

Now that our soldier is prepared to navigate through the scene using the grid graph, we have to implement a new script to control its behavior. In particular, we want the soldier to chase the player. When the soldier is too close to the player, then the soldier stops.

Let's create a new C# script and name it `EnemyAI` under the folder `Scripts`. Then, attach the script to the soldier object. Open the script in the editor.

The first thing to do is to add the following line at the beginning of the script:

```
using Pathfinding;
```

This allows us to use the `Pathfinding` package that we have imported.

Then, we need to create some variables; first of all, our target. This variable contains the `Transform` that the enemy is chasing; in our case, the player. Therefore, we can write:

```
// The target to follow or chase
public Transform target;
```

The next four variables are needed to describe the behavior of our soldier. In particular, one determines the speed of the character, another one how often the soldier should update its path. The third describes how close he or she needs to be a waypoint before changing direction; and finally, how far the character should be from the player to chase him or her. These variables are written in the following code:

```
// Character's movement speed
public float speed = 85;
// How often the path is updated every second
public float updateRate = 2;
// Required distance to be reached before continuing the path
public float nextWaypointDistance = 0.3f;
// Required distance to the target before stopping
public float endDistance = 0.8f;
```

Now, we need some variables to handle the internal logic. One variable is a flag to check whether the path has come to an end. Another one stores the path itself. We need a couple of variables to reference the soldier components to the `RigidBody` and the `Seeker`, respectively. Finally, we need to add an index to iterate on the path. Thus, we can add the following variables:

```
// Flag to check if the path has ended
public bool pathEnded = false;
// The calculated path
public Path path;
// Character's RigidBody component
Rigidbody2D charRigidBody2D;
// Character's Seeker component
Seeker seeker;
// Current waypoint
int curWaypointIndex=0;
```

The next step is to assign these references in the `Start()` function, and start a coroutine, which we are going to implement soon:

```
void Start () {
  // Assign the required components
  seeker=GetComponent<Seeker>();
  charRigidBody2D=GetComponent<Rigidbody2D>();
  // Start the path
  StartCoroutine(UpdatePath());
}
```

Then, in the `Update()` function, we need to make our character move. First, we need to check whether the path is completed; if not, we move the character accordingly, as written in the following code:

```
void Update () {
  if(path==null) return;
  // On reaching the end of the path
  if(curWaypointIndex==path.vectorPath.Count)
  {
    if(pathEnded) return;
    charRigidBody2D.velocity=Vector3.zero;
    pathEnded=true;
  }
  else
  {
    // If we approached the end distance
    if(Vector3.Distance(transform.position,target.position)
       <endDistance)
    {
      // Finish the path
      curWaypointIndex=path.vectorPath.Count;
    }
    else
    {
      // Move towards the current waypoint
      pathEnded=false;
      float
dist=Vector3.Distance(transform.position,path.vectorPath[curWaypointIndex])
;
      Vector3 dir=(path.vectorPath[curWaypointIndex]-
transform.position).normalized;
      charRigidBody2D.velocity=dir*speed*Time.fixedDeltaTime;
      if(dist<nextWaypointDistance) curWaypointIndex++;
    }
  }
}
```

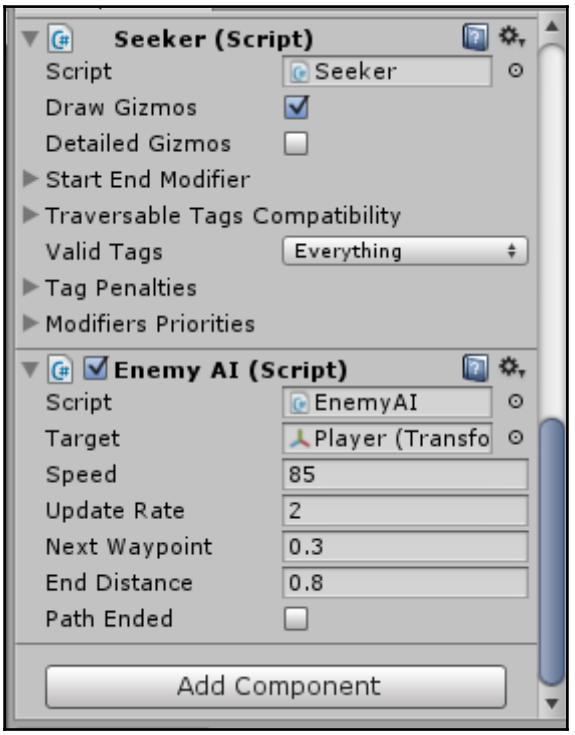
In the coroutine we started in the `Start()` function, we need to update the path at the rate specified in our variables. We can compute a new path by using the `Seeker` component. Therefore, we can write:

```
// Update the path then wait before updating again
IEnumerator UpdatePath () {
  if(Vector3.Distance(transform.position,target.position)>endDistance)
  {
    seeker.StartPath(transform.position,target.position,OnPathComplete);
  }
  yield return new WaitForSeconds(1/updateRate);
  StartCoroutine(UpdatePath());
}
```

Lastly, we have a function named `OnPathCompleted()`, which is called when the path has been computed. Here, we just assign the path computed to the path variable of our script, so to make the character move along this new path. Of course, don't forget to reset the index as well:

```
void OnPathComplete (Path p) {
  if(p.error)
  {
    // Report error
    print (p.error);
  }
  else
  {
    // Assign to the new calculated path
    path=p;
    curWaypointIndex=0;
  }
}
```

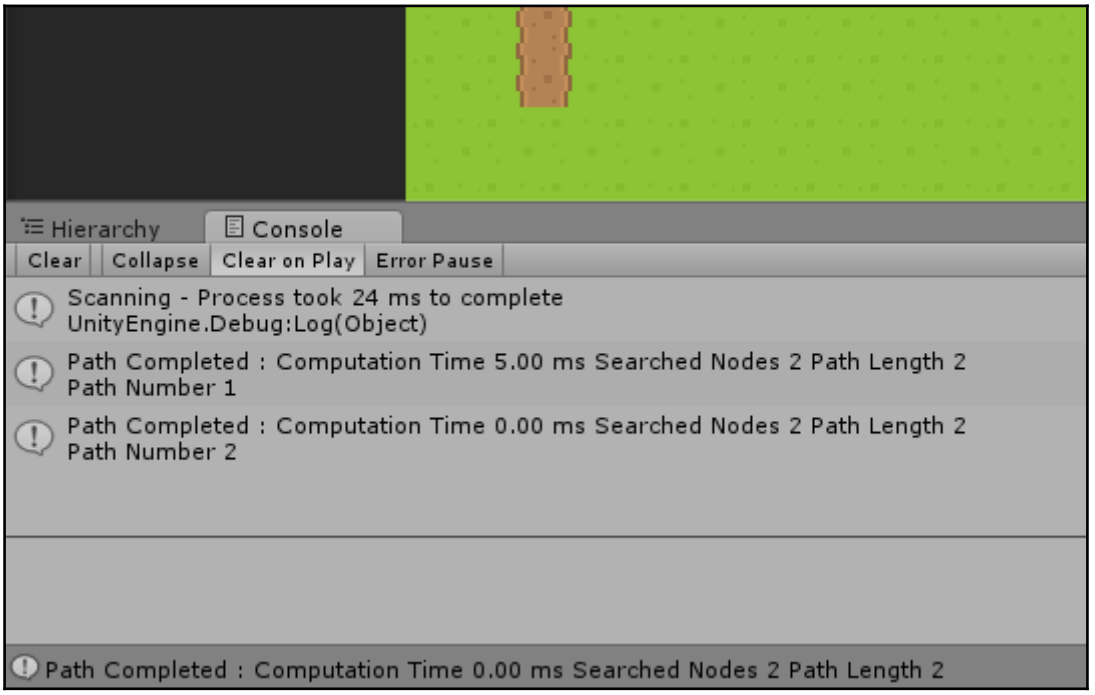
Save the changes and head back to the scene. If you haven't yet, attach the `EnemyAI` script that we just created to the soldier object. Then assign the player object to the target variable inside our script, as shown in the following screenshot:



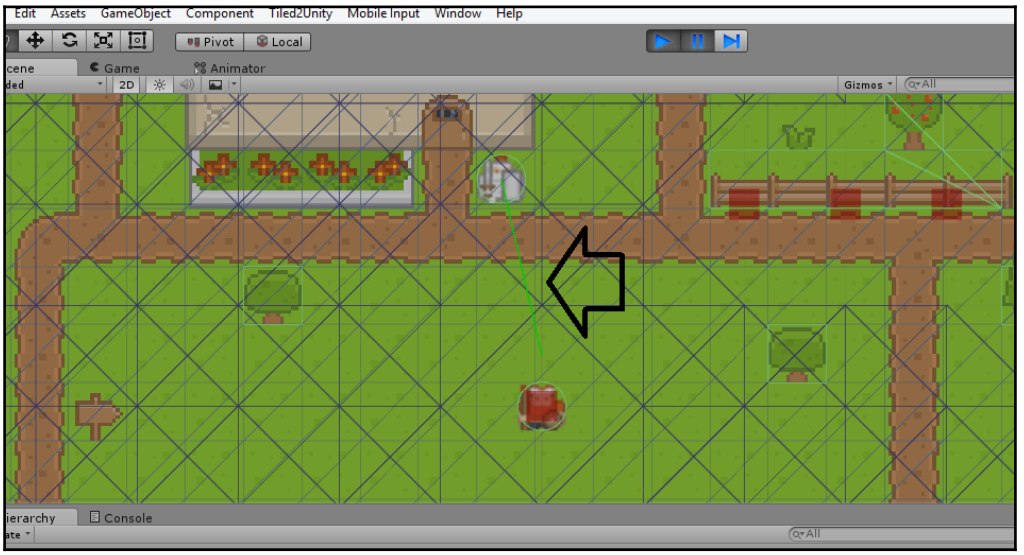
As for the other variables, it's better to leave their default values. However, feel free to change them if needed. Now, save the scene and click play to see what we have accomplished so far:

Final notes

As you can see, right after playing the scene, the soldier moves towards our player until he is close enough and then stops. You can also watch the console for details on the path generated from the `Pathfinder` class, as shown in the following image:



Additionally, if you have enabled gizmos, in the **Scene** view, you should be able to see green lines generated by the pathfinder class to indicate the path to the target object, as shown in the following image:



You can use this feature to understand how pathfinding works by moving the player object across the scene and looking at the gizmos indicating the generated path.

What you will also notice is that the soldier will move around all obstacles in the scene until it arrives at the `player` object:

If you are passionate or just want to learn more about AI in games, you can regularly check the following website, where you will also find advanced pathfinding techniques, which are still developing in the academic environment: `francescosapio.com`.

Summary

In this chapter, we walked through basic pathfinding techniques and the AStar Algorithm, by adding our very first AI character, which in turn added more life to our game. We discussed pathfinding, the AStar Algorithm, and how to apply AI in Unity.

Unluckily, this is it for our RPG. In fact, to develop an entire game with all the features would require several books. However, in these last two chapters we gave a good foundation for learning more.

In the next chapter, we will move onto our last project, where we will use all of the skills that we have learnt so far to create a strategy game.

7
Tower Defense Basics

For our last game, we will work on creating a 2D strategic game. In particular, we will create a **Tower Defense** game in which the player is able to place towers along the path where the enemies head toward the player's fortress. Since the entire process will take some time, we will spend the next three chapters creating this game.

In this chapter, we will go through the creation of the basic elements for the game. The following is what we will see in detail:

- Setting up the scene
- Creating bullets for the towers
- Implementing the towers
- Giving life to the enemies

Tower Defense games

In the previous chapters, we worked on both our Platformer game and the RPG. Here, instead, we will see how we can use Unity to create a Tower Defense game.

Usually, in these kinds of games, there is a road along which the players have to place towers. This is in order to stop a horde of terrible enemies that try relentlessly to reach the end of the path to apply some damage to the player.

Creating a Tower Defense game with many levels can be done relatively easily, once all the main logic behind the game has been implemented. In fact, due to the modularity of each component, creating a new map is very easy, and also the re-playability of the game is high, since in every level the player can decide to use different strategies.

In this chapter, we will see how to implement the basic elements, which we will use in further chapters to create our Tower Defense game.

Before jumping into the action, we need to get our assets ready to be used in the game.

Getting ready

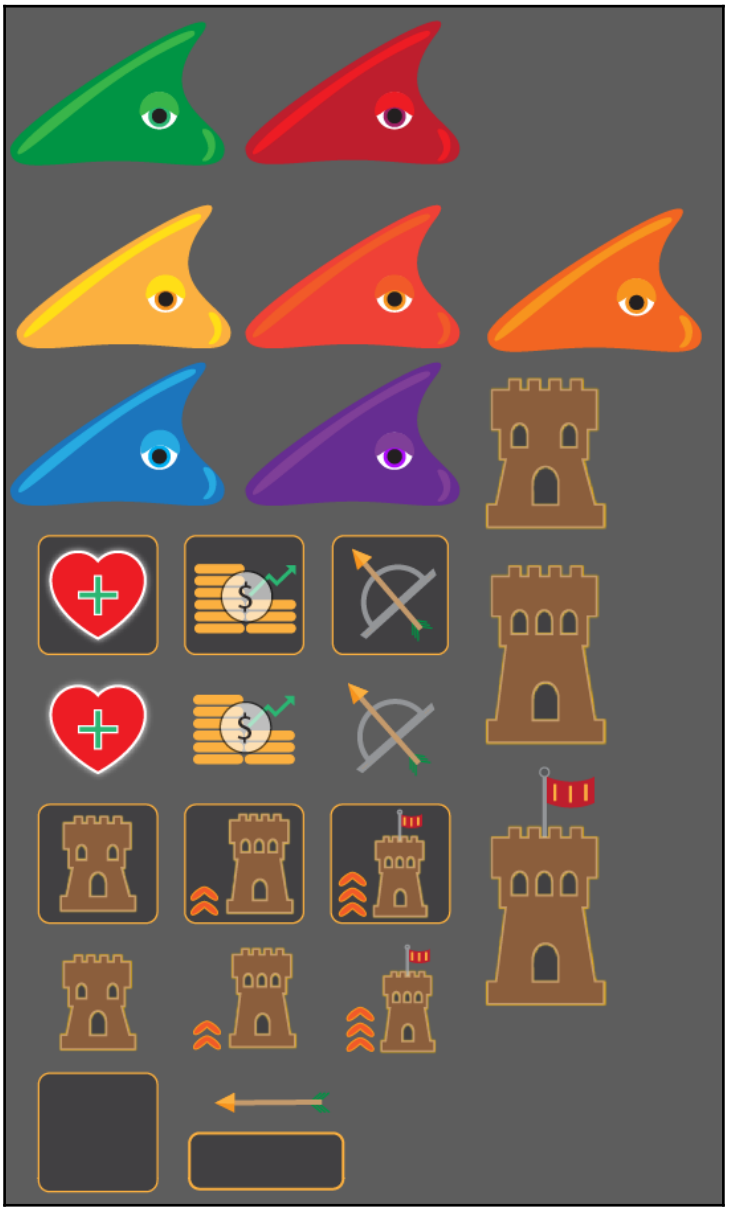
Let's open up Unity and create a new project. We can call it `Tower Defense`, and after we have checked the 2D setting, let's click on create.

We have seen how it's possible to quickly create a tiled map in *`Chapter 4, Level Design`*, so feel free to create your map in that way. However, in this chapter, we will use a free package, which we can download at the following link: `player26.com`.

The package, created by Lauren S. Ferro, includes all the basic assets to create our Tower Defense game, including a very nice map that perfectly fulfills our requirements. In particular, we will find:

- A map with a road for Tower Defense games
- A set of different slime enemies
- Three levels of upgraded towers
- Multiple icons for each object of the package
- An arrow as a bullet
- Other interesting graphics (that we are not going to use) such as decorations, trees, and so on

The following image can give us a better idea of which kind of graphics this package contains:



Once imported, we may want to change the settings of our graphics, such as setting the quality to `Truecolor`.

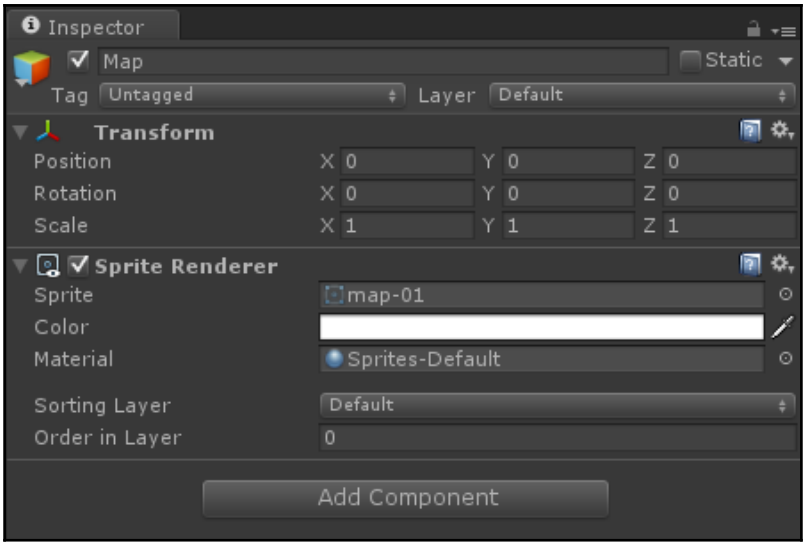
So, after we have imported all of them, let's start.

If our project is set to 3D, or if for some other reason the assets that we have imported are not set as Sprites, we have to do this in order to use them in this chapter. It is possible to change this setting by selecting them from the **Project** panel and changing the **Texture Type** to `Sprite (2D and UI)` in the **Inspector**.

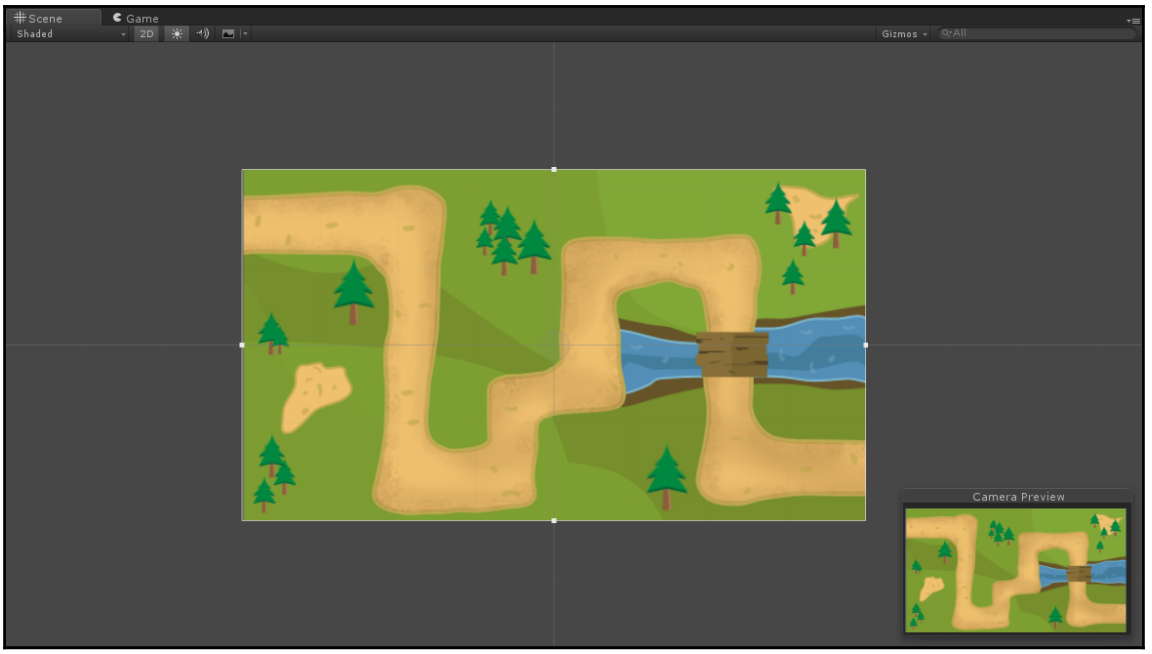
Setting up the scene and creating the map

The first thing to do is to create the map where the game will take place. This can be easily done by creating a new Sprite by right-clicking on the **Hierarchy** panel and then selecting `2D Object/Sprite`.

Now, we can assign the map in the package to the Sprite variable by dragging and dropping it on the variable. The name of the asset should be `map-01`. Moreover, we should set all the components of its position to 0, especially the *Z*-axis, and rename it, `Map`. Our **Inspector** should appear like this:



The next step is to set the camera. To do this, we need to select it and then change its `Size` variable to fit our map. In this case, we can set its value to `22.5`. As a result, the whole map is visible by the camera, as in the following image:



Keep in mind that some of the space could be needed to create the UI, which we will see in the next chapter. However, the position and the size of the camera can be changed later, when we will have a better idea of which kind of UI will fill that extra space.

Finally, if we have some other nice assets or particle effects, we can add them to the scene to decorate it even more. For instance, in the package, there is also a tree, or an extra texture that can be placed on top of the map to improve the road.

Now that we have the scene ready with the map, let's start to build the main elements for our strategic game, starting with the bullets.

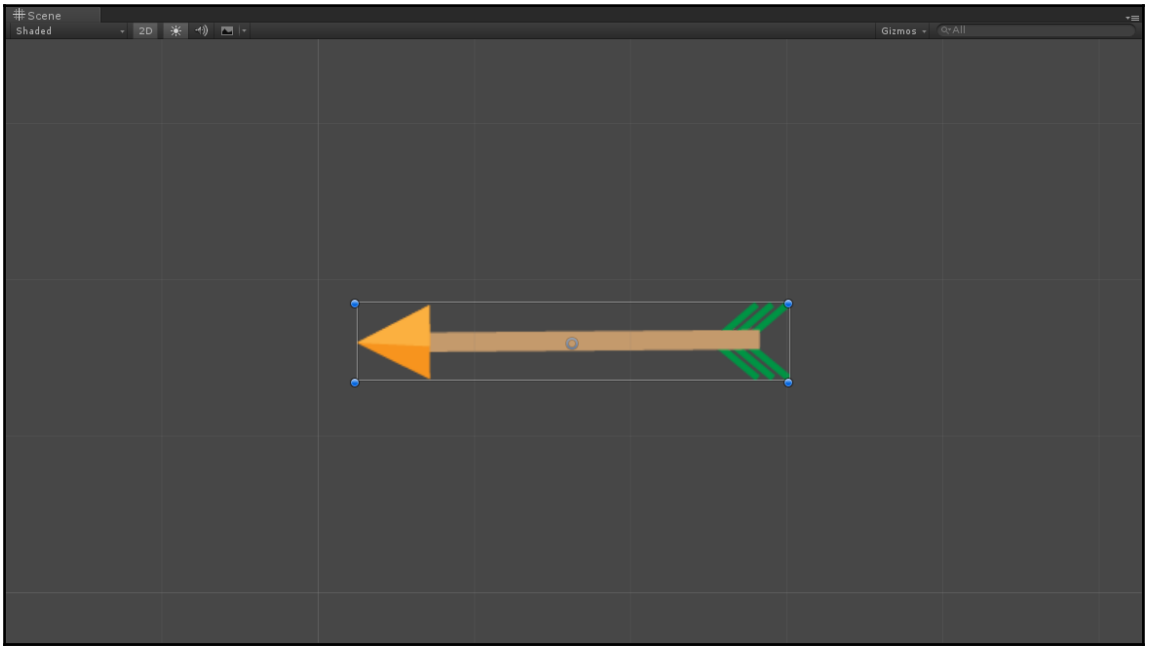
Bullets

In this section, we will learn how to create the bullets that our towers will shoot against the enemies.

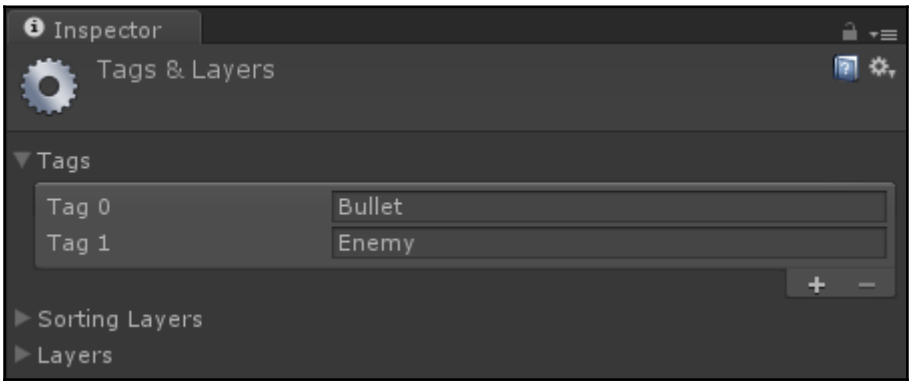
Creating the bullet prefab

Since these bullets are going to be thrown by the towers, we need to create a prefab that allows us to quickly instantiate one of these. In our game environment, the bullets will be arrows. Feel free to use your own graphics, if you have any.

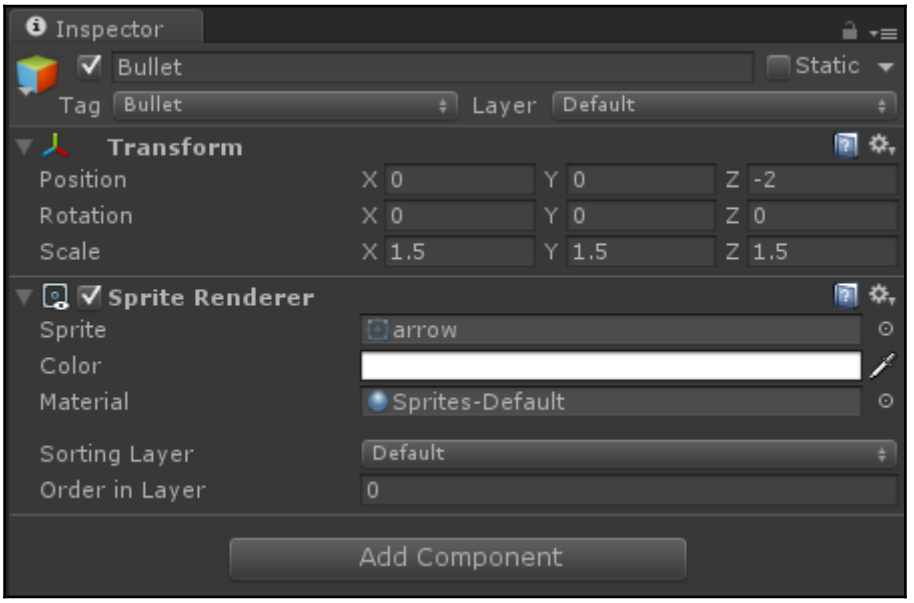
By right-clicking on the **Hierarchy** panel, we can select `2D Object/Sprite` in order to create a new `Sprite`. Of course, we need to assign the graphic of the arrow, which can be found in the graphic package, and adjust its scale to fit our game environment. In this case, we can set `1.5` to the whole scale vector. Furthermore, we need to set the *Z*-axis of the position to `-2`. In the end, the arrow should look like this:



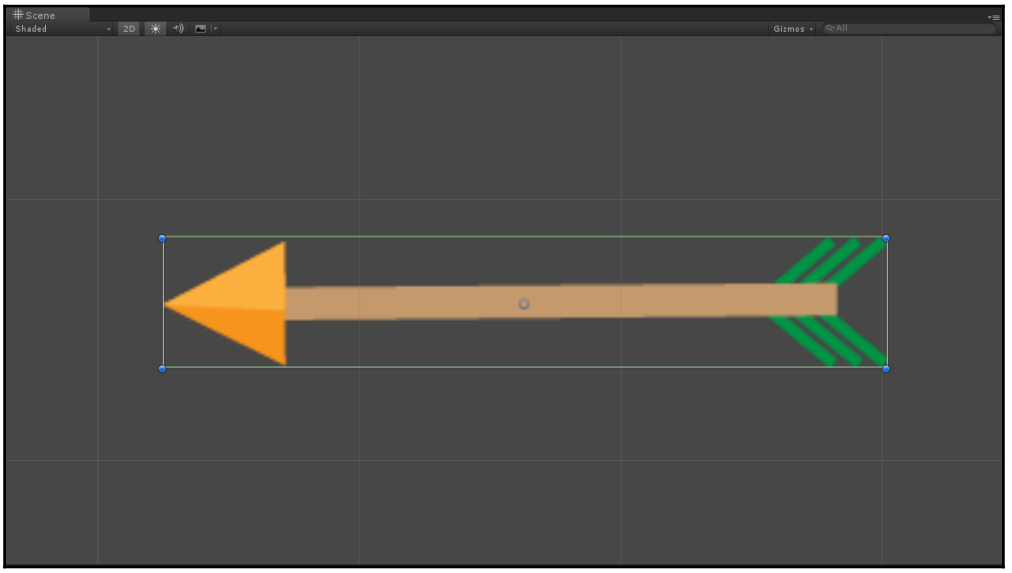
Next, we need to assign a tag to it. Therefore let's click, in the **Inspector**, on `Tag/New Tag`. Now the **Inspector** should display the **Tags and Layers** menu. Since we are interested in tags, expand the **Tags** menu and by clicking on the **+** we are able to add new tags. In particular, we should add both `Bullet` and `Enemy`. We will use the last one later on, when we create the enemy. In the end, our screen should look like this:



Now, come back to our bullet, and finally we can assign the right tag. Furthermore, it's good practice to change the name to `Bullet`. As a result, the **Inspector** should appear like this:

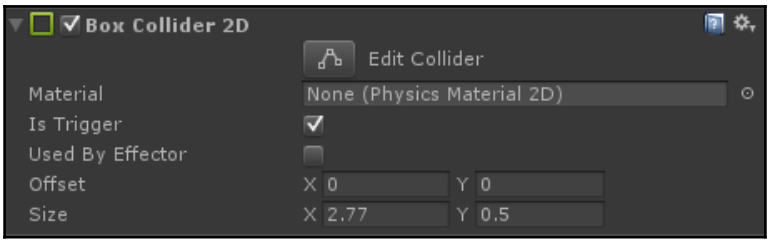


Since the bullet will collide with the enemies, it needs to have a collider attached to it. We can add this by navigating to **Add Component** | **Physics 2D** | **Box Collider 2D**. Of course, if we are using different graphics to the one in the package, we may need to adjust the collider in order to wrap the bullet. In this case, since the graphic is just a rectangle, it is automatically done correctly:

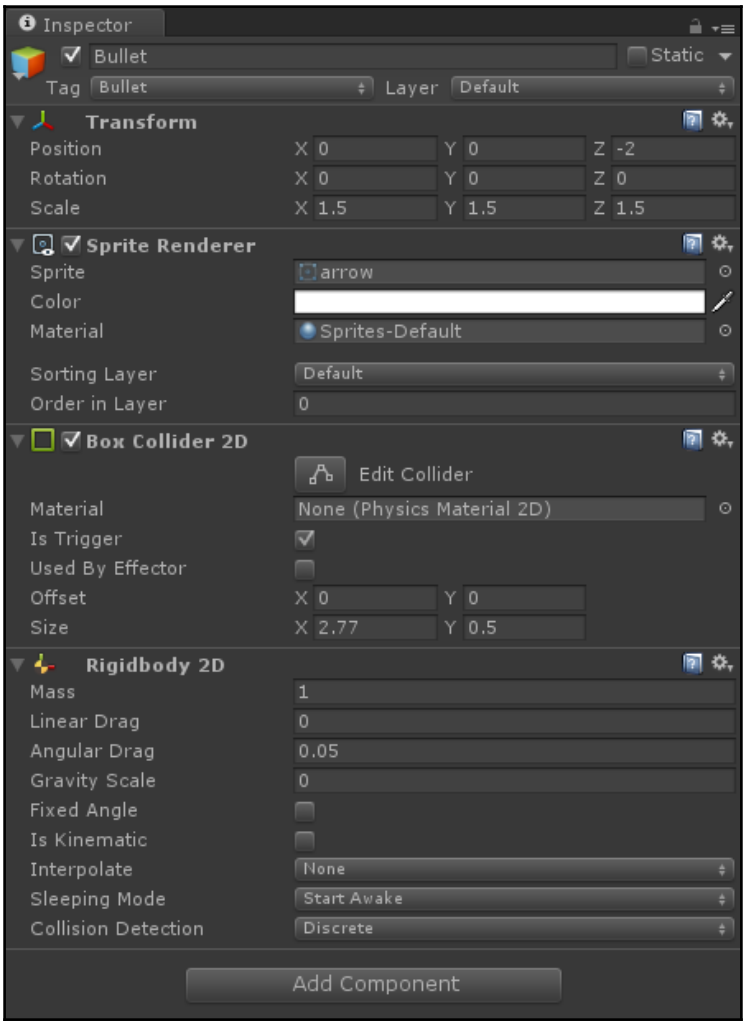


If we are using our own graphic for the bullet and it has a complex shape, we can also use another kind of collider, such as a `Circle` collider or a `Polygon` one. Later in the chapter, the enemy's collider can be easily done by using an `Edge` collider. However, for the sake of simplicity, we will use the `Box` collider 2D throughout the entire project.

Since we want the arrows to be detected from the enemies, inside the collider, we need to set the **Is Trigger** variable to true, as shown in the picture below:



Another component that we need to detect the collision with the enemies is a `Rigidbody2D`. We can add it by navigating to **Add Component** | **Physics 2D** | **Rigidbody 2D**. Furthermore, we need to set its **Gravity Scale** to zero, since our game is top view and we don't want to see our bullet fall off the map. We could also change the gravity in the **Physics** settings, but since we are not going to use it, just keep the **Gravity Scale** at 0. The whole Inspector should now look like the following:



Another interesting feature that can make our bullet/arrow more realistic is a Trailer Renderer. In this way, when it is thrown, we can easily implement a small tail that, by adjusting the setting properly, could be made to look like the air movement behind the arrow, typical of any comic/toon graphics.

Finally, we need to create a new Prefab by right-clicking on the **Project** panel and then selecting **Create** | **Prefab**. Drag the arrow inside it, and erase the old one from the scene.

Scripting the bullet

After having selected the prefab that we created in the previous section, let's add a new C# script to it by clicking on **Add Component** | **New Script**. The main aim of this script is to move the bullet along a direction, and this can be done in a few lines of code.

First, we need to create two variables. One is for the speed and the other one is for the direction.

Please note that the direction will be assigned dynamically from the tower when it shoots, whereas the value of the speed will be stored in the prefab and doesn't change at runtime, at least in this implementation. Of course, you can play with this speed value and add more lines to make it change so you have different kinds of bullets.

Therefore, we can write the following:

```
public float speed = 1f;
public Vector3 direction;
```

If our bullet cannot hit an enemy, we should prevent it from going on forever. We can achieve this by delaying the destruction of the bullet by 10 seconds. This means that after an amount of time, the bullet will be destroyed. Furthermore, we need to properly rotate the bullet toward the direction where it is heading, so the arrow will point toward that direction. To do this, we first normalize the direction, in case it isn't already, and then we calculate the angle. In doing this, we need to flip both the *X* and *Y*-axes, since our arrow is pointing left. Thus, let's write in the `Start()` function the following lines:

```
void Start () {
  direction = direction.normalized;
  float angle = Mathf.Atan2(-direction.y, -direction.x) * Mathf.Rad2Deg;
  transform.rotation = Quaternion.AngleAxis(angle, Vector3.forward);
  Destroy(gameObject, 10);
}
```

Finally, in the `Update()` function, we need to move the bullet in that direction:

```
void Update () {
  transform.position += direction * Time.deltaTime * speed;
}
```

And that's all for the bullet. So, after we have saved the prefab, since we added our script and we therefore need to update it, let's move to the next section to learn how to build the towers that will shoot these bullets.

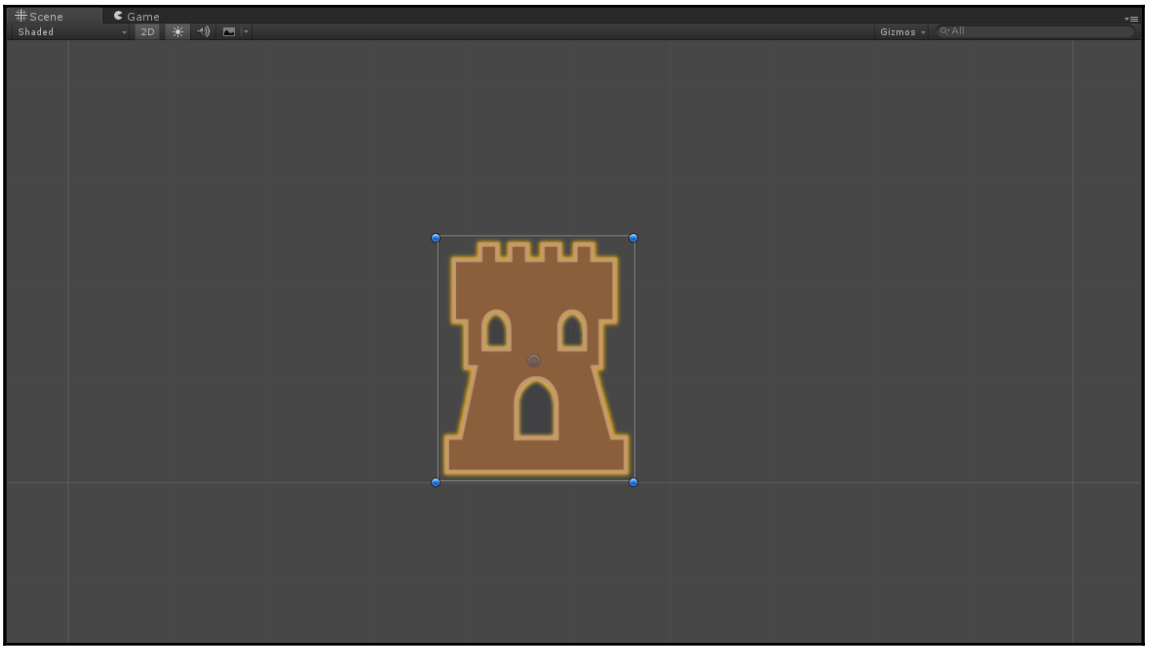
Towers

In this section, we will learn how to create the towers of our game. In fact, these are so important that they even give their name to this sub-genre of games.

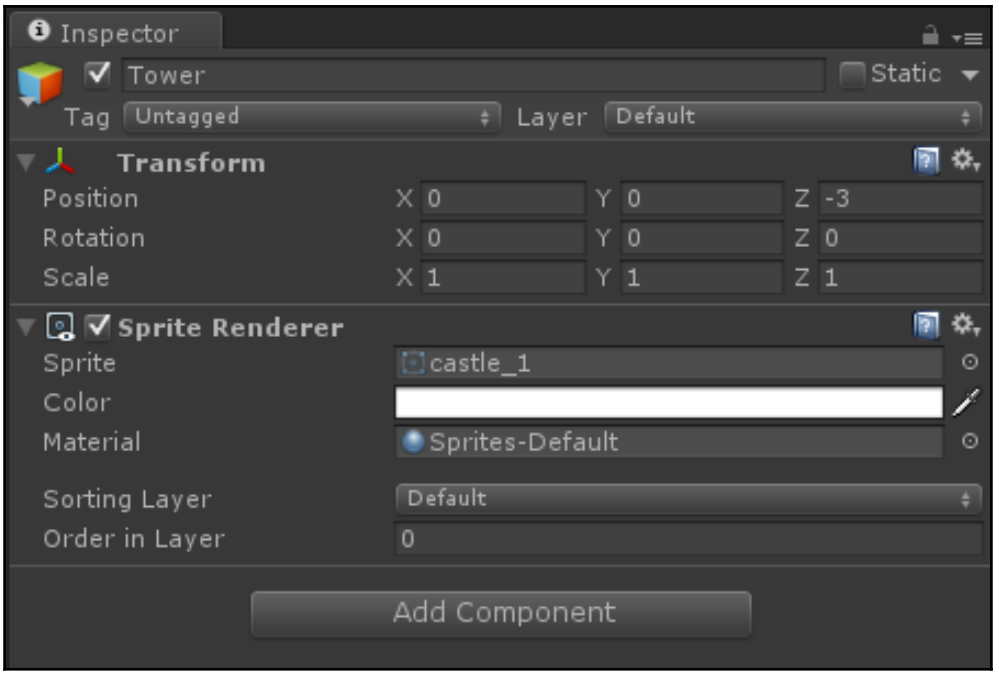
Creating the tower prefab

As we did for the bullet, the first thing to do is to create the prefab for our tower. Therefore, again we need to create a new `Sprite` by right-clicking on the **Hierarchy** panel and then clicking on **2D Object** | **Sprite**.

We have to assign the graphic to the tower, and it can be found in the package. Since there is more than one tower, because the player has the ability to upgrade the tower, we can choose the starting one. It can be found with the following filename: `castle_1`. This is what it looks like:



Probably, we will need to scale it to fit the map. By using the assets of this chapter, we can set all the scale vectors to `2`. Furthermore, we have to set the *Z*-axis of the position to `-3`. As usual, it's good practice to rename the `GameObject` to `Tower`, and thus, in the end, the Inspector should appear like the following:



Finally, we need to save the tower as a prefab. Therefore, let's create one by right-clicking on the **Project** panel and then selecting **Create** | **Prefab**. Drag the tower inside it, and erase the old one from the scene.

Scripting the towers

The next step is to give behaviors to our towers. In particular, they have to shoot at the enemies when they get close. There are different ways to do this and different strategies that the tower itself could choose, starting with which enemy it should target. Here, we are going to implement that the nearest enemy to the tower will be the one targeted.

Other examples that can inspire you to implement different behaviors for the towers are the `furthest enemy` from the tower, the `first enemy` with respect to the path or, also, the `last enemy`.

The first thing to do after creating a new script on the **Tower** prefab is to add some variables. We need three public variables to store, respectively, the bullet prefab that is used to instantiate bullets, the range radius of the tower, and the reload time. Furthermore, we need a fourth private variable to store the time elapsed since the last bullet shot in order to respect the firing rate of the tower. Therefore, we can write the following:

```
public float rangeRadius;
public float reloadTime;
public GameObject bulletPrefab;
private float elapsedTime;
```

In the `Update()` function, we need to first check whether the time that has elapsed is enough to shoot again and then reset the time. Otherwise, we just increment the variable:

```
void Update () {
  if(elapsedTime >= reloadTime){
    elapsedTime = 0;
    //Rest of the code
  }
  elapsedTime += Time.deltaTime;
}
```

Now, after resetting the `timeElapsed` variable, we verify whether there is some enemy in range of the tower. Actually, for now, we are searching for everything that has a collider:

```
      Collider2D[] hitColliders =
Physics2D.OverlapCircleAll(transform.position, rangeRadius);
      if(hitColliders.Length != 0){
        //Rest of the code
      }
```

If we detect different colliders, we need to check which one is an enemy by using tags and then to find which one of these (if there is more than one) is the closest to the tower. This can be done by looping over all of them and taking the minimum distance among all of them, after having verified that they are enemies:

```
        float min = int.MaxValue;
        int index = -1;

        for(int i=0; i<hitColliders.Length; i++){
          if(hitColliders[i].tag == "Enemy"){
            float distance =
Vector2.Distance(hitColliders[i].transform.position,transform.position);
            if (distance < min){
              index = i;
              min = distance;
            }
          }
        }
```

Then, we can check to see if there is an enemy among all of our collisions. If so, we instantiate a bullet and give to it the direction of the target we found:

```
        if(index == -1)
          return;
        Transform target = hitColliders[index].transform;
        Vector2 direction = (target.position -
transform.position).normalized;
        //Create Bullet
        GameObject bullet = GameObject.Instantiate(bulletPrefab,
transform.position, Quaternion.identity) as GameObject;
        bullet.GetComponent<BulletScript>().direction = direction;
```

Now, we can save the script.

One last thing to do is to drag the prefab of the bullet in the `bulletPrefab` variable and set the other variables as we need for the game. For instance, we can set the `reloadTime` to `2` and the `rangeRadius` to `10`. Finally, save the tower prefab before we remove it from the scene.

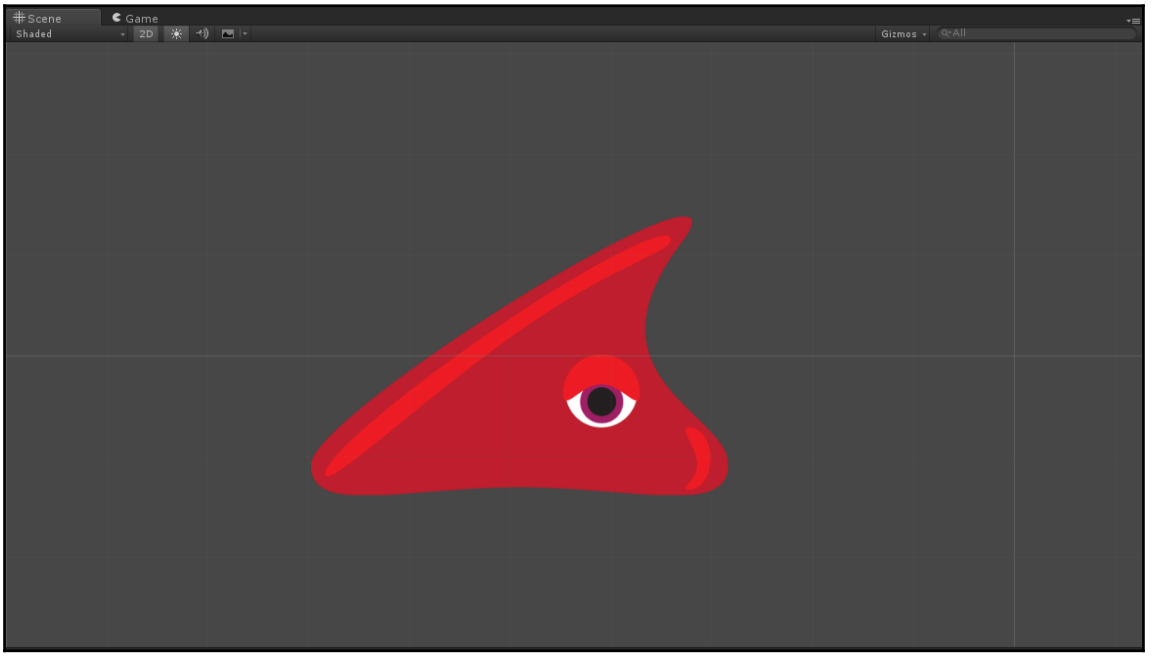
Enemies

In this section, we will learn how to create the enemies for our Tower Defense game.

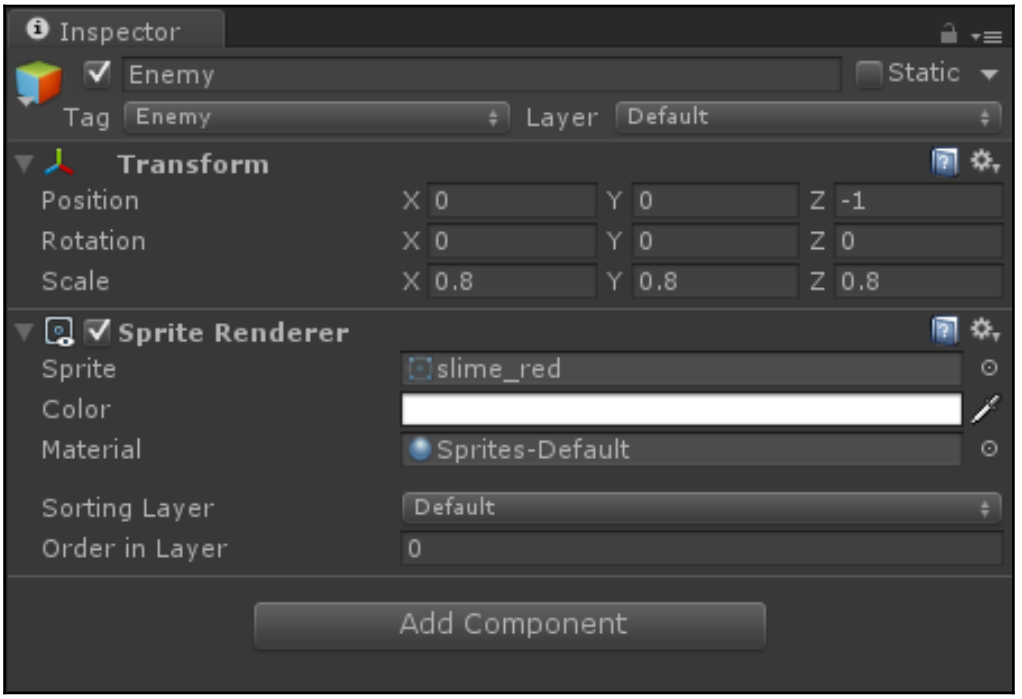
Creating the enemy prefab

Since we will see how the game manager will spawn a lot of enemies later, in *`Chapter 9`*, *Finishing the Game*, we need to create a prefab to store all the data for one enemy, including its graphics, components, and scripts.

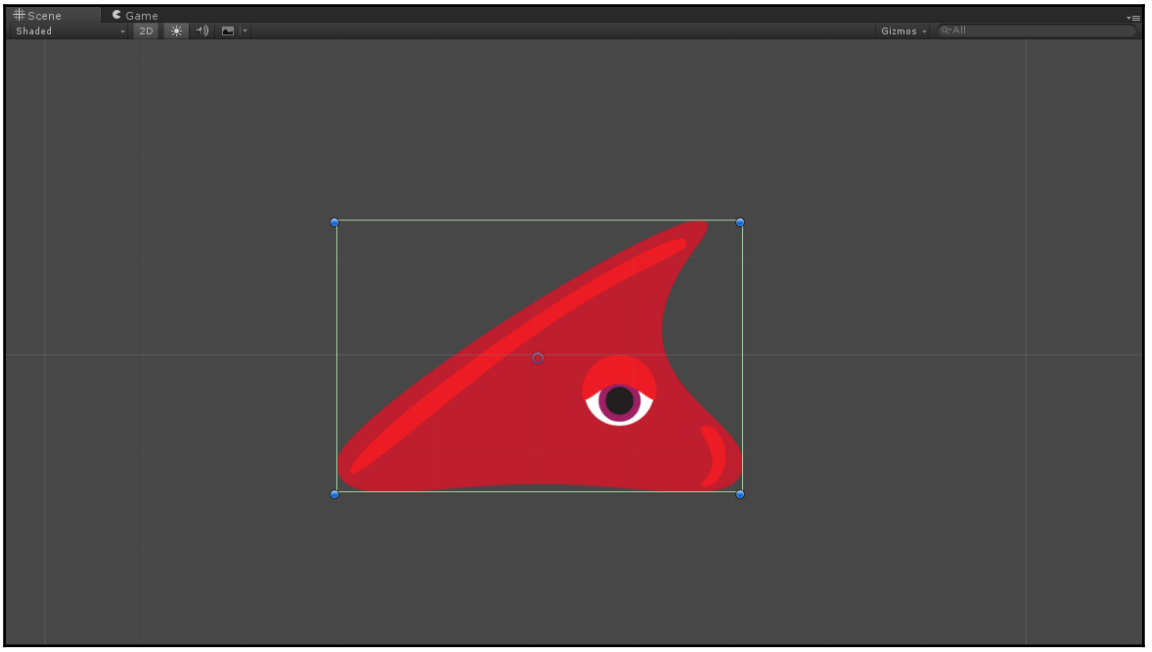
By right-clicking on the **Hierarchy** panel, we can select **2D Object/Sprite** in order to create a new `Sprite`. Of course, we need to assign the graphic and adjust its dimensions to fit our game environment. In this case, we can chose the `slime_red` as our enemy. Like the previous components, towers and bullets, it also needs to be scaled. In this case, we can set the whole scale vector to `0.8`. Furthermore, we should set the value of the *Z*-axis, for the position, to `-1`. Once we have done this, it should look like this:



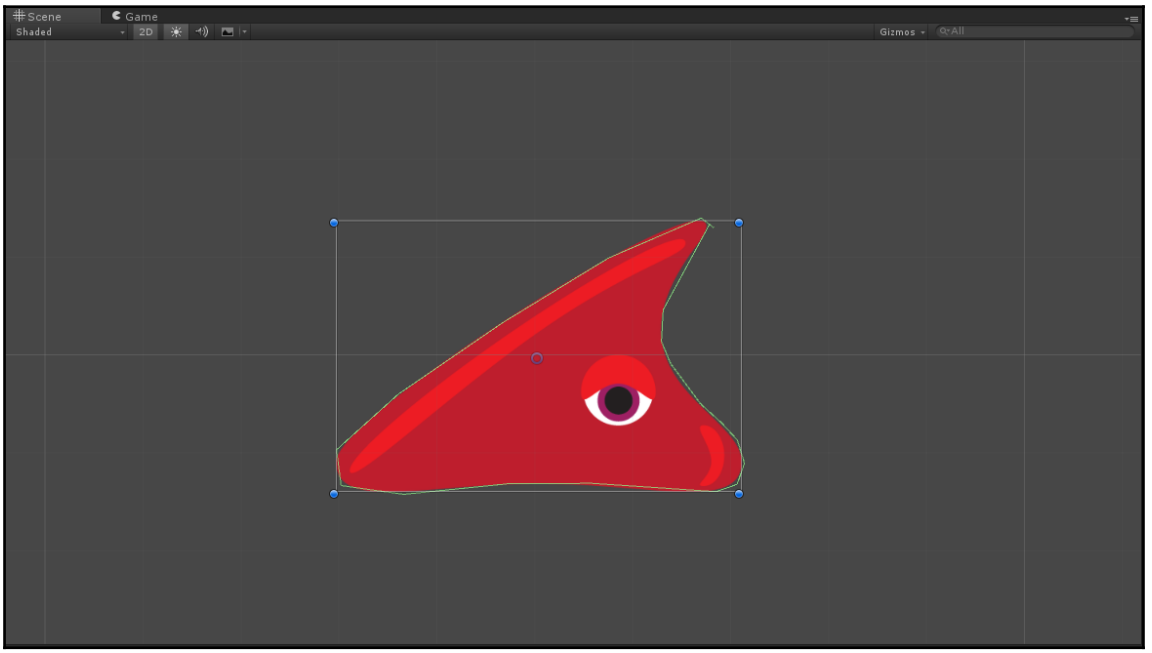
In order to be detected by the towers, our enemy should have the `enemy` tag. Since we have already created this tag in the bullet section, it can easily be assigned to our enemy. As usual, it's always a good practice to rename the `GameObject` as `Enemy`. The **Inspector** should look like the following:



Then, we need to add a **Box Collider 2D** by clicking on **Add Component** | **Physics 2D** | **Box Collider 2D** and setting the size variable to get the box to fit our enemy as closely as possible. In the case of our graphic, this is done automatically, since the slime is almost a rectangle. In the end, we should see something like the following image:



However, we should always consider other kinds of colliders if we want to achieve more precise behaviors. For instance, by using an `Edge` collider (you can find this by clicking on **Add Component** | **Physics 2D** | **Edge Collider 2D**), we can achieve something like this:



In addition, besides deciding on the type of collider that we have chosen, we need to set **Is Trigger** to true, since we are going to use this collider as a trigger to detect bullets (there's more information about this in the next section).

Next, we can create a new **Prefab** by right-clicking on the **Project** panel and then **Create** | **Prefab**. Finally, drag the enemy inside it and erase it from the scene.

Scripting the enemies

Now it's time to script our enemy. Let's create a new script by right-clicking on the **Project** panel and then selecting **Create | C# Script** (or also directly on the enemy prefab) and renaming it `EnemyScript`. Finally, double-click on the file to open it.

In order to make the slime alive, we need to create a script that it is able to make the enemy do the following:

- Move along the designed path on the map
- Damage the player's fortress if he reaches the end (we will see this in `Chapter 9`, *Finishing the Tower Defense Game*)
- Detect whether a bullet from the player's towers hit him

Moving along the designed path

To achieve this behavior, we are going to use a technique called *waypoints*. This is very useful if the enemy has to follow a set of designed paths, as in this case. In contrast to the previous chapter, where we needed an online pathfinding algorithm to decide where the enemy should move next, here, since the path is fixed, the waypoints technique is both easier to implement and also faster from a computational point of view.

In its basic implementation, the waypoint technique consists of storing all the key *waypoints* of the path, and makes the enemy move along them.

In more complex implementations, waypoints can be connected in different ways and these connections can also be created automatically by letting the waypoints *find* each other. Furthermore, they can also contain other information, such as which is the closest waypoint to the player. Here, the enemy can *ask* the waypoints where to head toward in order to find the player without running a complete pathfinding algorithm on the map itself.

We need to create four variables. The first is to specify the speed of this particular enemy, and two are needed to store, respectively, the path to follow and an internal counter to know which piece of the path the enemy is traversing. The last variable is just a constant threshold to detect whether an enemy has reached a waypoint. It is needed only for numerical robustness, since the distance from the waypoint will never be exactly zero.

Therefore, let's start to add the following variables:

```
public float speed = 1f;
private Vector3[] waypoints;
private int counter = 0;
private const float changeDist = 0.001f;
```

Now, in the `Update()` function, we should implement this movement. The first thing to do is to check whether the counter has reached the last waypoint, which means that the enemy has reached the player's fortress and we should apply damage to it. But, since we will see how to create the game manager for this game in the last chapter, for now we can just destroy the enemy with the following lines:

```
if(counter==waypoints.Length){
  Destroy(gameObject);
  return;
}
```

Then, we need to calculate the distance to the next waypoint where the enemy is heading. If the distance is less than the `changeDist` variable, we need to update the counter, which means changing the waypoint. Otherwise, just move the enemy closer to the next waypoint according to its speed:

```
else{
      float dist = Vector3.Distance(transform.position,
waypoints[counter]);
      if(dist < changeDist){
        counter++;
      }else{
        float step = speed * Time.deltaTime;
        transform.position = Vector3.MoveTowards(transform.position,
waypoints[counter], step);
      }
    }
```

We have used the `Vector3.MoveTowards()` function that allows us to move a `GameObject` toward another one, specifying a step. More information about this can be found in the official documentation on Unity. Here is the link:

`http://docs.unity3d.com/ScriptReference/Vector3.MoveTowards.html`.

Now we have finished implementing the movement of our little slimes; however, we still need to set the waypoints. We will do this in the last chapter of this book.

Detecting towers' bullets

While our enemy is trying to head toward the player's fortress to destroy it, at the same time he is being shot with bullets by the towers. Therefore, in some way, we have to detect when a bullet hits our enemy. In order to do this, we can use the collision system already implemented in Unity. In this case, we should use the 2DCollision since our game is in 2D. In particular, we are going to implement an OnTriggerEnter2D() function. More can be found here:
`http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerEnter2D.html`.

First, we need to check whether the object that hit the enemy is really a bullet. This can be done by checking whether the bullet has the `Bullet` tag. If so, we destroy both the enemy and the bullet in the following way:

```
void OnTriggerEnter2D(Collider2D other) {
  if(other.tag == "Bullet"){
    Destroy(other.gameObject);
    Destroy(gameObject);
  }
}
```

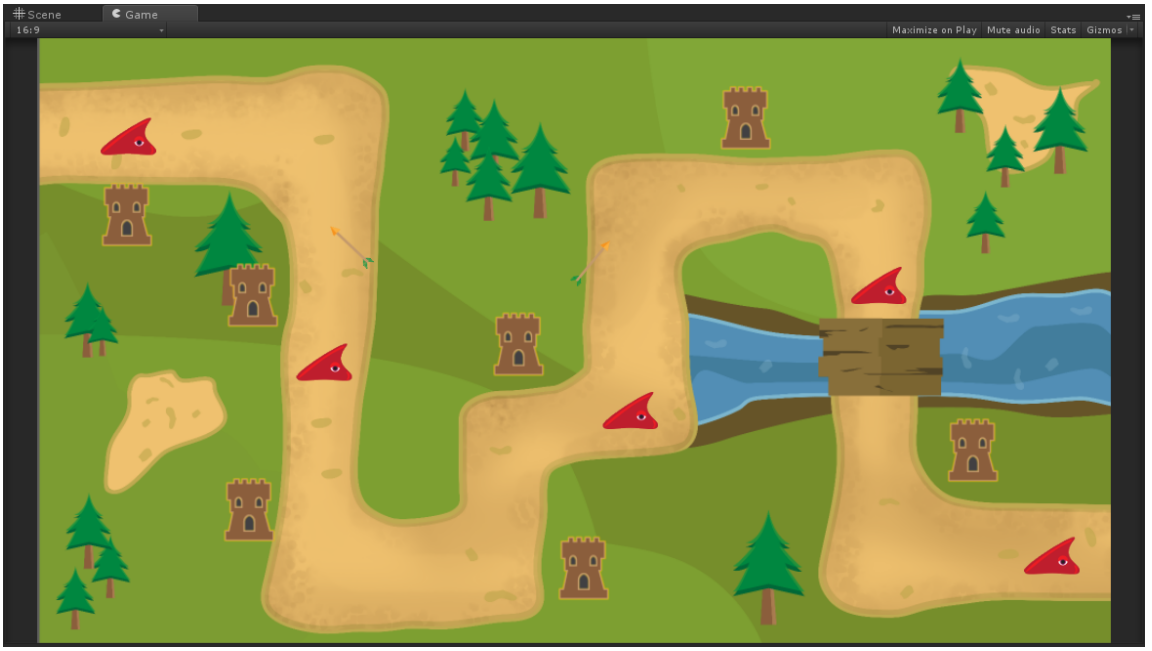
As a result, every time a bullet hits the enemy, both will be destroyed.

Finally, we have finished scripting our enemy. Now, save the script and attach it to the enemy prefab.

Summary

In this chapter, we started the Tower Defense game, designed the game level, and created the prefabs. Inside these latter ones, we already have most of the logic of our game implemented in the scripts attached to the prefabs.

Before progressing further, we should take some time to put all the pieces together, to have an overview of how the game will appear. It should look something like the following:



So, let's move into the next chapter, where we will integrate the UI into our game!

8
User Interface for the Tower Defense Game

In Tower Defense games, the **User Interface** (**UI**) often provides a way for the player to interact with the game. Through the UI, it is possible to build, sell, or upgrade towers. Furthermore, the UI is also important to visualize stats such as money and lives.

Since this is a big topic, we will not have enough time to cover everything here. However, we will see all the basics we need to implement our UI by recapping and extending the ideas seen in `Chapter 4`, *Level Design*. Some suggestions on how to implement it will be given later in the chapter as well.

Lastly, you should definitely consider buying a book that is specifically about UIs. For instance, the *Unity UI Cookbook, Packt Publishing*, has a perfect set of recipes ready to use. There you will find all the concepts treated here in more depth, and much more.

In this chapter, we will learn how to implement the UI for our Tower Defense game by looking specifically at:

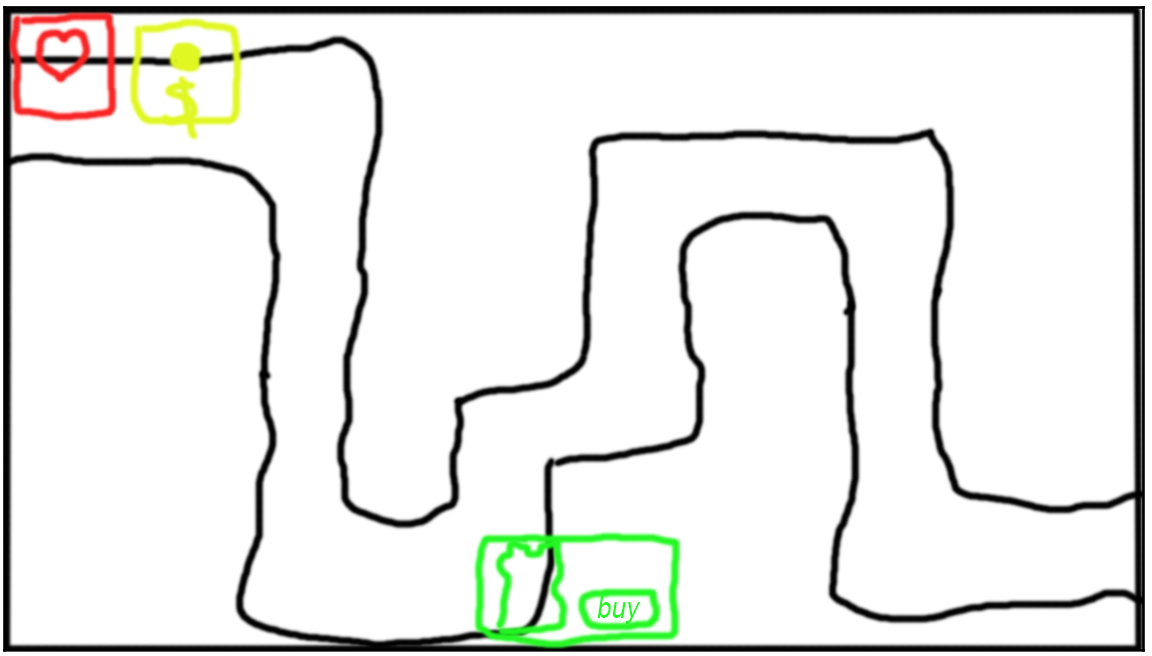
- Creating a lives counter
- Implementing a money system
- The tower seller
- Upgrading the tower

Getting ready

We will use the graphics from the same package as the previous chapter to build our UI too. Therefore, be sure to have them imported, and as Sprites, in order to use them in the UI.

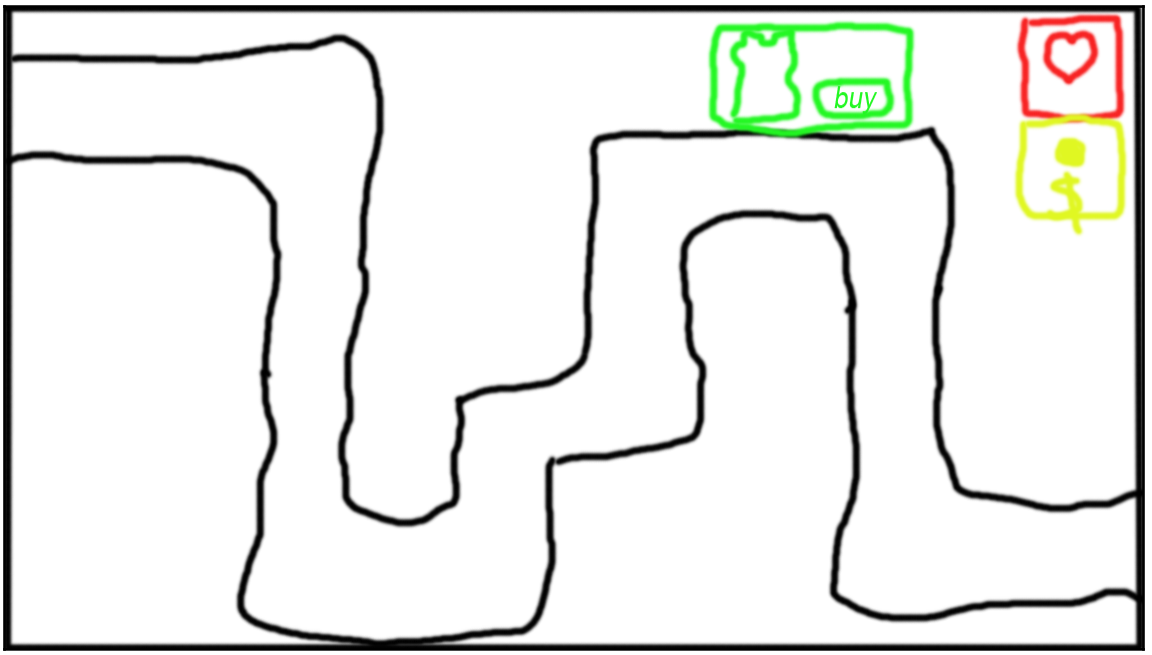
Designing the UI

A very important step in game development is to design the UI. This doesn't mean to just create the graphics, but to decide where and how it will appear on the game screen. For instance, take a look at the following design:



As you can see, the design could be great for some games, but it doesn't fit in our game. Because of the map, we should definitely avoid placing some of the components along the path that our enemies will follow.

A much better design is the following:



We will actually implement this one, but feel free to design your UI according to your own game. As you can see, the UI components do not overlap with the path.

There are some other elements that should be taken into account. For instance, a menu can appear and disappear, like a pop-up menu. This is something that is done to improve the design of our UI. However, these go beyond the scope of this chapter. Therefore, the topics can be referred to in more specific books, such as the one mentioned at the beginning of the chapter.
Another suggestion is to create a UI image with the background, the one with the path, and enlarge it to the all canvas. By doing this, we will be able to immediately see how our UI feels with respect to our level. However, once the UI is finished, we should remember to remove it.

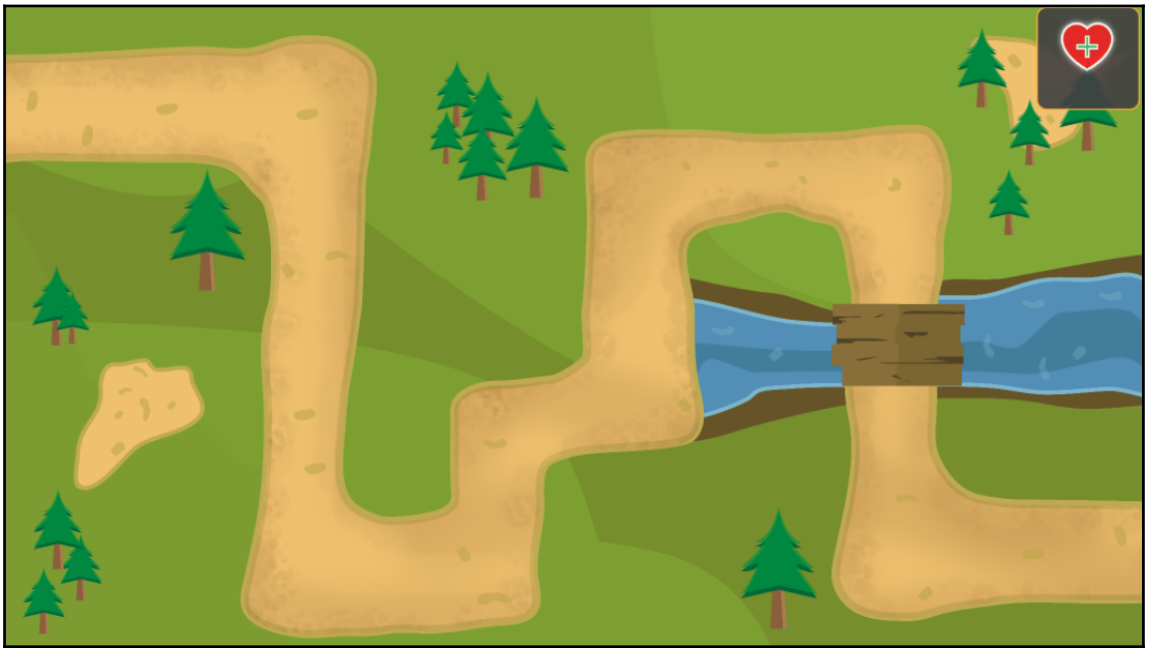
Creating a lives counter

The first thing to create is a lives counter, in order to keep track of the number of lives of the player. In fact, the goal of the player is to not allow his or her life to reach zero.

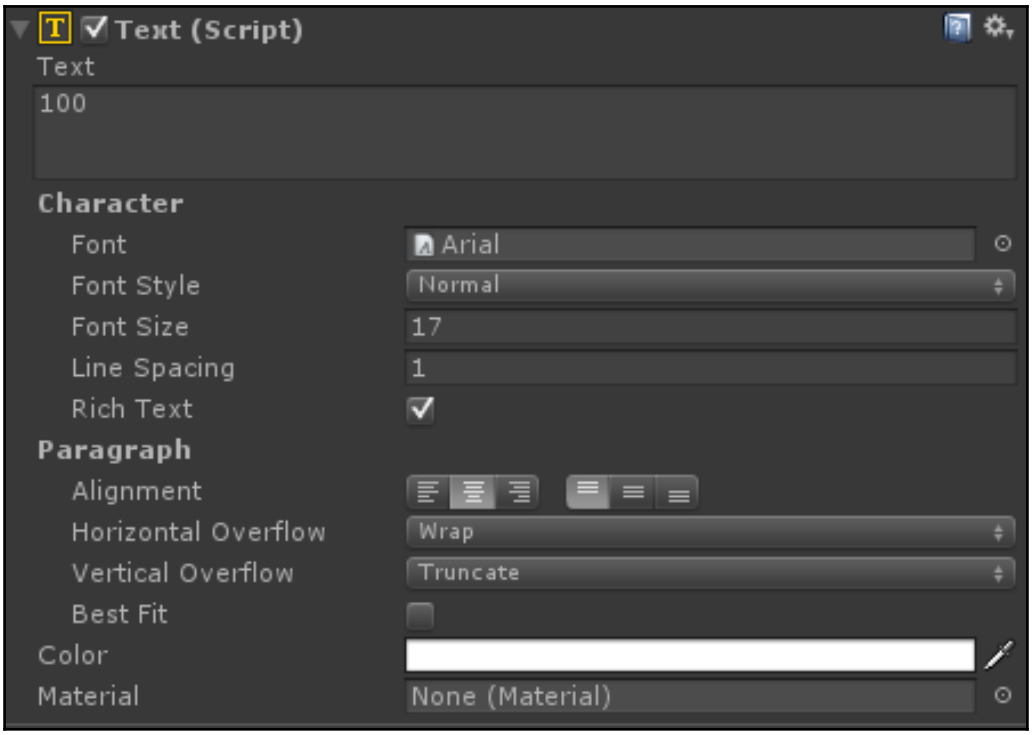
What we are going to do here is to create a more stable and flexible framework than the lives counter created in `Chapter 4`, *Level Design*. This is because we need a solution that can be scalable and easily extended if we are planning to create a great Tower Defense game.

Creating and placing the lives counter

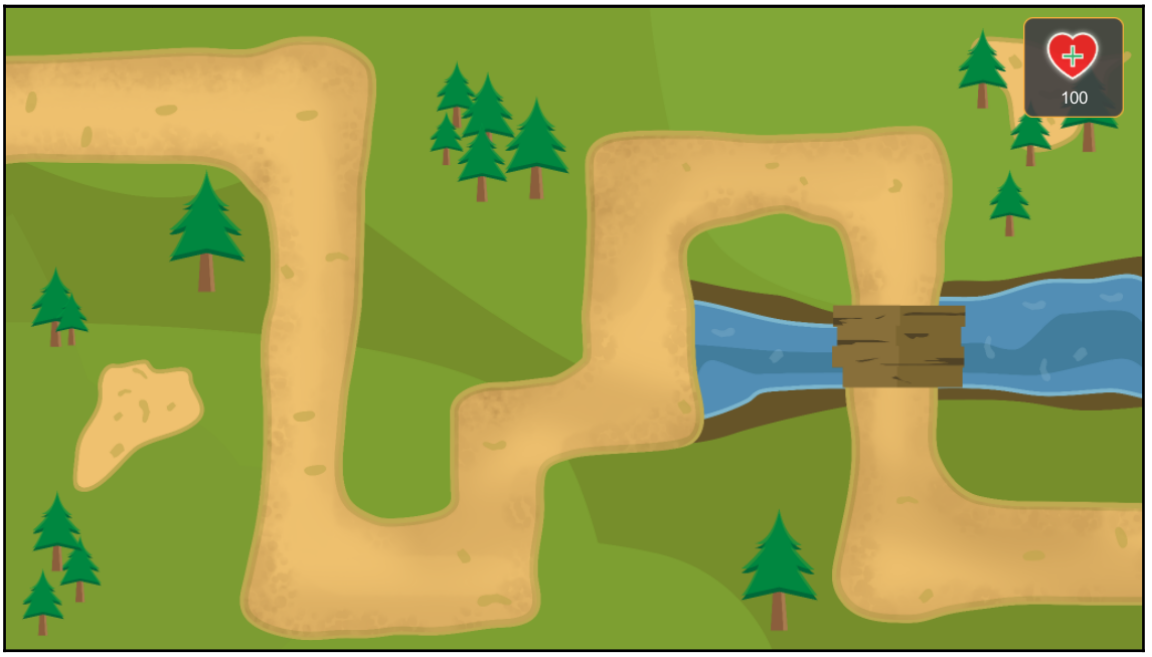
Let's start by creating a new image, by right-clicking on the **Hierarchy** panel and then **UI/Image**. We should also rename it `LivesCounter` and assign to the **Source Image** the `health_square` image in our package. We may want to press the **Set Native Size** button and later scale it down to fit the screen. Also, we can make it just a tiny bit transparent, by reducing the alpha channel of the color variable, let's say to `232`. Finally, we can drag and drop it in the top-right corner, as shown in the following image:



Now, we need to add some text to our counter, so the player can understand how many lives are remaining. Right-click on `LivesCounter` and then **UI/Text**. Also, rename it `LivesCounterText`. We need to change its setting as shown in the following image:



In particular, we have set the color variable to white and the size to `17`. In this way, it has the right dimensions and color to fit in our design for the UI. As text, we used a placeholder with just the number `100`. At the end, we have to place it just under the heart, but still inside the box, as in the following image:



It looks nice so far, but it won't change if we press play. We still need to add a script on it. This will handle the interaction with the counter, by updating it automatically when the number of lives changes. To create the script, select `LivesCounter` and, in the **Inspector**, navigate to **Add Component** | **New Script**. Name it `LivesCounterScript`, and then click on **Create and Add**.

Scripting the lives counter

Let's open the script by double-clicking on it. In order to be able to use the UI classes, we need to import a namespace. This can be done by adding the following line at the beginning of our code:

```
using UnityEngine.UI;
```

Now that we can also use the `UI` classes, we need three variables. One is a public variable that allows us to decide the maximum number of lives. The other two are private, to keep track of the `uiText`, in order to write the number of lives, and the number of lives itself:

```
private Text uiText;
public int maxLives;
private int lives;
```

Since the number of lives is a private variable and we need to retrieve its value, we need to also implement a get function, like the following one:

```
public int getLives(){
    return lives;
}
```

Next, we have to set a couple of variables in our `Start()` function, in particular, the reference to the `uiText`, by using the `GetComponent()` function and setting the number of lives equal to the maximum. Finally, we call a function to update the counter graphic, but we will see it in a few steps:

```
void Start () {
      lives = maxLives;
      uiText = this.GetComponentInChildren<Text>();
      updateLivesCounter();
}
```

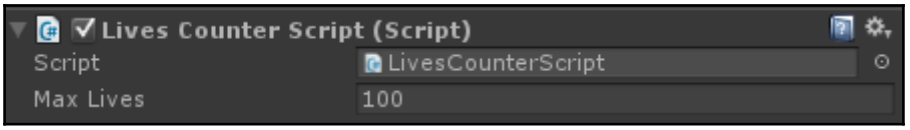
Then, we need to expose a public method that the **Game Manager**, which we will build in the next chapter. In this method, it is possible to reduce the number of lives and check whether it has reached zero. By doing this, we need to keep in mind that the function will return true if the player has no lives. Of course, we also need to update the graphic and keep the number of lives to zero as minimum, as robustness of our code:

```
public bool loseLife(int damage){
    lives -= damage;
    if (lives > 0){
        updateLivesCounter();
        return false;
    }
    lives = 0;
    updateLivesCounter();
    return true;
}
```

Finally, we can write the update function that we have called in the previous function. It just updates the graphic of the `uiText` by using the current number of lives:

```
private void updateLivesCounter(){
    uiText.text = lives.ToString();
}
```

After we have saved the script, it is ready to be used. Also, remember to assign a maximum number of lives. In this example, we will set it to `100`. Our script will be like the following:

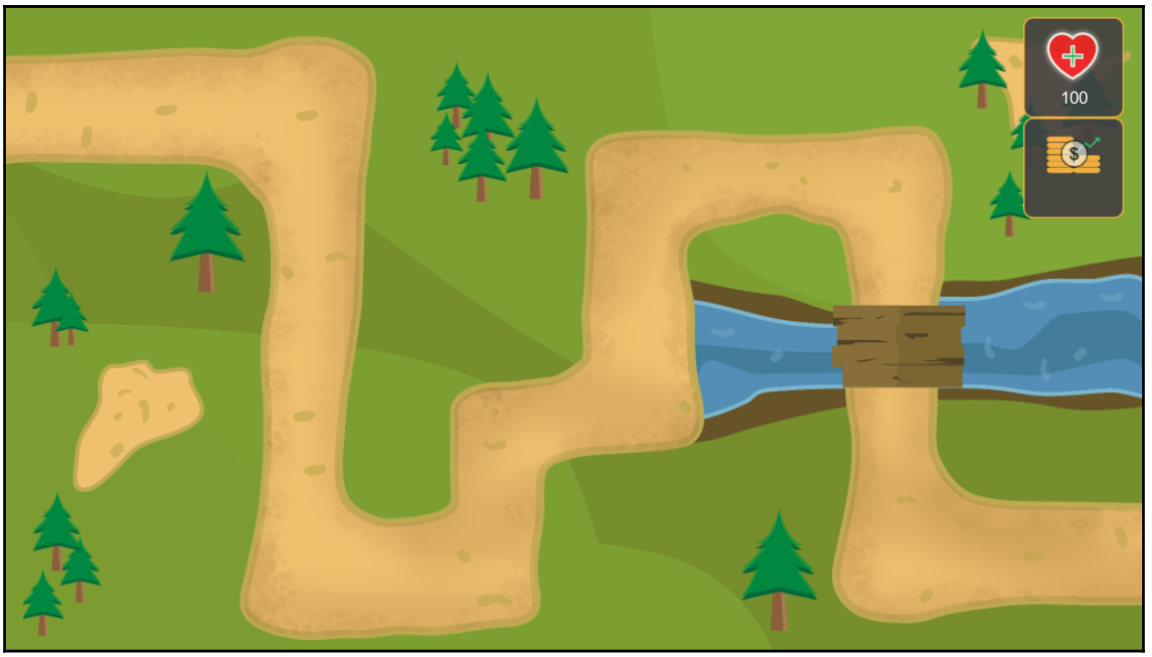


Implementing a money system

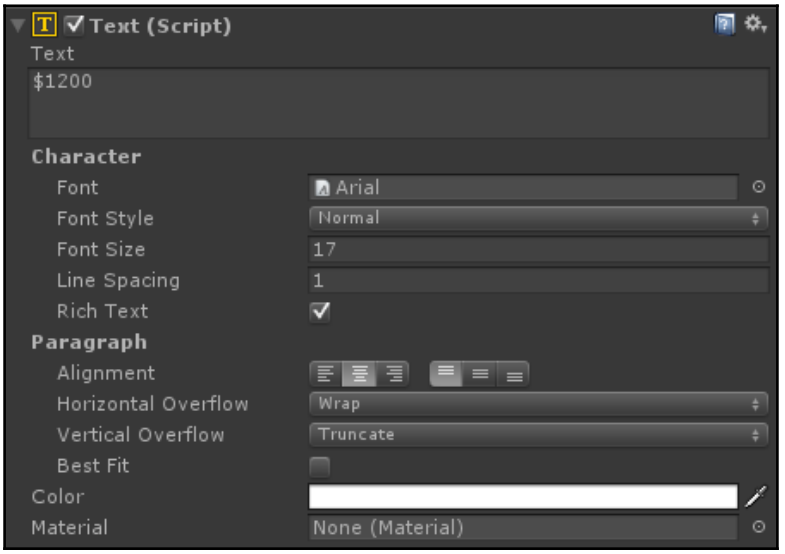
The player will earn money by killing enemies, and he is able to spend it by buying new towers to defend his fortress. By doing this, we need a game element that is able to handle a money system, in which the amount of money can increase and decrease according to the player's actions. This section will teach us how to create a money system and integrate it in the UI.

Creating and placing the money counter

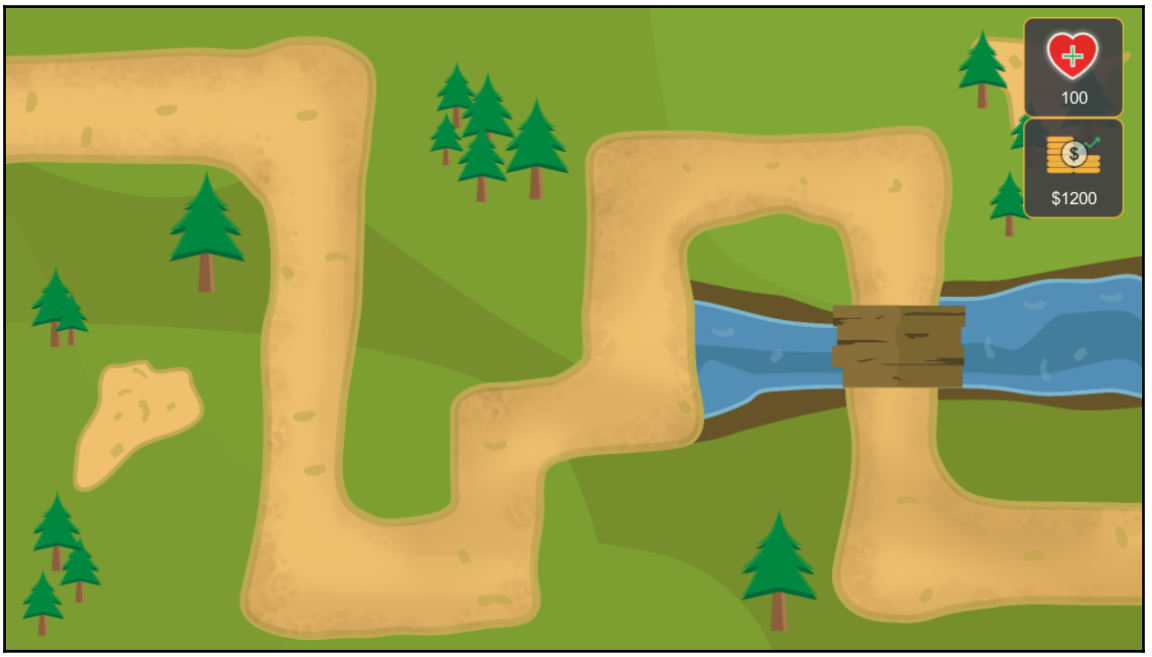
As we previously did in the Lives Counter, we need to create the UI structure. So, let's create a new UI Image by right-clicking on the Hierarchy panel and then **UI/Image**. Rename it `MoneyCounter` and assign to the **Source Image** the `currency_square` image of our package. Also, we should press the **Set Native Size** button and then scale it down. As before, make it a tiny bit transparent by setting the alpha channel of the color variable to `232`. Finally, place it just under the `LivesCounter`, as in the following image:



Similarly to the `LivesCounter`, let's create a `uiText` and rename it `MoneyCounterText`. Its settings are the same as the `LivesCounterText`, except the text placeholder:



As a result, our scene will be like this:



Again, it looks nice, but it won't change if we press play. We still need to add a script on it. This will handle the interaction with all the rest of the game, by updating the amount of money. To create the script, select `MoneyCounter` and, in the Inspector, navigate to **Add Component** | **New Script**. Name it `MoneyCounterScript`, and then click on **Create and Add**. Double-click to open it.

Scripting the money counter

The money counter works similarly to the lives counter we already built in the previous section. Since we are going to use the `UI` classes once more, we need to add the following at the beginning of our script:

```
using UnityEngine.UI;
```

Again, we need two public variables to store the reference to the `uiText` and the money of the player. However, we don't need a maximum, since the player is supposed to increase his money as much as he can:

```
private Text uiText;
private int money = 0;
```

In the `Start()` function, just get the reference to the `uiText` and update the graphic through a function. We can write it in a few steps:

```
void Start () {
      uiText = this.GetComponentInChildren<Text>();
      updateMoneyCounter();
}
```

Now, we need a generic function to increase or decrease money by a certain amount. We have to take into account that it is not possible to have a negative amount of money, and the graphic needs to be updated:

```
public void changeMoney(int ammount){
    money += ammount;
    if (money < 0){
        money = 0;
    }
    updateMoneyCounter();
}
```

Furthermore, a function to get the amount of money is useful for many reasons, such as checking to see whether the player can afford a tower. We need this function, since the amount of money is a private variable, and we don't want to make it public since it can only change with the `changeMoney()` function, which updates the graphic too:

```
public int getMoney() {
    return money;
}
```

Finally, the function to update the graphic, which is very similar to the one used for the lives counter:

```
private void updateMoneyCounter(){
    uiText.text = "$" + money.ToString();
}
```

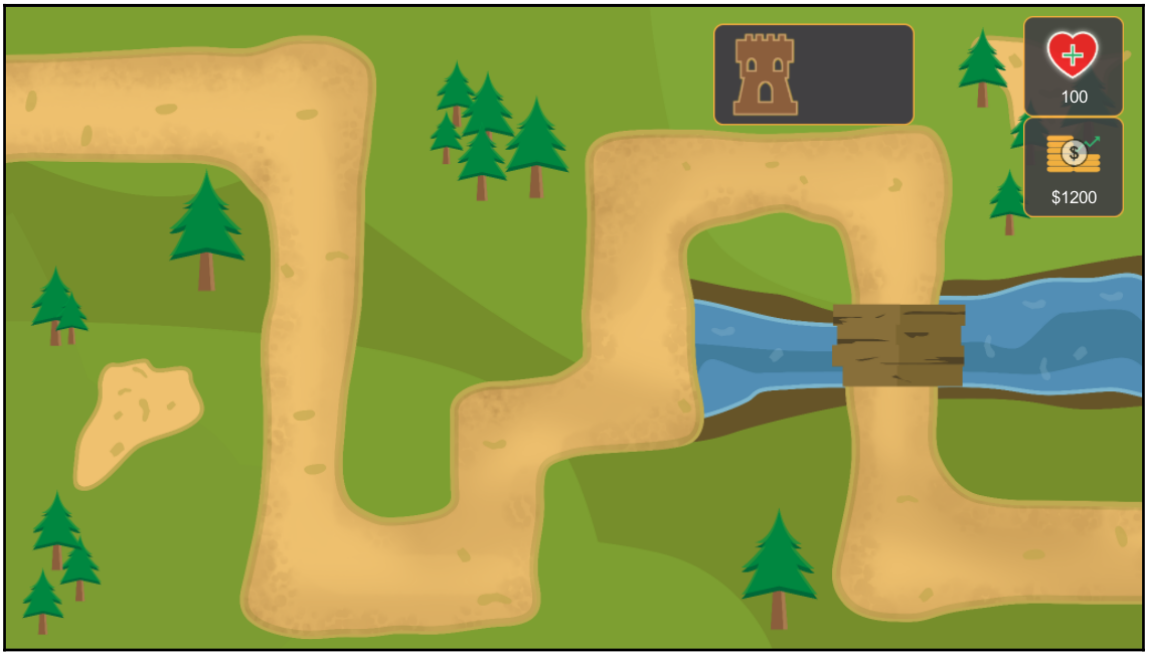
We don't have any parameters to set, so just save the script and our money system is ready to go.

The tower seller

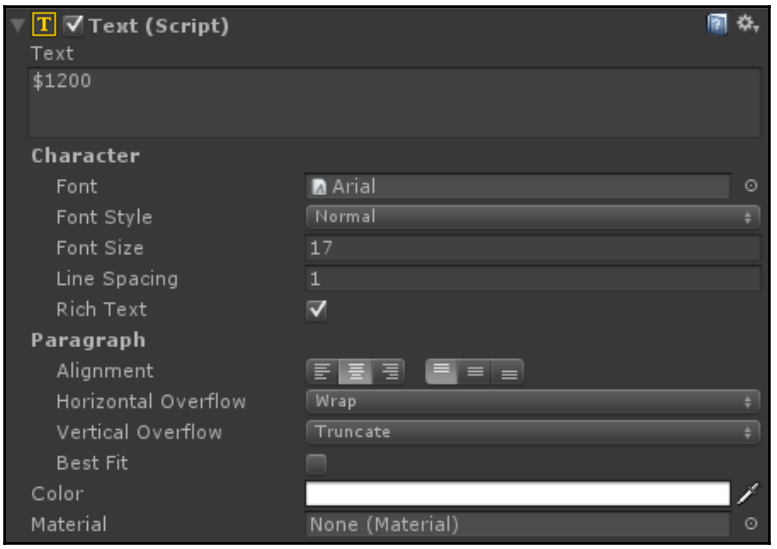
One of the key gameplay elements of a tower defense game is, of course, the ability to buy and place towers. Buying a tower is something that the player should be able to do through the UI. In this section, we will learn how to create a button for the player to buy towers. In the next chapter, we will also see how to place it in the map.

Creating and placing the tower seller

Let's start by creating a new Image by right-clicking on the **Hierarchy** panel and then **UI/Image**. We should also rename it `TowerSeller` and assign to the **Source Image** the `tower_rect` image in our package. We may want to press **Set Native Size** and then scale it down to fit the screen. Finally, we can drag and drop it next to the `LivesCounter`, as shown in the following picture:

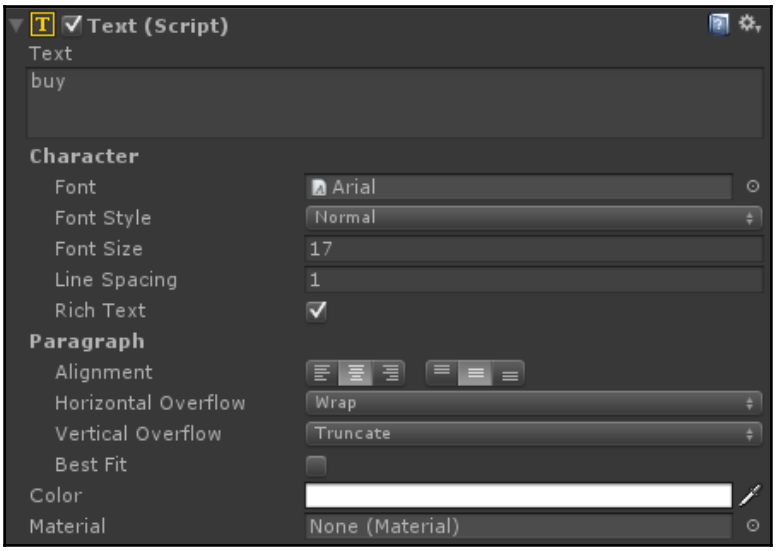


Now, we need to add text where the price of our tower will be shown. Right-click on **TowerSeller** and then **UI/Text**. Rename it `PriceText`, and we need to change its setting to the following:



In particular, we have set its color to white and the **Font Size** to `17`. In this way, it has the right dimensions and color to fit in our UI. Finally, we have to place it just next to the `LivesCounter`, but still inside the box, as in the next image:

The next step is to add a button where the player can click to buy the tower. Right-click on TowerSeller again and then **UI/Button**. We have to assign the **Source Image** to the `empty_rect` image in our package. Also, we should adjust the text inside the button by clicking on its child, called `Text`, as default. Similar settings to the price text apply here too, and change the text into `buy`. At the end, we should have something similar to this:



If we did everything right, our scene should look like the following:

Now, we need to add a script to it. This will handle the interaction with the player, by giving him the possibility to buy a tower. To create the script, select TowerSeller and, in the Inspector, navigate to **Add Component** | **New Script**. Name it `BuyTowerScript`, and then click on **Create and Add**. Double-click to open it.

Scripting the tower seller

The first thing to do is to add the following line at the beginning of our script:

```
using UnityEngine.UI;
```

As result, we will be able to use the UI classes.

We need a private variable to store our `MoneyCounter`, since we need to get how much money the player has. In fact, we need to compare that value with the price to check whether the player has enough money to actually buy the tower:

```
    MoneyCounterScript moneyCounter;
```

On the other hand, we need three public variables. One is needed to store the `uiText` to show the price. Another one is the price itself, since you may want to adjust it without changing the script. The last one is the `towerprefab`, to instantiate it if the player buys it:

```
    public Text uiPrice;
    public int price;
    public GameObject towerPrefab;
```

In the `Start()` function, we can set our private variable by finding the `MoneyCounter` in the scene. Furthermore, we need to update the `uiText` that shows the price, since we suppose it won't change over time:

```
  void Start () {
        moneyCounter =
GameObject.Find("MoneyCounter").GetComponent<MoneyCounterScript>();
        uiPrice.text = "$" + price;
  }
```

Now, we need to create a function that is called every time the button is pressed. The first thing to do here is to get the current money for the player and then compare this to the price. If the player has enough money, then we need to reduce the amount of money by calling the `changeMoney()` function we have created in the previous section of this chapter. We need to pass to it a negative value, since we are reducing the player's money. Finally, instantiate a new tower on the mouse position:

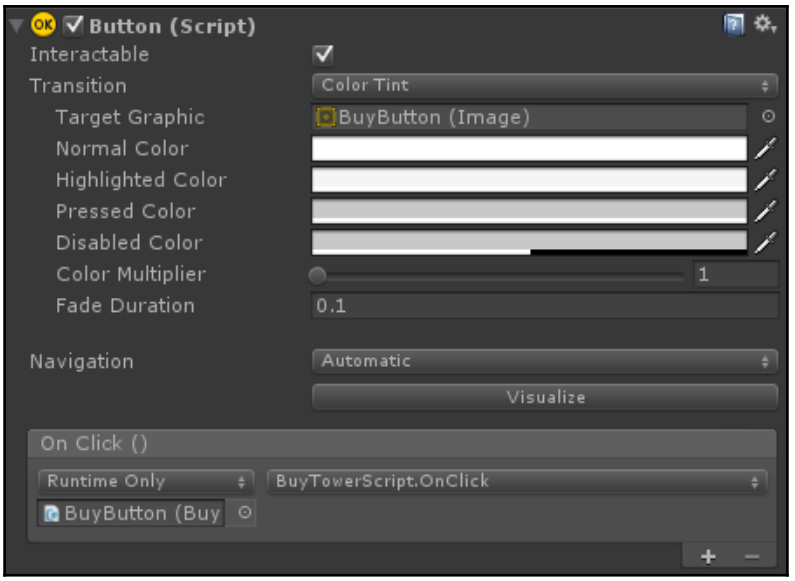
```
public void OnClick() {
    int money = moneyCounter.getMoney();
    if (money >= price) {
      moneyCounter.changeMoney(-price);
      Instantiate(towerPrefab,Input.mousePosition,Quaternion.identity);
    }
}
```

Here, we are not giving any feedback to the player when he presses the buy button. In fact, when he or she hasn't got enough money, there is nothing set to relay this information back to the player. Since, in general, it's a good practice to give feedback to the player, we need to add an `else` statement. Inside this branch, we can implement a way to warn the player that he or she hasn't enough money. For instance, it could be a sound or voice, red text on the screen, or both. This depends also on the kind of atmosphere you want to give to your game.

We can save the script and return back to the Unity **Editor**. In fact, we haven't finished yet, since we need to assign our variables to the script, and call the `OnClick()` function every time the buy button is pressed.

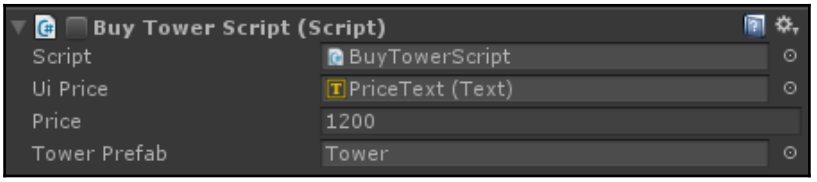
Finishing the tower seller

In order to finalize our seller, we need to link the `OnClick()` function with the event of pressing the buy button. This can be done by using the UI events integrated into Unity. Select **BuyButton** from the **Hierarchy** panel. In the **Inspector**, at the bottom of the **Button** component, there is a tab called **On Click ()**. To add a new event, press the **+** in the lower-right corner. Now, in the **Object** variable, we have to drag the script that is just below the tab, `BuyTowerScript`. Then, click on the drop-down menu where `no function` is written and select **BuyTowerScript** | **On Click ()**. Once we have done all of this, the **Button** component should appear like the following:



Now, we still have to assign the variable to the script that we have just created.

Drag and drop the **PriceText** from the **Hierarchy** panel into the `uiPrice` variable. Then, set a price for your tower, for instance, `1200`. Finally, drag and drop from the **Project** panel the `towerprefab` we created in the previous chapter. However, keep in mind that we are going to modify it in the next chapter. In the end, our script should be like the following, in the **Inspector**:



Upgrading the towers

Upgrading and selling the towers might be a little bit tricky. For this reason, this section is completely optional and you can feel free to skip it. Furthermore, the code here could be optimized, but it has been left without optimization for the sake of learning. In fact, this way it is easier to understand. Once all the key concepts that have been covered in this section are clear, the reader is invited to improve the code as an exercise.

How it works

Every time the player selects a tower, a menu appears and gives the player the possibility of upgrading or selling the tower. Therefore, the menu that we are going to create in this section will be disabled from the beginning, and it will be enabled when a tower is selected. We will see how a tower is selected in the next chapter, but the key idea is that every time a tower is selected, it has to be communicated to our menu which tower it is. In this way, our menu can correctly upgrade the tower, or destroy it if the player decides to sell it.

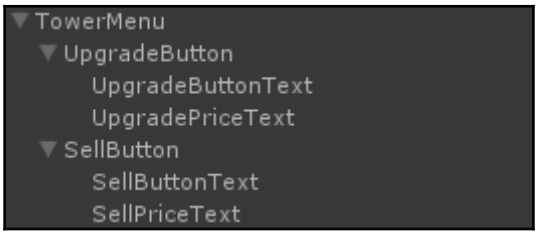
Creating and placing the tower menu

As for the previous UI elements that we have created so far, let's start by creating an UI Image and renaming it `TowerMenu`. Change its **Source Image** to `tower1_rect` from our package. Scale it properly and place it in the screen as in the following image:

Next, we need to create two buttons, one for upgrading the tower and the other for selling it. Rename `UpgradeButton`, along with its text, as `UpgradeButtonText`, and `SellButton`, along with its text, as `SellButtonText`. For both, we should use as **Source Image** the `empty_rect` from the package. As for the text, we can use the same settings we used for the **BuyButton**, and replace the text variable with `upgrade` and `sell`. It should now appear like this:

Please note that the text of the two buttons has moved slightly up. This is because we now have to add a UI text to each button and rename them, respectively, `UpgradePriceText` and `SellPriceText`. Their font size should be smaller than the other one, a good value being `12`. So our **Tower Menu** should appear like the following:

Since we have added a lot of components, here is the recap on how the **Hierarchy** Panel should be:



Now, we need to add a script to it. This will handle all the interaction with the player, by giving him or her the possibility to upgrade or sell a tower, and with the towers themselves. To create the script, select **TowerMenu** and, in the **Inspector**, navigate to **Add Component | New Script**. Name it `TowerMenuScript`, and then click on **Create and Add**. Double-click to open it.

Scripting the tower menu

Since we are going to use the UI, as we did with the other sections in this chapter, we need to add the following line at the beginning of our script:

```
using UnityEngine.UI;
```

As result, we will be able to use the `UI` classes.

This script has many variables, so let's break them down. The first one is a variable to store the `MoneyCounter`, as we did in the `TowerBuyScript`. We need this reference, since by upgrading or selling the player's tower, his amount of money will change:

```
MoneyCounterScript moneyCounter;
```

Then, we need a public variable that is the current selected tower. It has to be public, since it will be set by another script that we will see in the next chapter. However, since we don't need to change it through the **Inspector**, we can hide it using an attribute:

```
[HideInInspector]
public TowerScript currentTower;
```

Next, we need some variables to store our UI components. In particular, we need a private variable to store the Image where this script is attached. Since it is easy to get, we will leave this to the `Awake()` function. The other two are for the two prices, the upgrade price and the selling price:

```
private Image uiImage;
public Text upgradePriceText;
public Text sellPriceText;
```

Now, we need all the different settings for each level of the upgrades. These include the graphic needed to change the menu and the price values. Since there are three different levels of upgrade, we need to repeat these variables three times. To visualize them better in the Inspector, we could use a `Header` attribute for each group. So, this is the first one:

```
[Header("Level 0 Settings")]
public Sprite menuLevel0;
public int upgradePriceLevel0;
public int sellPriceLevel0;
```

And this is the second one:

```
[Header("Level 1 Settings")]
public Sprite menuLevel1;
public int upgradePriceLevel1;
public int sellPriceLevel1;
```

Finally, this is the third group of variables. Of course, we don't need the price to upgrade it, since it is the maximum level:

```
[Header("Level 2 Settings")]
public Sprite menuLevel2;
public int sellPriceLevel2;
```

The last variables we need are just internal values to store and take into account the current situation of the tower. Their names are quite self-explanatory:

```
private int level;
private int currentUpgradePrice;
private int currentSellPrice;
```

After so many variables, its time to start to writing our functions. The first one is the `Awake()` one. We don't use `Start()`, since this component will start disabled. Here, we just take the references to the `MoneyCounter` and the Image component where this script is attached:

```
  void Awake () {
        moneyCounter =
GameObject.Find("MoneyCounter").GetComponent<MoneyCounterScript>();
        uiImage = GetComponent<Image>();
  }
```

Then, we need to write an `OnEnable()` function, which is called every time the component is enabled again. This happens when a tower is selected. Here, we need to retrieve the current level of upgrading of the tower and update all the graphics of the component, including the prices. This can be done using a `switch` statement:

```
    void OnEnable() {
     if (!currentTower)
            return;
        level = currentTower.upgradeLevel;
        switch (level) {
            case 0:
                uiImage.sprite = menuLevel0;
                upgradePriceText.text = "$" +
upgradePriceLevel0.ToString();
                currentUpgradePrice = upgradePriceLevel0;
                sellPriceText.text = "$" + sellPriceLevel0.ToString();
                currentSellPrice = sellPriceLevel0;
                break;
            case 1:
                uiImage.sprite = menuLevel1;
                upgradePriceText.text = "$" +
upgradePriceLevel1.ToString();
                currentUpgradePrice = upgradePriceLevel1;
                sellPriceText.text = "$" + sellPriceLevel1.ToString();
                currentSellPrice = sellPriceLevel1;
                break;
            case 2:
                uiImage.sprite = menuLevel2;
                upgradePriceText.text = "-";
                sellPriceText.text = "$" + sellPriceLevel2.ToString();
                currentSellPrice = sellPriceLevel2;
                break;
        }
    }
```

The next function we have to write is called when the `UpgradeButton` has been pressed. Here, we just have to check whether the player has enough money, and eventually call the `Upgrade()` function on the tower. Probably, the compiler will give you an error, because that function is not defined yet in the `TowerScript`. We will create it in the next chapter:

```
public void upgrade() {
    if (level == 2)
        return;
    int money = moneyCounter.getMoney();
    if (money >= currentUpgradePrice) {
        moneyCounter.changeMoney(-currentUpgradePrice);
        currentTower.Upgrade();
        gameObject.SetActive(false);
    }
}
```

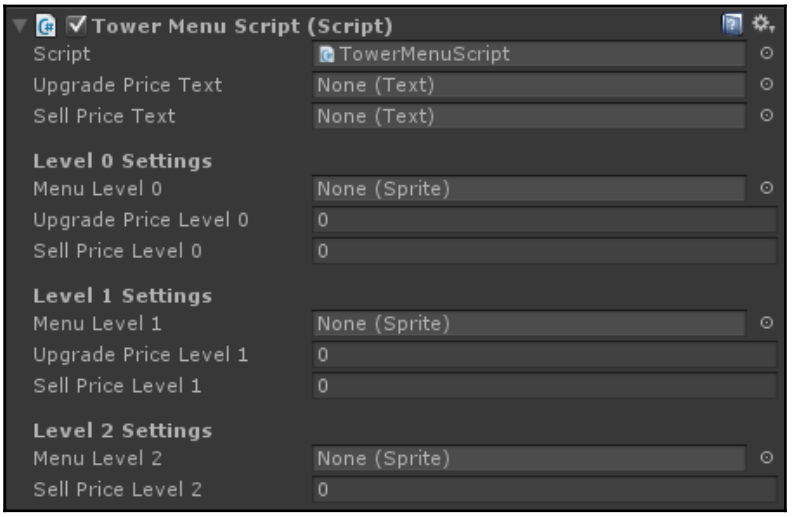
The last function is called when the `SellButton` has been pressed to sell the tower. Here, we just give the player the money back and destroy the tower:

```
public void sell() {
        moneyCounter.changeMoney(currentSellPrice);
        Destroy(currentTower.gameObject);
        gameObject.SetActive(false);
}
```

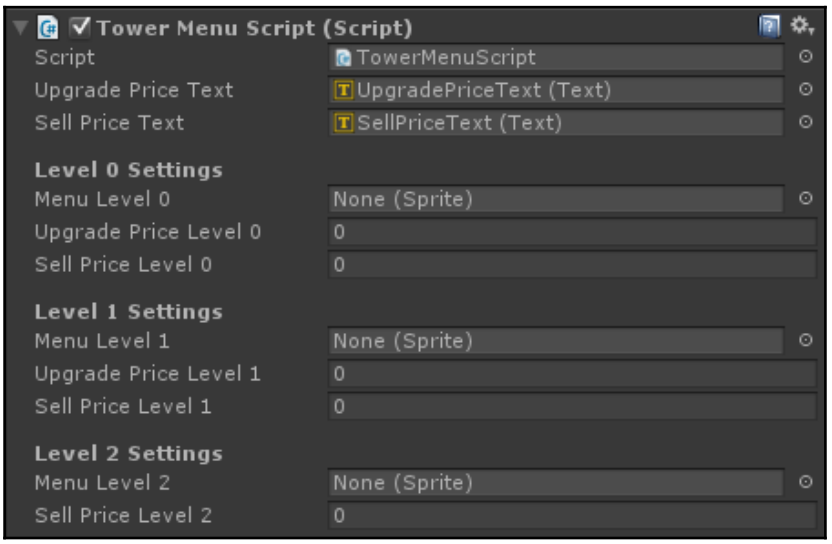
Finally, we have finished this long script. Let's save it and come back to Unity, where we still have to set all its values.

Finalizing the tower menu

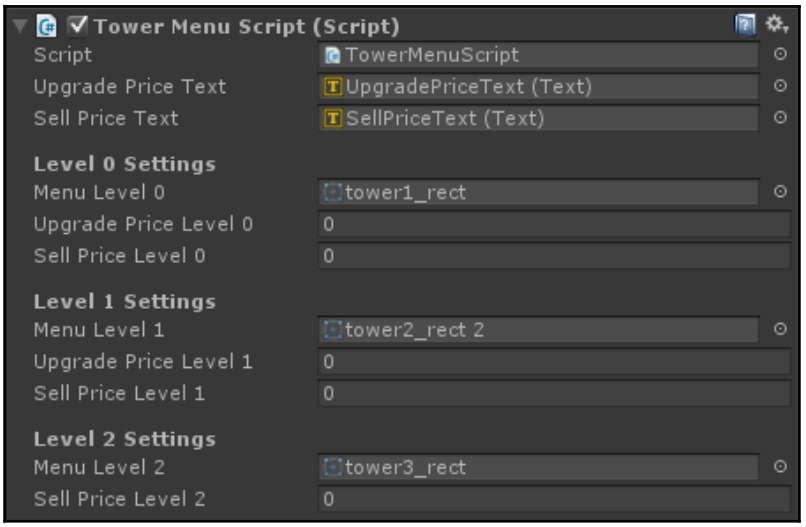
If we look at the **Inspector**, we should see our script with all the visible variables:



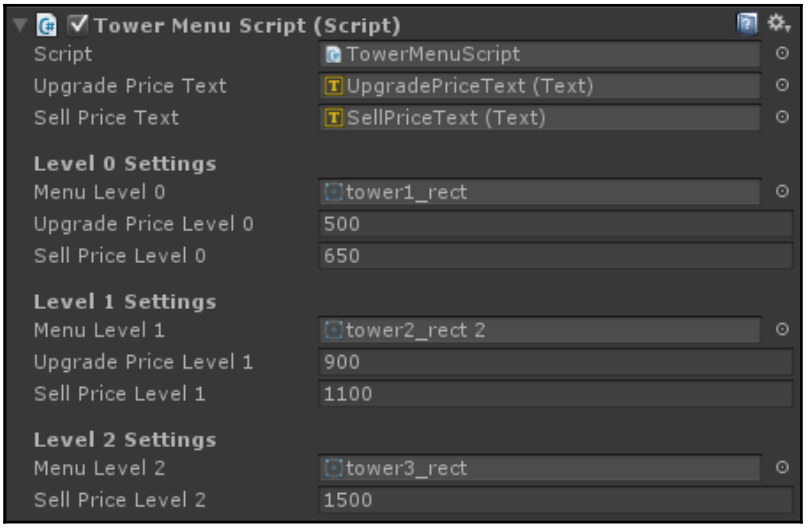
Let's start to drag and drop inside the **Upgrading Price Text**, our `UpgradePriceText` from the **Hierarchy** panel and inside our **Sell Price Text**, our `SellPriceText`, always from the **Hierarchy** panel. Now, our script should look like the following:



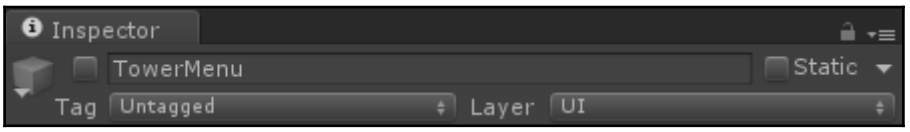
The next step is to fill, for each level, the graphic for the menu. Of course, for level zero we use the one we used as placeholder, the `tower1_rect` from our package. For the other two levels, we always use `tower2_rect` and `tower3_rect` respectively, from our package:



Here, we can fill the remaining variables according to our game. In this example, we are going to use the following values:

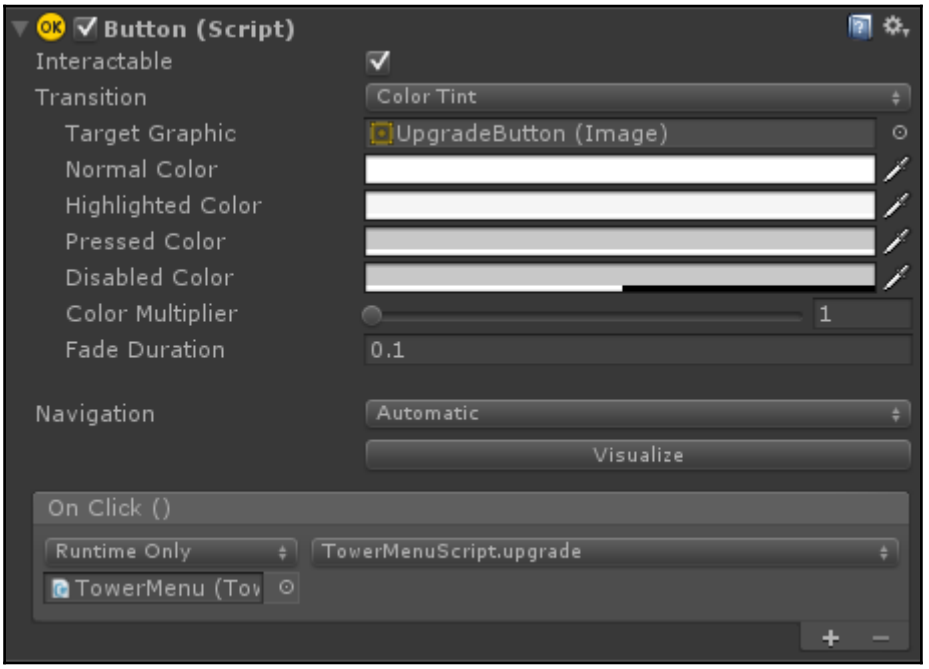


Finally, we need to disable the entire `TowerMenu`, by unchecking the small box next to its name in the **Inspector**:

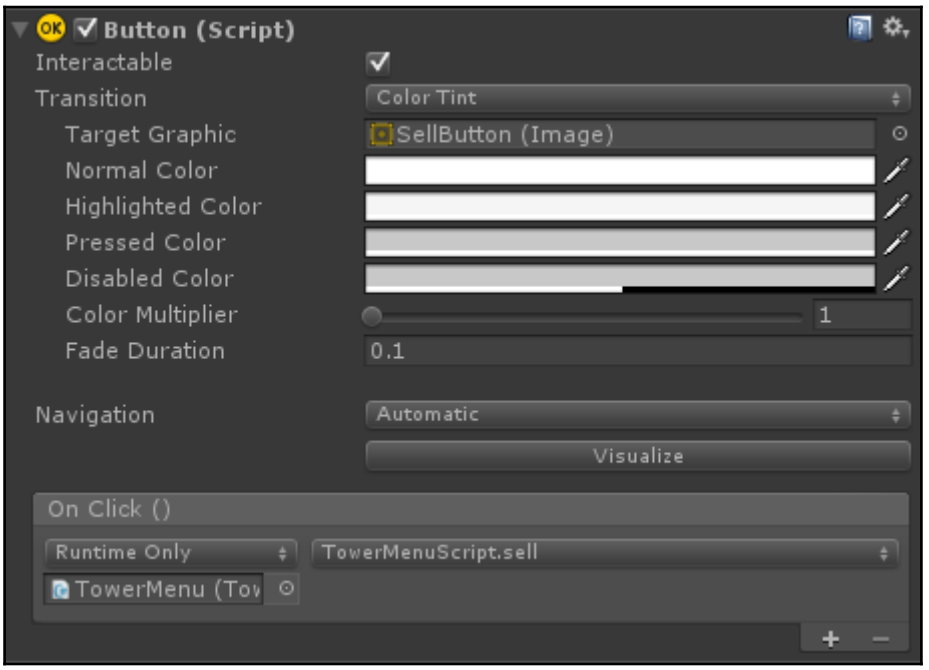


The last thing to do is to assign the right functions to the buttons that we have created.

Select **UpgradeButton**, and, in the **Inspector**, add a new event in the **On Click ()** tab. Drag the **TowerMenu** into the `Object` variable and, from the drop-down menu, select **TowerMenuScript.upgrade**. As a result, we should see this in the **Inspector**:



Similarly, select **SellButton** and add a new event in the **On Click ()** tab. Drag the **TowerMenu** again into the Object variable, and this time, from the drop-down menu, select **TowerMenuScript.sell**. Here is the final result in the **Inspector**:



Congratulations on having completed this optional section to integrate the possibility of selling and upgrading the towers into your game. Of course, to finish what we have started in this section, we have to follow the optional section in the next chapter as well.

Summary

In this chapter, we created the UI for the `Tower Defense` project. Inside this, we have implemented the logic to keep a lives counter, a money counter, and a tower seller. Furthermore, for those who wanted to challenge themselves, we have also seen how to implement a menu for the towers to make them sellable and upgradable.

In the next chapter, we are going to polish the project and finalize it by implementing a game manager and wrapping everything together.

9
Finishing the Tower Defense Game

Now that we have created all the single game elements for our Tower Defense game, we need to bring them all together. We will do this by creating a **Game Manager** that will control the entire flow of the game.

In particular, we will learn how to exchange information from the Game Manager to other elements, such as indicating to enemies the way to go, or decreasing the number of lives that the player has when the enemies have reached the end of their path. This also requires that we integrate the UI which we created in the previous chapter.

Later, we will see how to allow the player to place their own towers where they want in the map, but still having some constraints about the areas.

Then, we will learn how we can spawn enemies and how to set the *game over* conditions, both for winning and losing.

Finally, for those who have finished the optional section in the previous chapter about how to upgrade the tower, here they will learn how to finish what they started by modifying the `TowerScript`.

At the end of this chapter, there is a section with some suggestions about how you might improve your game on your own while improving your skills along the way. As the saying goes: *practice makes perfect.*

To summarize, this chapter will deal with the following topics:

- Creating waypoints to move enemies
- Integrating the UI that was developed in the previous chapter into the game
- Using colliders to implement constraints for different areas of the map
- Allowing the player to place their own towers, with respect to some constraints
- Creating a spawn system to generate enemies
- Finishing the gameplay by imposing game over conditions
- Upgrading the towers (optional section only for those who followed the previous optional section)
- Reading about different ways to keep improving your skills after you have finished with this book

Getting ready

During this chapter, we are going to handle most of the logic remaining within a single script. Let's start by creating a game object to hold it. Right-click on the **Hierarchy** panel and then click on **Create Empty**. Rename it `GameManager`. This name is important because we will use it to find this game object in other scripts. Next, click on **Add Component** | **New Script** and call it `GameManagerScript`. As a result, we are ready to implement the finishing touches to our game.

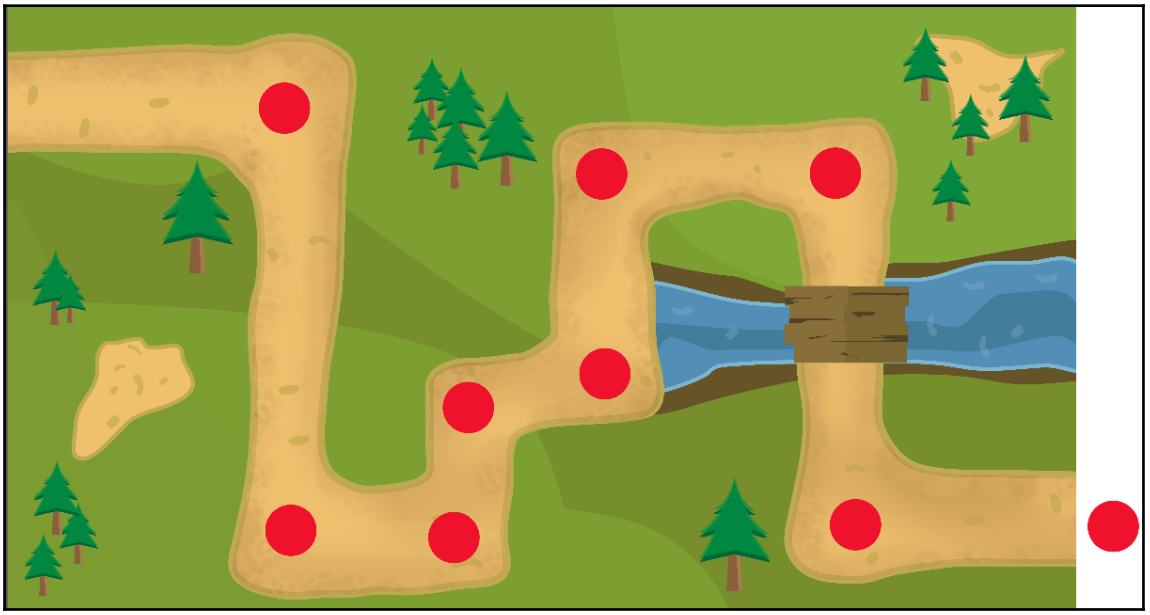
Waypoints for enemies

Now, the first thing to finish is how the enemies move around the map. In *`Chapter 7`*, *Tower Defense Basics*, we saw how to script them in order to make them move from one waypoint to another. As a short recap, a waypoint is a special point on the map where the enemies change their direction to move towards another waypoint. They can contain logic to actually lead the character to specific places that change over time, such as next to the player. They can also perform part of the decision-making process. For example, imagine a Tower Defense game where the path of the enemies splits in two. In this case, the waypoints can be used to decide which direction a particular enemy should take. The advantage of waypoints is that, in some cases, they can be more efficient than implementing a complete Pathfinding algorithm.

Here, we don't need to implement a particular logic behind the waypoints. However, they are a useful tool since they allow us to move enemies around the map easily. In this section, we will learn how to create waypoints.

Getting the waypoint coordinates

Before we start creating waypoints, we first need to decide where to place them on the map. Therefore, we need to find all the places where our enemies change their direction. In this simple case, they are located in all the corners of our path. In the following screenshot, they are indicated by the red dots:



As we can see, there are nine dots and we need one waypoint that is slightly outside the map. This final waypoint is the finishing spot, where the enemy succeeds in his mission – to damage the player. In fact, in the `EnemyScript`, when the enemy reaches the end, we just destroy it. For this reason, this spot must be outside the visible area.

Now that we have marked them with red dots, we need to take note of their coordinates on the map. A fast way to do it is to drag an enemy prefab around the map in the **Scene** view and then to take note of the position of our waypoints. In this case, we could obtain the following data:

Waypoint number:	**X-coordinate**	**Y-coordinate**
1	-18	15
2	-18	-17
3	-6	-17

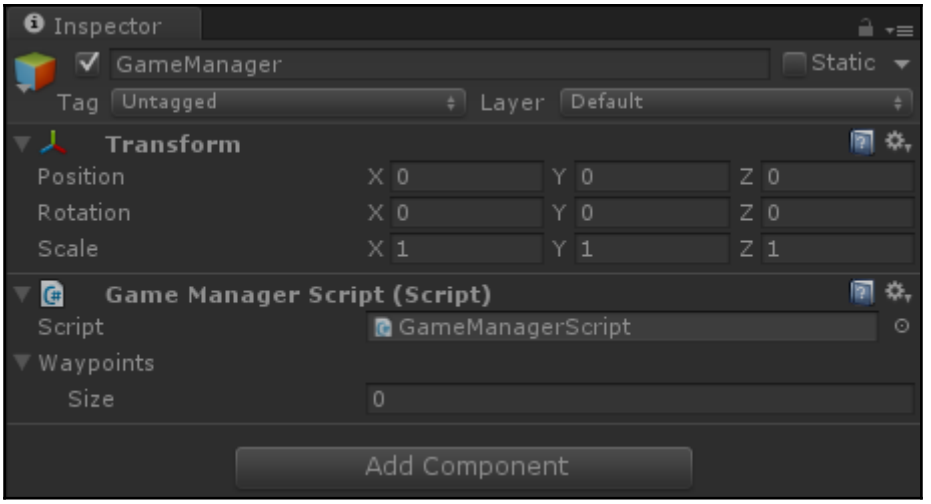
4	-6	-7
5	5	-4
6	5	11
7	22	11
8	23	-15
9	42	-15

Implementing waypoints in the Game Manager

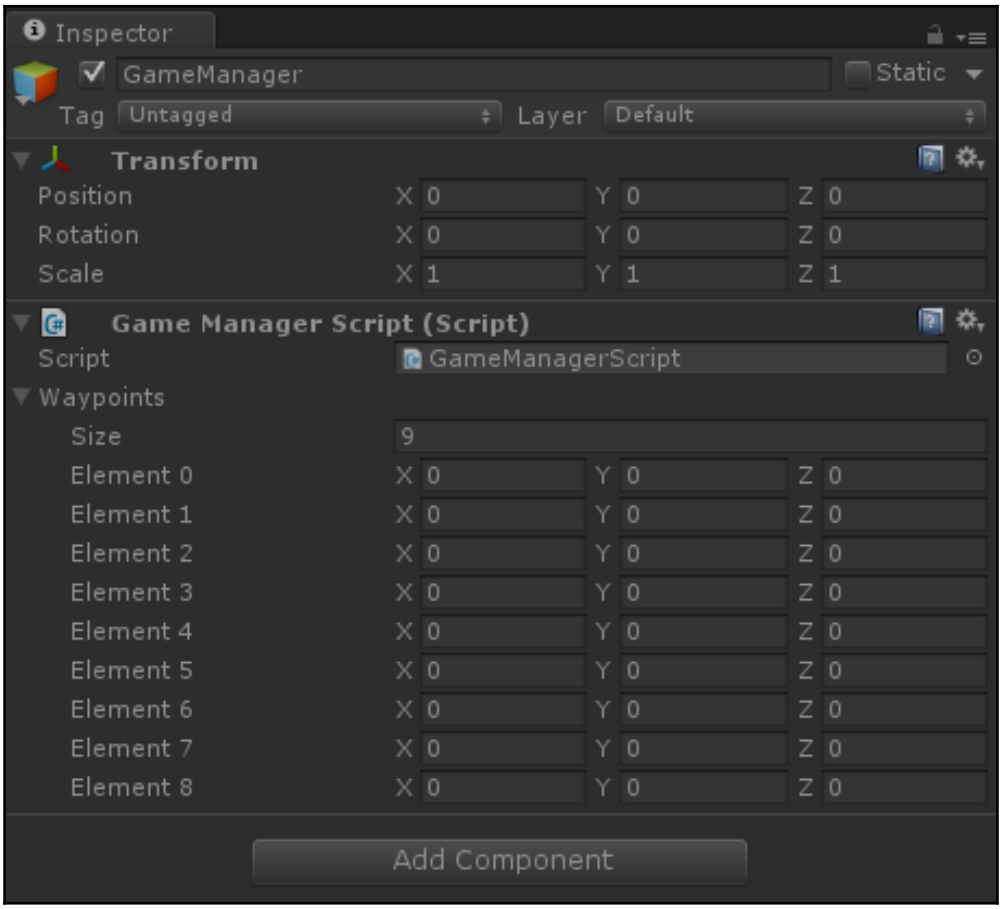
The next step is to actually add them into our Game Manager. Therefore, let's start by opening our `GameManagerScript` so we can create a new transform array variable to insert those values. Of course, we need to set this variable to public, so we can set the values in the **Inspector**:

```
public Vector3[] waypoints;
```

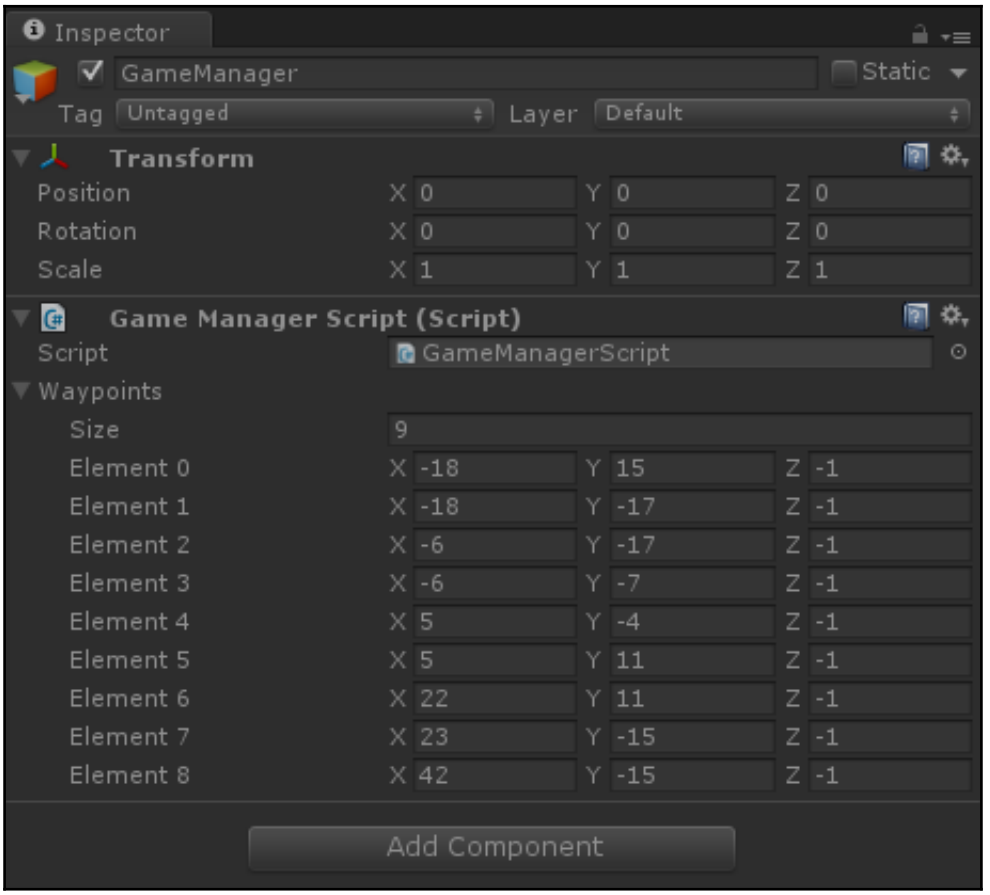
Save the code, and you should see something like this in the **Inspector**:



We need to set the number of elements of our array with the number of waypoints that we have found, in this case, nine. Therefore, our **Inspector** should now look as follows:



Finally, we can fill all those values with our waypoint positions. But what about the Z-axis? Since we don't want the enemies to change their Z-axis, we can just set its value to the same Z-axis value of our `Enemy` prefab, which is `-1`. In the end, we should have something like this:



Passing waypoints to the enemies

One last step to complete in order to fully integrate the waypoints into our game is to actually give them to the enemies. If you remember from `Chapter 7`, *Tower Defense Basics*, we created a variable called `waypoints`, but we never set its value. Also, for this reason, we got so many errors if we tried to press the play button. Let's fix this by finally assigning this variable to our enemies.

To begin, open the `EnemyScript`. Since the waypoints change with the map that we are playing, we don't want to set them all the time for each enemy. Therefore, we can get them from the Game Manager, which is specific for a particular level. The first thing to do is to get a reference to it. We can do this by adding the following variable type of `GameMangerScript`:

```
private GameManagerScript gameManager;
```

Then, in the `Start()` function, we can get a reference to it, by using the `Find()` and the `GetComponent()` functions together:

```
  void Start () {
        gameManager =
GameObject.Find("GameManager").GetComponent<GameManagerScript>();
  }
```

Now we can finally get all the waypoints by copying the array on the `GameManager` into the local variables of the enemy by using this line:

```
  void Start () {
        gameManager =
GameObject.Find("GameManager").GetComponent<GameManagerScript>();
     waypoints = gameManager.waypoints;
  }
```

Let's save our script so that we can finish dealing with the waypoints.

Integrating the UI into the game

Now that we have our enemies ready to challenge the player, let's integrate the UI that we have created in the previous chapter into the game.

In particular, we are going to see how to integrate the **Lives Counter** and the **Money Counter** since they are the two core gameplay elements.

Integrating the Lives Counter

We need to integrate the lives counter for situations such as when an enemy reaches the end of its path, the number of lives of the player is decreased. This happens in the following way: the enemy triggers a function in the Game Manager when it has reached the end, and the Game Manager updates the number of lives in the Lives Counter.

Therefore, let's start by opening our `GameManagerScript` and adding another variable. This is required to store the `LivesCounterScript` reference, so that we can have access to the lives of the player:

```
private LivesCounterScript livesCounter;
```

To get the reference, we can set the variable inside the `Start()` function:

```
  void Start () {
        livesCounter =
GameObject.Find("LivesCounter").GetComponent<LivesCounterScript>();
  }
```

Now we need to add a function that is called from the enemy and updates the number of lives:

```
public void enemyHasReachedTheFortress() {
    livesCounter.loseLife(10);
}
```

As you can see, there is a `10` in the code. This is the *number of lives* that the player loses for every enemy that has reached the end of its path. A much better solution is to create a public variable that can be freely set in the **Inspector** of the Game Manager, so that we can change this value more quickly. As a result, your code will be more flexible. However, there is also a more flexible solution, where this value is stored inside the `Enemy` prefab. In this way, it changes with different enemies, and it can be passed as a parameter to the `enemyHasReachedTheFortress()` function. The implementation of both alternatives is left as an exercise.

Save the script and then open the `EnemyScript`. Here, we have to call the function that we have just created. From the previous section, we saw how this script receives a reference to the Game Manager inside the variable `gameManager`, so we don't need to get the reference again.

Now, immediately after `if(counter==waypoints.Length){` and before `Destroy(gameObject);`, we need to add the following line of code that just calls the function in the `GameManager`:

```
void Update () {
  if(counter==waypoints.Length){
            gameManager.enemyHasReachedTheFortress();
    Destroy(gameObject);
    return;
  }else{
  //...
```

Finally, save this script so that the Lives Counter is integrated in our game.

Integrating the Money Counter

Integrating the Money Counter is easier than the previous UI element. This is because we don't have to pass through the `GameManagerScript`. This is due to the fact that the money doesn't deal directly with the **Game Over** conditions, but we will see this in a little while.

Open the `EnemyScript` and let's add a variable to keep track of our Money Counter:

```
private MoneyCounterScript moneyCounter;
```

To get the reference of the `MoneyCounterScript` and store it in the variable, as we did for the Lives Counter, we need to add a line in the `Start()` function of the enemy:

```
  void Start () {
        gameManager =
GameObject.Find("GameManager").GetComponent<GameManagerScript>();
     moneyCounter =
GameObject.Find("MoneyCounter").GetComponent<MoneyCounterScript>();
        waypoints = gameManager.waypoints;
  }
```

Now, inside the `OnTriggerEnter2D()` function, and immediately after the `if(other.tag == "Bullet")`, we can call a `changeMoney()` function to increase the amount of money that the player has:

```
void OnTriggerEnter2D(Collider2D other) {
  if(other.tag == "Bullet"){
           moneyCounter.changeMoney(80);
    Destroy(other.gameObject);
    Destroy(gameObject);
  }
}
```

As you can see, there is an `80` in the code. This is the *amount of money* that the player has earned when they kill an enemy. A much better solution is to create a public variable. Therefore, it is easily adjusted in the **Inspector** of the prefab for this specific enemy. In addition, we can have different instances of this script with different values. This means having different enemies that give a different amount of money to the player. Furthermore, another interesting way to do this is to set a random value. As a result, your code will be more flexible to improve your gameplay. The implementation of the two alternatives is left as an exercise.

Save the script, and now the Money Counter is integrated into our game.

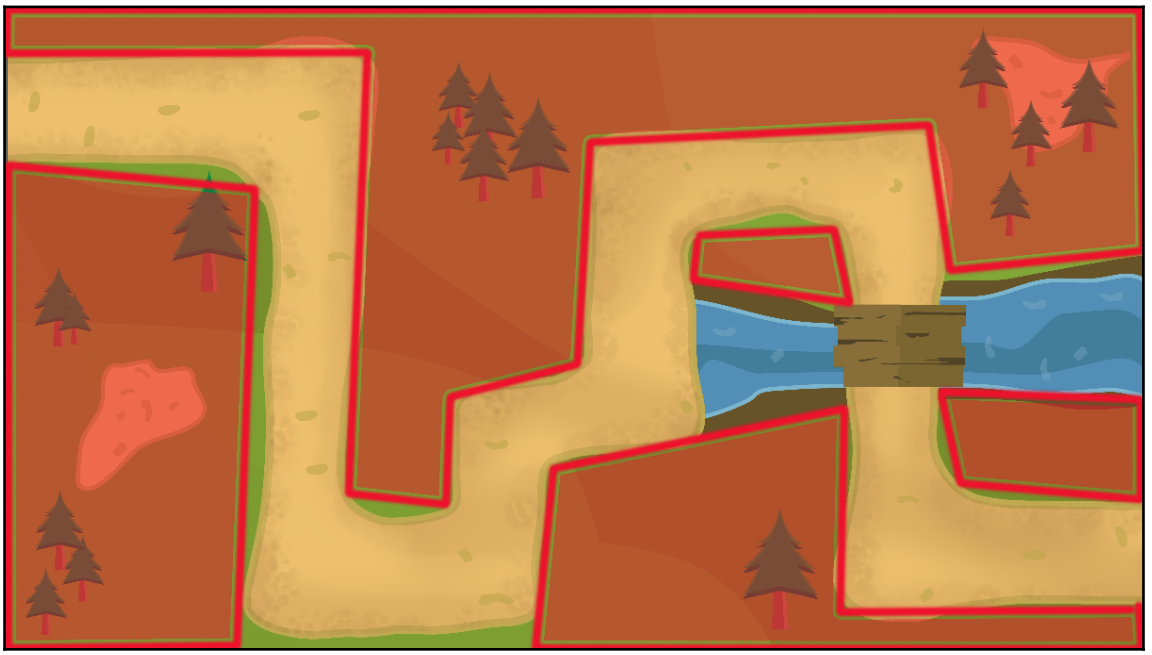
Placing the towers

When the player buys a tower, they have the possibility to place it where they want. In order to include this feature inside our game, we need to make some changes to our towers.

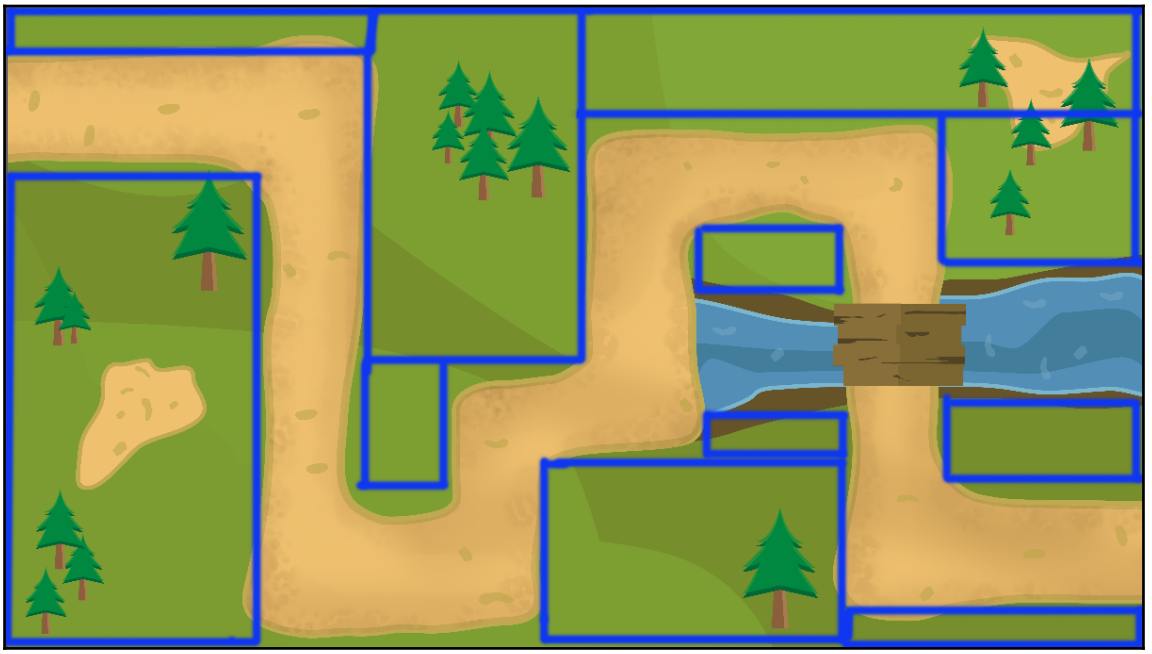
Allowed areas

To begin with, we should notice that the player is not free to place their towers wherever they want to on the map. In fact, they cannot place them along the path where the enemies are moving or in areas where there is water. Therefore, we need to implement this constraint.

Thus, we need to look at our map and find all the spots where the player can place the tower. In our case, the spots that we are looking for are the following:



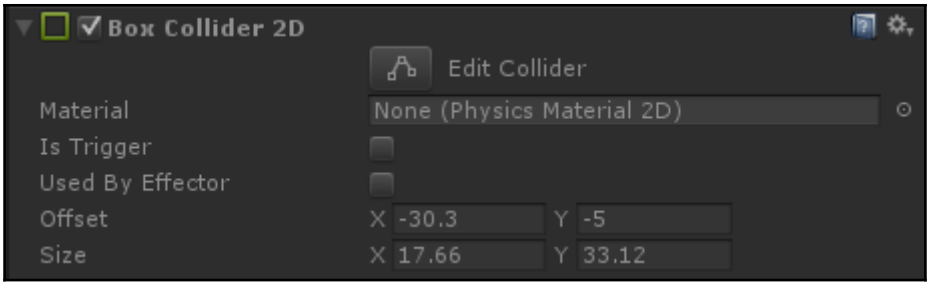
As we can see, it has a custom shape. Even if it is possible to implement a custom shape, it can be much more convenient to think in terms of rectangles and then to split our shape into rectangles. Of course, this can be done in more than one way; however, using fewer rectangles to cover the entire area is better. On the other hand, by using more rectangles you are able to better approximate your areas. A possible choice can be the following:



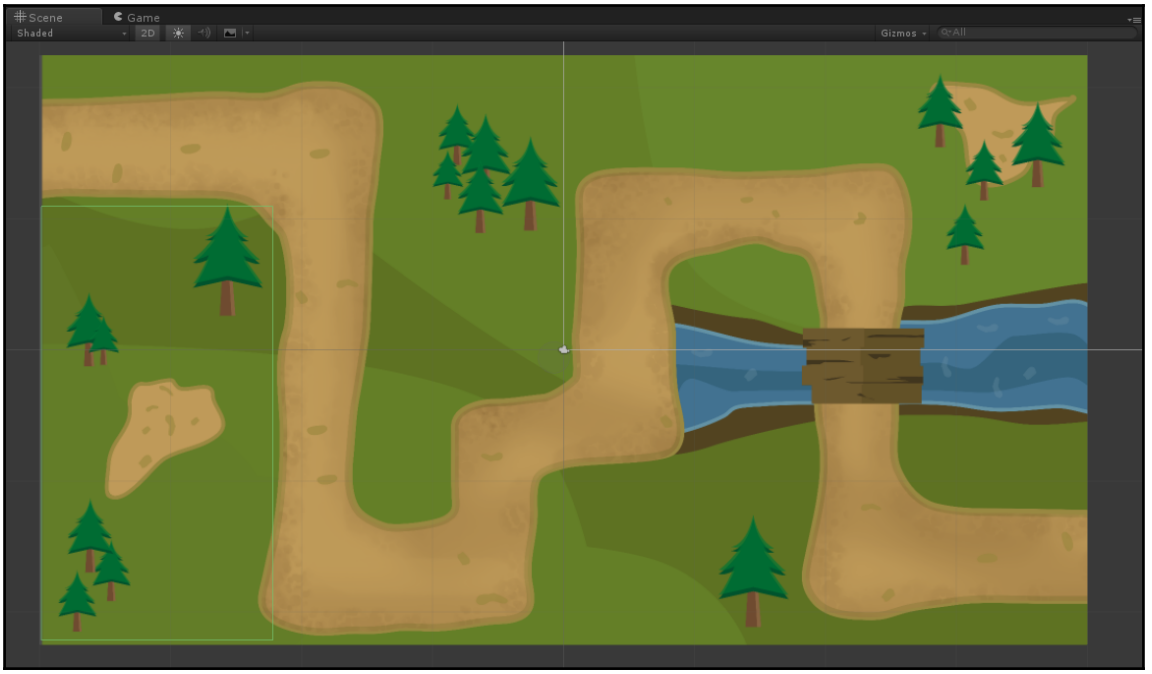
In the end, we have found eleven areas.

The idea here is that the placing script, which we will write in the next section, will look at a flag that is inside our Game Manager to see if the mouse is inside an allowed position to place a tower. We can implement this very easily by using a **Box Colliders 2D** on the `GameManager` object.

Let's start by adding a **Box Collider 2D** on the `GameManager` by clicking on **Add Component** | **Physics 2D** | **Box Collider 2D**. Then, we need to resize it to the same dimensions as one of the rectangles we have found, and by using the offset parameter, place it onto the map. For instance, by using this data:

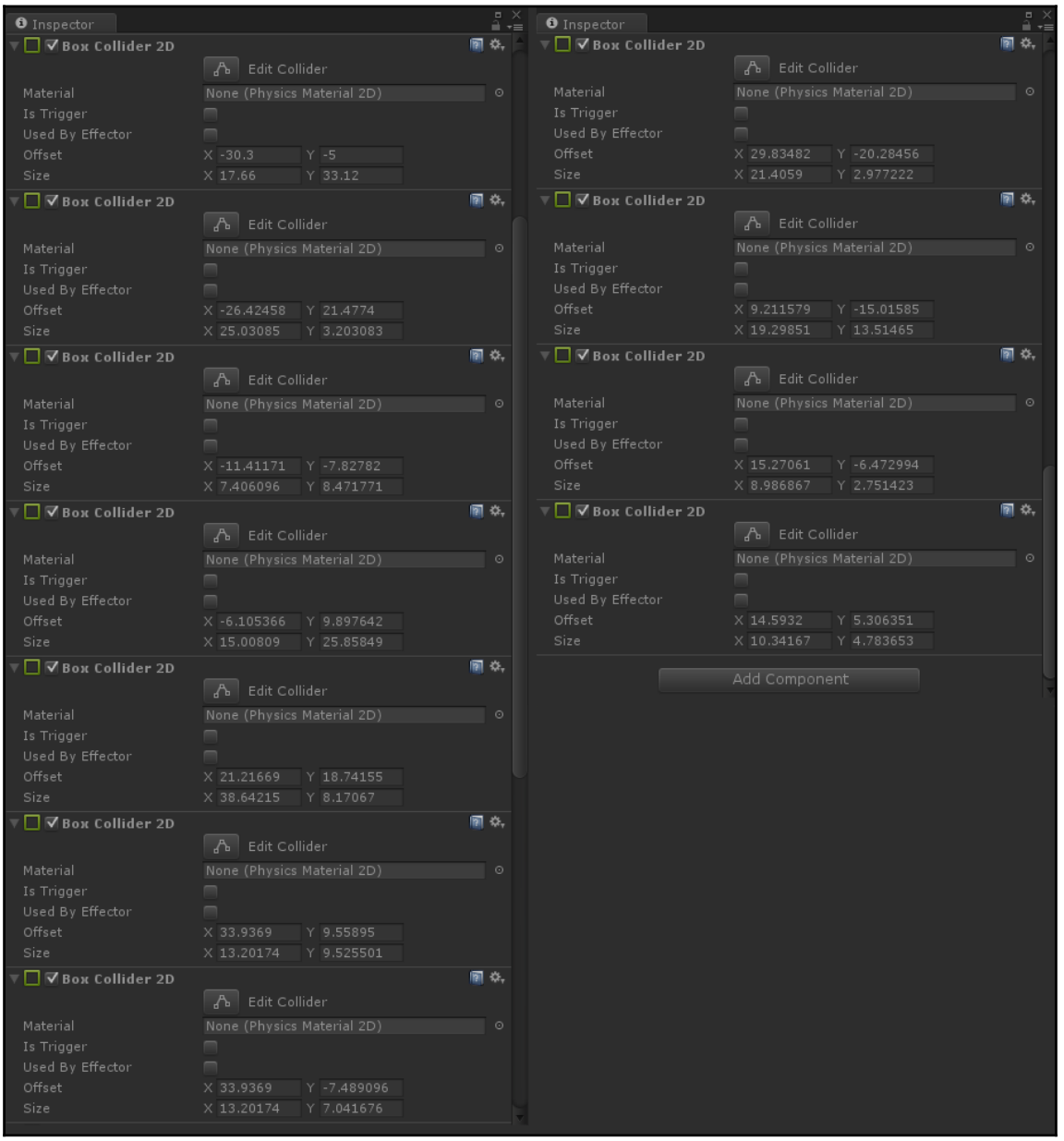


We should have a scene that looks something like this:

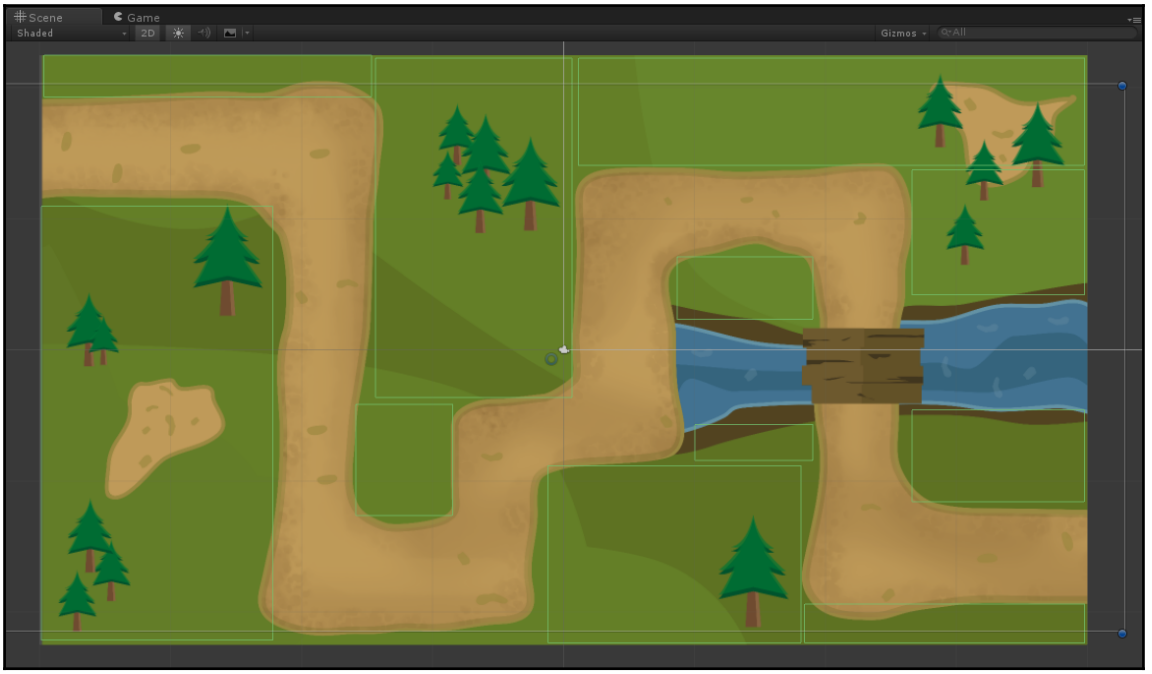


To speed up the process of placing the rectangles, you could use the `Edit Collider` button on the **Box Collider 2D** component.

We need to repeat this for all the areas that we have found, and so, in the end, the **Inspector** will look like the following (in the image, the Inspector has been split in two in order to fit all the components in one image):



And our map will have our rectangles represented as colliders, as follows:



Now, the second part of the story is to actually trigger the flag inside our `GameMangerScript`, so let's open it.

To begin with, we need to add a new private variable that holds the flag:

```
private bool isAreaAllowed;
```

It is just a Boolean that indicates whether the position of the mouse is inside an allowed area or not.

Since it is a private variable, we need to also expose a `get` method to retrieve its value:

```
public bool GetIsAreaAllowed() {
    return isAreaAllowed;
}
```

The next step is to detect when the mouse enters into one of these areas. Since they are colliders, this can be done by using two special functions in Unity: `OnMouseEnter()` and `OnMOuseExit()`. The engine calls them every time the mouse enters or exits respectively from one of the colliders attached to the `GameObject` of the script. In this case, to the colliders attached to the `GameManager`.

So, we can implement `OnMouseEnter()` in order to change the flag to a positive value:

```
void OnMouseEnter() {
    isAreaAllowed = true;
}
```

And the `OnMouseExit()`, instead, to set it to `false`:

```
void OnMouseExit() {
    isAreaAllowed = false;
}
```

Just save our script, and we are done with defining the allowed areas. We will see how to let the player place the towers in the next section.

Scripting the placement script

Now that we have our areas, we can proceed by selecting the tower prefab and then clicking on **Add Component** | **New Script** and naming it `PlacingTowerScript`.

The key concept of this section is that a new tower should follow the mouse of the player until they click on the screen to place it.

Before we start with this, we need to take the references to the `GameManager`. Let's add the following variable:

```
private GameManagerScript gameManager;
```

And in the `Start()` function we can get the reference to store in the variable:

```
  void Start () {
        gameManager =
GameObject.Find("GameManager").GetComponent<GameManagerScript>();
  }
```

Now we can focus on the `Update()` function. Here, we need to get the mouse coordinates and convert them into a point in the map by using the `ScreenToWorldPoint()` function from our main camera. This ensures that the point will be exactly the one that the player is pointing to. Since this is inside the `Update()` function, it will move at each frame to the mouse position. In the end, we will add an `if` statement that checks whether the player clicks, and whether the click is inside an allowed area. If so, we can get rid of this script by destroying it:

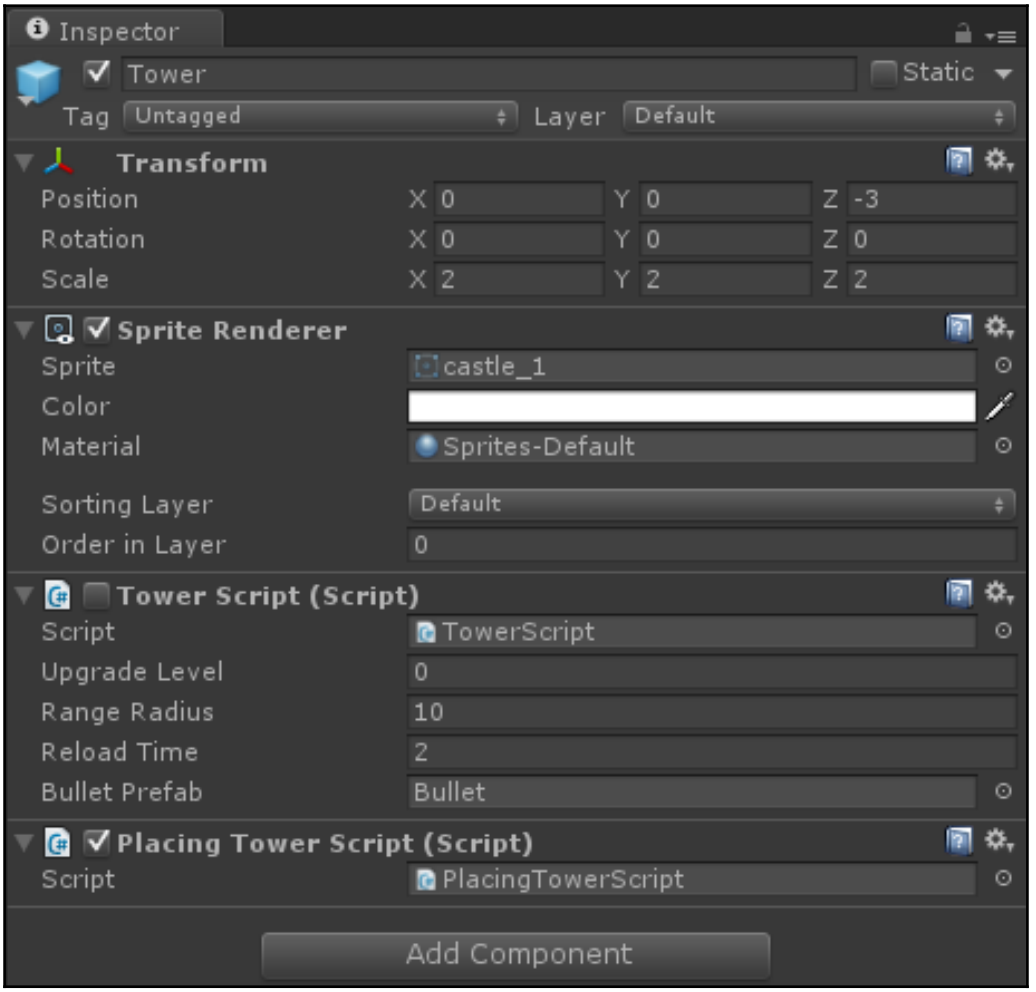
```
  void Update () {
        float x = Input.mousePosition.x;
        float y = Input.mousePosition.y;
        transform.position = Camera.main.ScreenToWorldPoint(new Vector3(x,
y, 7));
        if (Input.GetMouseButtonDown(0) && gameManager.GetIsAreaAllowed())
{
            Destroy(this);
        }
  }
```

Please note that number `7` is the new vector. We decided in `Chapter 7`, *Tower Defense Basics*, which of the *Z*-axes of the towers should be at `-3`. This function starts from the **Main Camera** that we have placed at `-10`, on the *Z*-axis, thus we need to add `7` in order to reach `-3`.

We can save the script, and now our tower, once created, moves along with the mouse button until a click is performed.

Final tweaking of the Tower prefab

There are still a few things to fix. First, we don't want the tower being able to shoot at the enemies while it is in `placing mode`, when the player is still deciding where to place it. Therefore, we need to go into our prefab and disable the `TowerScript` as follows:



In this way, when a tower is created, it will not act as a tower.

Now, the problem is that when it is placed, we want it to act again. In order to do this, we need to change our script. Let's open the `PlacingTowerScript` again and just after the if-statement in the `Update()` function, add this line of code:

```
        if (Input.GetMouseButtonDown(0) && gameManager.GetIsAreaAllowed())
{
         GetComponent<TowerScript>().enabled = true;
            Destroy(this);
        }
```

As a result, when the tower is placed, the `TowerScript` is also enabled again.

Furthermore, we want to add a collider to the tower when it is placed. As a result, the player will not be able to place another tower over an existing one. Inside the same `if`-statement, let's add the following line:

```
        if (Input.GetMouseButtonDown(0) && gameManager.GetIsAreaAllowed())
{
         GetComponent<TowerScript>().enabled = true;
      gameObject.AddComponent<BoxCollider2D>();
         Destroy(this);
        }
```

Creating an enemy spawner

The next thing to do is to actually spawn enemies against the player. This can be done in many different ways. Here, we are going to see a simple way to do that.

The key concept is that we have a coroutine that gradually spawns enemies.

If you are not familiar with coroutines, you can check out the following documentation: `http://docs.unity3d.com/Manual/Coroutines.html`.

Let's get started with opening the `EnemyScript` and adding a new variable that holds the position where we want to spawn the enemies. Since we want to set this in the **Inspector**, we need to set it to public:

```
public Vector3 SpawnPoint;
```

Furthermore, we need one more variable, public again, that stores the prefab for the enemy in order to spawn all the enemies:

```
public GameObject enemyPrefab;
```

Also, we need some other variables that we are going to use to set how many enemies the coroutine is going to spawn and how fast:

```
public int numberOfEnemiesToSpawn = 50;
public float minSpawnTime = 1f;
public float maxSpawnTime = 3f;
```

Now, in the `Start()` function, we need to start this coroutine, which we are going to create in the next step, and we pass the number of enemies to spawn as the parameter:

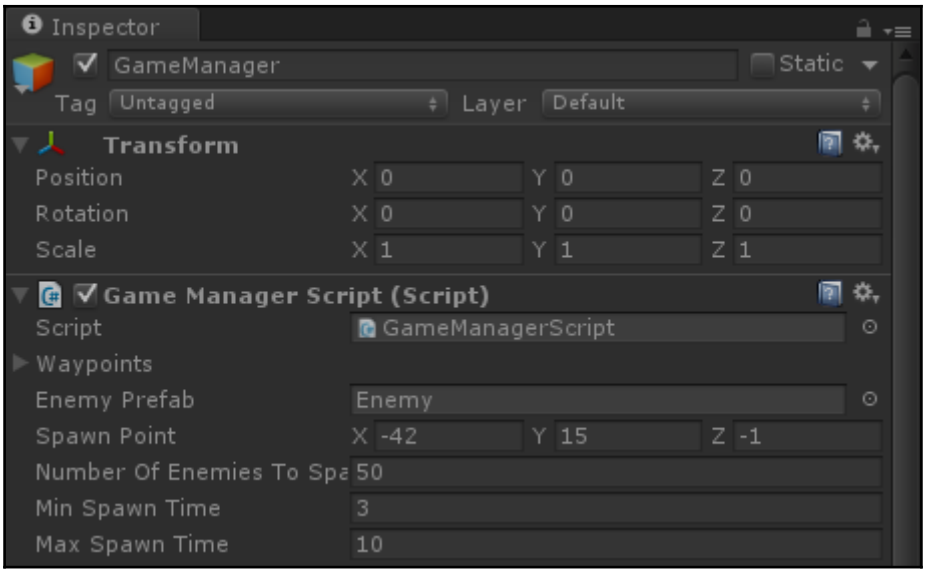
```
  void Start () {
        livesCounter =
GameObject.Find("LivesCounter").GetComponent<LivesCounterScript>();
     StartCoroutine("SpawnEnemies", numberOfEnemiesToSpawn);
  }
```

Let's create our coroutine by paying attention to use the same name that we have used in the `Start()` function. We need to loop over all the number of enemies and spawn the same number. Since we don't want them all together, we need to use a `WaitForSeconds()` function to create a delay. We need to be careful when setting a value for this delay as it can change the balance of the game a lot, so we need to consider how to design and implement this in order to maintain balance. In this case, we are going to use a simple linear interpolation to decrease the delay over time (so the more the game goes on, the faster the enemies are spawned):

```
    IEnumerator SpawnEnemies(int number) {
        for (int i = 0; i < number; i++) {
            Instantiate(enemyPrefab, SpawnPoint, Quaternion.identity);
            float ratio = i*1f / (number-1);
            float timeTowait = Mathf.Lerp(minSpawnTime, maxSpawnTime, 1 -
ratio);
            yield return new WaitForSeconds(timeTowait);
        }
    }
```

There are really lots of ways to implement this delay, and you should find the one that best suits your game. However, our implementation allows us to tweak its value by regulating the maximum and the minimum time that an enemy is spawned.

Finally, we can save our script. In the **Inspector**, we still need to set the values of the new variables that we have just created. Therefore, good values to set could be as follows:



Finishing the gameplay

We are almost there!

There are a few things to take into account in order to finally complete this project. If we press play, the game works, but there are no game over conditions. In particular, our game can have two outcomes: the player wins, by destroying all the enemies, or he fails in defending his fortress.

Winning conditions

Sometimes games go in our favor and the player rises victorious! In this case, he has managed to destroy all the enemies, taken away resources from his opponents, and left them with the bitterness of defeat.

A very easy way to see whether the player has destroyed all the enemies is to perform this check inside the coroutine of the previous section. At the end of the `for` loop, we can have a `while` loop where every second it checks whether there are no more enemies. In this case, we just run our game over screen (win). So, after we have opened the `GameManager`, let's add this to the end of our coroutine:

```
IEnumerator SpawnEnemies(int number) {
    for (int i = 0; i < number; i++) {
        //...
    }
 //GameOverConditions
 bool isGameOver = false;
 while (!isGameOver) {
     if (GameObject.FindGameObjectsWithTag("Enemy").Length == 0) {
         isGameOver = true;
         //GameOver Screen (win)
     }
     else {
         yield return new WaitForSeconds(1);
     }
 }
}
```

This is not the best way to implement such a coroutine, since we are using a very expensive function, `Find()`. However, it is called only every second and not every frame. In any case, a better implementation would be to have a counter which, once the spawning process has finished, keeps track of how many enemies are left. Every time that an enemy dies, it decreases. The implementation of this other method is left as an exercise.

By doing this, we have finally given the player the possibility to win the game.

Losing conditions

Nevertheless, the life for our player is not always be good, especially with a horde of slimes at the door of his fortress. When too many enemies cross the end of the path and the number of lives of the player goes to zero, there is nothing he can do, this is game over and the player has lost.

In order to implement this sad scenario, we need to check whether the number of lives of the player has reached zero. This can be done every time that an enemy reaches the end of the path. For this reason, we passed this through the `GameManagerScript` to decrease the number of lives in the *Integrating the UI in the game* section. Therefore, let's open the `GameMangerScript`, and modify the `enemyHasReachedTheFortress()` function to check whether the number of lives goes to or below zero. If so, we need to stop the coroutine that spawns enemies and check the winning conditions and display the game over screen:

```
public void enemyHasReachedTheFortress() {
    livesCounter.loseLife(10);

 //GAMEOVER CONDITIONS
 if (livesCounter.getLives() <= 0) {
     StopCoroutine("SpawnEnemies");
     //GameOver Screen (loose)
 }
}
```

By doing this, we have just allowed the terrible slimes to overcome the player.

Upgrading towers

If you have reached this point, it means that there is just one thing missing from your game. However, this is an optional section, and it is only for those who have done the optional section of the previous chapter, since we are going to finish what we started.

Finishing the TowerScript

In the previous chapter, we created a `TowerMenuScript` that interacts with one instance of the `TowerScript` stored inside the `currentTower` variable. In particular, it has access to one variable of the tower called `upgradeLevel`; however, we have never created this variable. So, open the `TowerScript` and let's add this one line:

```
public int upgradeLevel = 0;
```

It starts with the value 0, since now we have set the `TowerMenuScript`, the levels of our towers start from 0 up to 2.

Furthermore, `TowerMenuScript` calls the `Upgrade()` function of our tower, that we don't have. In this function, we need to increase the stats of the tower, including the level, and then we can also close the `TowerMenuScript`. To do this, we need to reference it, but for now, assume that it is stored inside the `towerMenu` variable. Of course, feel free to change how the stats are improved as you like. Therefore, we can write:

```
public void Upgrade() {
    rangeRadius += 1f;
    reloadTime -= 0.5f;
    upgradeLevel++;
    towerMenu.SetActive(false);
}
```

As we already have seen, having values inside the code is not a good practice. Replacing them with variables is left again, as an exercise.

Now we would like to open the **Tower Menu** when a tower is selected. As we have seen for the allowed areas to place the tower, we can use the function `OnMouseDown()`. So let's write it down, keeping in mind that it is called when the mouse clicks on our tower. We need to disable the `TowerMenu`, update its `currentTower` variable, and then enable it again:

```
void OnMouseDown() {
    towerMenu.SetActive(false);
    towerMenu.GetComponent<TowerMenuScript>().currentTower = this;
    towerMenu.SetActive(true);
}
```

As we already know, this function works only with a collider attached. But every time a tower is placed by the `PlacingTowerScript`, a collider is added. Therefore, we don't need to worry about it.

Now, everything seems to work, but we haven't yet defined the variable `towerMenu`! Since the `TowerMenu` is only one, we can share this variable among all the `TowerScript` instances by making it static. So, let's write:

```
public static GameObject towerMenu;
```

We have now declared it, but it is not set yet. This is because we will set this from the `TowerMenuScript`, and thus it has to be public.

Final adjustments to the TowerMenuScript

We need to open our `TowerMenuScript` again, because when the game starts it has to set the `towerMenu` variable to all the `TowerScript` instances. We can achieve this by adding the following line to the `Awake()` function:

```
  void Awake () {
        moneyCounter =
GameObject.Find("MoneyCounter").GetComponent<MoneyCounterScript>();
        uiImage = GetComponent<Image>();
     TowerScript.towerMenu = this.gameObject;
  }
```

As a result, we will have the `towerMenu` variable set, ready to use in the function written in the previous section.

Furthermore, we also need to disable the game object related to the `TowerMenuScript` from the beginning, because we don't want to show it from the beginning:

```
  void Awake () {
        moneyCounter =
GameObject.Find("MoneyCounter").GetComponent<MoneyCounterScript>();
        uiImage = GetComponent<Image>();
        TowerScript.towerMenu = this.gameObject;
      gameObject.SetActive(false);
  }
```

By doing this, we have finished implementing our upgrade and selling tower system. Even if this was harder than the rest of the game elements, it was definitely worth it.

Practice makes perfect

Now your game is ready! Congratulations!

However, there is always space to improve your skills. Thus, the aim of this section is to give you some directions on how you can improve your game and continue developing your skills.

The first thing that you can do is to extend the game into more than one level. This means creating a new map and loading it when the player wins the current one. Also, you need to redo some of the steps completed in this chapter, such as tweaking the `GameManager` to adapt it to the map. This includes setting waypoints and the allowed areas again.

In order to add a bit more fun to the game, you can create other kinds of enemies. Actually, the package used in this book has different enemies that you can use. They will use the same `EnemyScript` we have written, but with different parameters, such as moving faster. Also, remember to change the spawning system to spawn them too, maybe to spawn more than one enemy per time and organize them into waves as well.

Another interesting thing you could do is to create a map with different paths for the enemies, and use waypoints to decide which one the enemy will take. Additionally, you can implement the closure of the Tower menu when the player clicks on an empty spot in the map.

If you are thinking of publishing your game, then you shouldn't forget about audio and music. They are nice additions, and can create a great atmosphere for gameplay. Furthermore, audio can be used to give some feedback to the player of what is going on. For instance, when they try to place a tower in an area where they cannot, a sound may help them to understand that they cannot place it there. Moreover, you can complement audio feedback with visuals, which are also important for a game's success.

Finally, if you really want to push your skills and challenge yourself, you should try to optimize the code presented in this book to make it run more efficiently. Some suggestions on how to make some improvements have been left in the text as tips.

Summary

In this chapter, we finished creating our Tower Defense game. We have implemented a `GameManager` to wrap all the single pieces created inside the last two chapters together. In particular, we have integrated the UI as well as the game over conditions, along with a script to place the towers inside our game:

Finally, don't forget to have fun with your own video game!

Goodbye

Unfortunately, this beautiful journey into the world of game development by examples in Unity has come to an end, and we have learned a lot.

Index

L

M

O

P

R

S

T

U

V

W