A STEP-BY-STEP GUIDE TO CREATING YOUR GAMES

LEARN

C# PROGRAMMING

BY CREATING GAMES

WITH UNITY

(BEGINNER)

PATRICK FELICIA

Table of Contents

Learn C# Programming

By Creating Games

with Unity

_____

Learn C# with Unity

Patrick Felicia

Learn C# Programming

By Creating Games

with Unity

First published: July 2024

Published by Patrick Felicia

Credits

Author: Patrick Felicia

About the Author

Patrick Felicia is a lecturer and researcher at SETU (South East Technological University), where he teaches and supervises undergraduate and postgraduate students. He obtained his MSc in Multimedia Technology in 2003 and his PhD in Computer Science in 2009, from University College Cork, Ireland. He has published several books and articles on the use of video games for educational purposes, including the Handbook of Research on Improving Learning and Motivation through Educational Games: Multidisciplinary Approaches (published by IGI), and Digital Games in Schools: a Handbook for Teachers, published by European Schoolnet. Patrick is also the Editor-in-chief of the International Journal of Game-Based Learning and the Conference Director of the Irish Conference on Game-Based a popular conference on games and learning organized throughout Ireland.

Support and Resources for this Book

You can download the resource pack for this book; it includes solutions scripts for some of the sections in this book, as well as some of the files needed to complete some of the activities presented in this book.

To download these resources, please do the following.

If you are already a member of my list, you can just go to the member area using the usual password and you will gain access to all the resources for this book.

If you are not yet on my list, you can do the following:

Open the following link: http://learntocreategames.com/books/
Select this book C# Programming by Creating Games with
On the new page, click on the link labelled or scroll down to the bottom of the page.
In the section called your Free Resource enter your email address and your first name, and click on the button labeled I want to receive my bonus
After a few seconds, you should receive a link to your free start-up pack.
When you receive the link, you can download all the resources to your computer.

This book is dedicated to Mathis

_____

[ ]

Table of Contents

Contents

Preface

After teaching Unity and Game Programming for over 10 years, I always thought it could be great to find a book that could get my students to learn and apply key programming concepts while creating an interesting video game.

Many of the books that I found were too short and did not provide enough details on the reasons behind the actions recommended and taken; other books were highly theoretical, and I found that they lacked practicality and that they would not get my students' full attention. In addition, I often found that game development may be preferred by those with a programming background, but that people with an Arts background, even if they wanted to know how to create games, often had to face the challenge of learning to code for the first time.

As a result, I started to consider a format that would cover both aspects: be approachable (even to the students with no programming background), keep students highly motivated and involved, using an interesting project, cover the core and most important programming concepts, provide answers to common questions, and also provide, if need be, a considerable number of details for some topics.

I then created the book series entitled Learn C# Programming by Creating Video Games With Unity, that did just that. It combines a solid foundation in C programming with a game engine that makes it easy to create great video games.

This is the reason why I created this new book series entitled Learn C # Programming by Creating Video Games With it is for people who would like to learn and master programming while applying these concepts to a fun and interesting project.

In this book, focused on C# Programming, you will learn about C# concepts and how to use these to create games with Unity.

[ ]

What you Need to Use this Book

To complete the project presented in this book, you only need the most recent version of Unity and to also ensure that your computer and its operating system comply with Unity's requirements.

Unity can be downloaded from the official website and before downloading it, you can check that your computer is up to scratch on the following page: At the time of writing this book, the following operating systems are supported by Unity for development: Windows XP (i.e., SP2+, 7 SP1+), Windows 8, and Mac OS X 10.6+. In terms of graphics card, most cards produced after 2004 should be suitable.

In terms of computer skills, all knowledge introduced in this book will assume no prior programming experience from the reader or knowledge of Unity. So for now, you only need to be able to perform common computer tasks, such as downloading items, opening and saving files, be comfortable with dragging and dropping items and typing, and be relatively comfortable with Unity's interface.

Who this Book is for

If you can answer yes to all these questions, then this book is for you:

Would you like to learn how to code in C#?
Would you like to know how to have fun creating video games while learning C#?
Would you like to discover more C# features that you can use to create video games with Unity?
Although you may have had some prior exposure to coding, would you like to delve more into C# and Object-Oriented Programming?

Who this Book is not for

If you can answer yes to all these questions, then this book is not for you:

Can you already create a full game in C#?
Are you looking for a reference book on C# programming?
Are you a professional C# developer?

If you can answer yes to all four questions, you may instead look for the other books in the series on the official website (http://www.learntocreategames.com).

How you will Learn from this Book

Because all students learn differently and have different expectations of a course, this book is designed to ensure that all readers find a learning mode that suits them. Therefore, it includes the following:

A list of the learning objectives at the start of each chapter so that readers have a snapshot of the skills that will be covered.
Each section includes an overview of the activities covered.
Many of the activities are step-by-step, and learners are also given the opportunity to engage in deeper learning and problem-solving skills through the challenges offered at the end of each chapter.
Each chapter ends-up with a quiz and challenges through which you can put your skills (and knowledge acquired) to the test. Challenges consist in coding, debugging, or creating new features based on the knowledge that you have acquired in the chapter.
The book focuses on the core skills that you need. While some sections go into more detail, once concepts have been explained, links are provided to additional resources, where necessary.
The code is introduced progressively and it is also explained in detail.
You also gain access to several videos that help you along the way, especially for the most challenging topics.

Format of each Chapter and Writing Conventions

Throughout this book, and to make reading and learning easier, text formatting and icons will be used to highlight parts of the information provided and to make the book easy to read.

Special Notes

Each chapter includes resource sections, so that you can further your understanding and mastery of Unity; these include:

A quiz for each chapter: these quizzes usually include 10 questions that test your knowledge of the topics covered throughout the chapter. The solutions are provided on the companion website.

A checklist: it consists of between 5 and 10 key concepts and skills that you need to be comfortable with before progressing to the next chapter.

Challenges: each chapter includes a challenge section where you are asked to combine your skills to solve a particular problem.

Author’s notes appear as described below:

Author’s suggestions appear in this box.

Code appears as described below:

```
public int score;
public string playersName = “Sam”;
```

Checklists that include the important points covered in the chapter appear as described below:

below: below: below: below: below: below: below: below: below: below: below: below:

How Can You Learn Best from this Book?

Talk to your friends about what you are doing.

We often think that we understand a topic until we have to explain it to friends and answer their questions. By explaining your different projects, what you just learned will become clearer to you.

Do the exercises.

All chapters include exercises that will help you to learn by doing. In other words, by completing these exercises, you will be able to better understand the topics and gain practical skills (i.e., rather than just reading).

Don’t be afraid of making mistakes.

I usually tell my students that making mistakes is part of the learning process; the more mistakes you make and the more opportunities you have for learning. At the start, you may find the errors disconcerting, or you may find that Unity does not work as expected until you understand what went wrong.

Challenge yourself.

All chapters include a challenge section where you can decide to take on a particular challenge to improve your game or skills. These challenges are there for you to think creatively and to apply the knowledge that you have acquired in each chapter using a problem-based approach.

Learn in chunks.

It may be disconcerting to go through five or six chapters straight, as it may lower your motivation. Instead, give yourself enough time to learn, go at your own pace, and learn in small units (e.g., between 15 and 20 minutes per day). This will do at least two things for you: it will give your brain the time to “digest” the information that you have just learned, so that you can start fresh the following day. It will also make sure that you don’t “burn-out” and that you keep your motivation levels high.

Feedback

While I have done everything possible to produce a book of high quality and value, I always appreciate feedback from readers so that the book can be improved accordingly. If you would like to give feedback on this book, you can email me at

Improving the Book

Although great care was taken in checking the content of this book, I am human, and some errors could remain in the book. As a result, it would be great if you could let me know of any issue or error you may have come across in this book, so that it can be solved and so that the book can be updated accordingly. To report an error, you can email me with the following information:

Name of the book.
The page or section where the error was detected.
Describe the error and what you think the correction should be.

Once your email is received, the error will be checked, and, in the case of a valid error, it will be corrected, and the book will be updated to reflect the changes accordingly.

Supporting the Author

A lot of work has gone into this book, and it is the fruit of long hours of preparation, brainstorming, and finally writing. As a result, I would ask that you do not distribute any illegal copies of this book.

This means that if a friend wants a copy of this book, s/he will have to buy it through the official channels (i.e., through Amazon or the book's official website:

If some of your friends are interested in the book, you can refer them to the book's official website where they can either buy the book, or join the mailing list to be notified of future promotional offers or enter a monthly draw and be in for a chance to receive a free copy of the book.

As an independent author, I pour my heart and soul into creating content that is both informative and engaging. Your feedback is not only a testament to the work I've done but also an essential component in helping others discover the book.

Reviews play a crucial role in the success of self-published authors. They offer potential readers a glimpse into the experiences of others and help them make informed decisions. A few words about what you enjoyed, what you learned, or how the book has helped you can go a long way. Positive reviews can boost the book's visibility on platforms like Amazon, making it more accessible to other aspiring game developers who can benefit from this resource.

So, if you find this book helpful, I kindly ask you to take a moment to leave a review on Amazon or your preferred online bookstore.

Writing a review doesn't have to be lengthy or complex. A few sentences about your favorite parts, the most valuable lessons you learned, or the projects you enjoyed the most can make a significant difference. Your review can inspire and encourage others to embark on their own

game development journey with the confidence that this book will be a worthwhile companion.

Thank you for your support.

[ ]

1

Installing Unity and Becoming Familiar with the Interface

This chapter helps you to progressively become familiar with Unity, the software that we will use to code in C# and to create video games. This chapter will explain and illustrate how to install this software, and how the different views and core features can be employed.

After completing this section, you should be able to:

Be more comfortable with Unity's interface.
Understand the role and location of the different views in Unity.
Know and use shortcuts to manipulate objects (e.g., move, scale, resize, duplicate, or delete) and move the view accordingly (e.g., pan or rotate).
Use the Inspector view.
Create and apply colors and textures to objects.
Create and combine simple built-in shapes.
Know how to search for and organize assets in your game efficiently.
Navigate through your scene and see it from both first- and third-person views.

If you need more help or information on the topics covered in this section, you can gain access to a FREE video training based on this book (i.e., 2-hour training) by using the following link: This training will show you exactly all of the steps covered in this book and may appeal to those who are more visual learners.

What is a game engine and should you use one?

Unity makes it possible to create video games without knowing some of the underlying technologies of game development, so that potential game developers only need to focus on the game mechanics and employ a high-

level approach to creating games using programming and scripting languages such as C# or JavaScript. The term high-level here refers to the fact that when you create a game with a game engine, you don't need to worry about how the software will render the game or how it will communicate with the graphics card to optimize the speed of your game. So, using a game engine would generally offer the following features and benefits:

Accelerated development: game engines make it possible to focus on the game mechanics. Because built-in libraries are available for common mechanics and features, these do not need to be rebuilt from scratch, and programmers can use them immediately and save time (e.g., for the user interface or the artificial intelligence).
Integrated Development Environment (IDE): an IDE helps to create, compile, and manage your code, and includes some useful tools that make development and debugging more efficient.

Graphical User Interface (GUI): while some game engines are based on libraries, most common game engines make it possible for users to create objects seamlessly and to perform common tasks such as transforming, texturing, and animating assets, through drag and drop features. Another advantage of such software is that you can understand and preview how the game will look without having to compile the code beforehand (e.g., through scenes).
Multi-platform deployment: with common game engines, it is possible to easily export the game that you have created to several platforms (e.g., for the web, iOS, or Android) without having to recode the entire game.

Advantages of using Unity

There are several game engines available out there. However, Unity has proven to be one of the best game engines. It has been used by game developers for several years and has been employed to produce successful 3D and 2D games. Several of these titles can be seen on Unity's website

With Unity, you can create 2D or 3D games and produce several types of game genres including First-Person Shooters (FPS), Massive Multiplayer Online Role-Playing Games (MMORPG), casual games, adventure games, and much more.

In addition to being able to create high-quality games with an easy-to-use interface, Unity makes it possible to export games to a wide range of platforms, including mobile platforms (e.g., Android, iOS, or Windows), Virtual Reality platforms (e.g., Oculus Rift, Google Cardboard, or PlayStation VR) or desktop platforms (e.g., Windows, Mac or Linux).

Unity includes all the necessary tools that you need to create great games and it also simplifies the application of useful techniques to improve the quality of your game. For example, it includes Visual Studio, an IDE that will help you to code faster, built-in Artificial Intelligence (AI) modules (e.g., NavMesh navigation) that you can use with no prior knowledge of AI, lights, built-in objects, or a finite state machine that you can apply to your characters for customized behaviors and animations.

Finally, to control the game, you can use high-level programming and scripting languages such as C#. This is useful for those who have already been exposed to this language to transfer their skills to game programming in Unity.

Downloading Unity Hub

Now that you have had an overview of Unity and game engines, it is time for us to start using Unity. However, before you can install and use Unity, you will need to download and install Unity Hub using the following steps:

Open the following link: This will help you to check that your computer complies with Unity’s requirements.
Once you have checked the requirements, we can download Unity Hub from the same.
Please follow the installation instructions.

Installing Unity

As mentioned in the previous section, you will need to install Unity Hub before you can install a version of Unity.

Once Unity Hub is installed, you can install a new version of Unity by going to the section called Installs and then selecting the option to Install

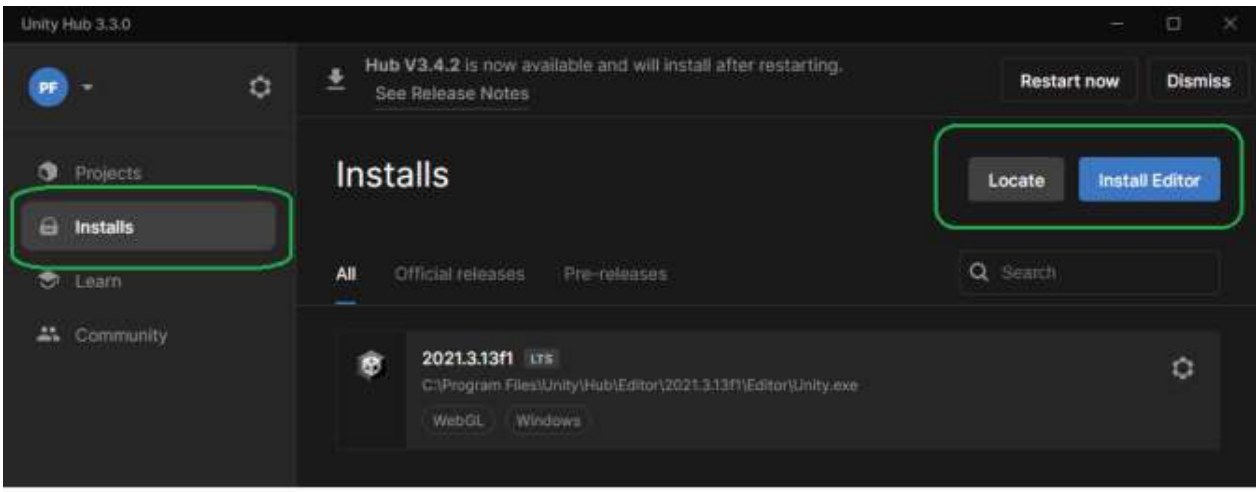


Figure 1-1: Installing Unity

You can then select the version of Unity that you need (for example, Unity 2021.3) as per the next figure.

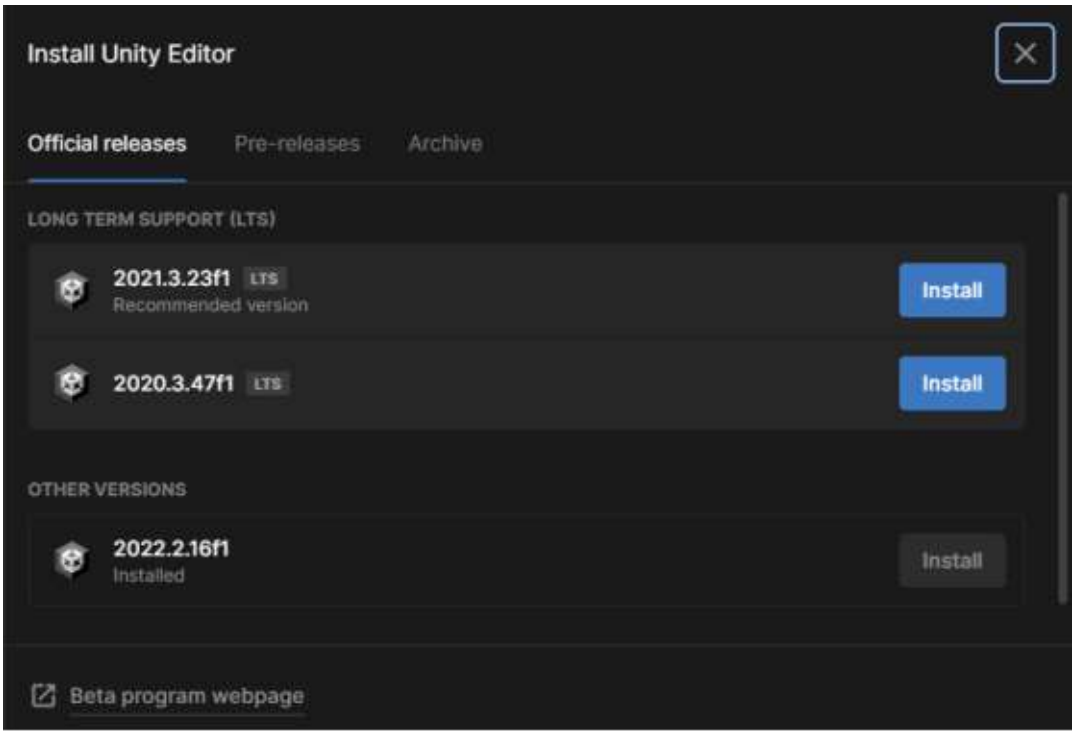


Figure 1-2: Adding a new version of Unity

You can press the button.
You will be asked whether you want to install additional modules. For the time being you can leave all the options as default (i.e., no additional modules).

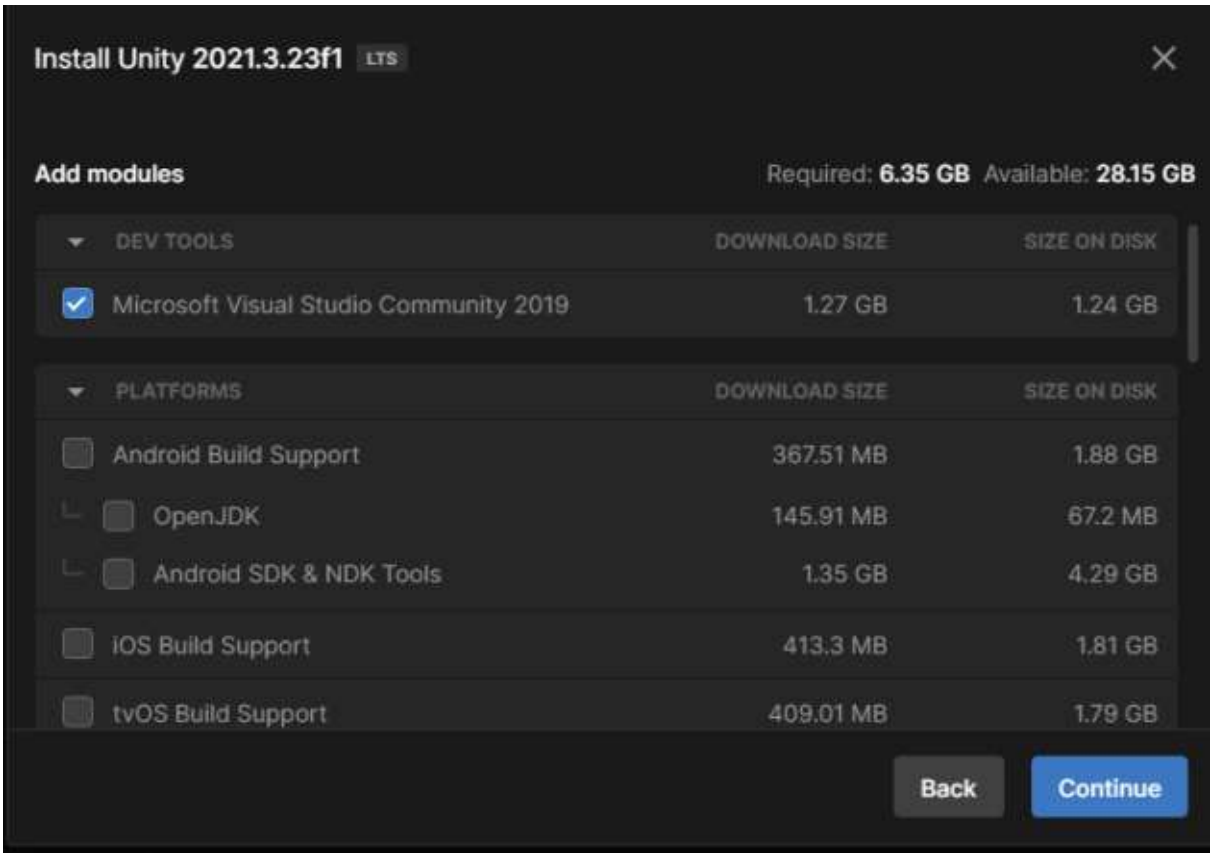


Figure 1-3: Selecting additional modules

Click on the button labelled and follow the instructions.
Once the new version of Unity is installed, it will be listed in Unity Hub in the section called

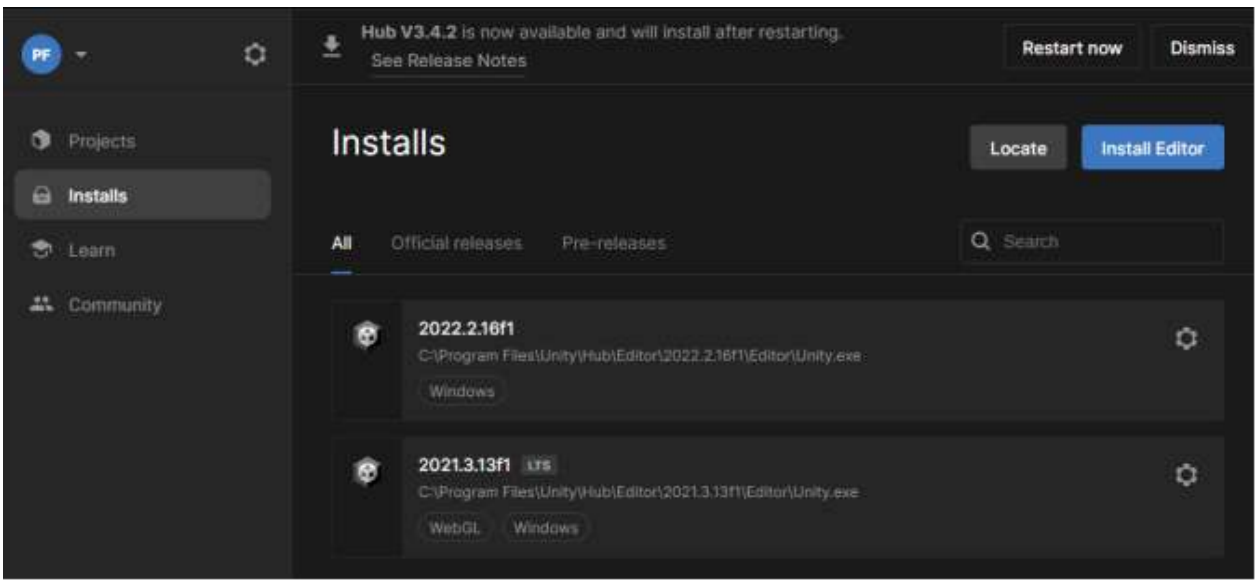


Figure 1-4: Checking that the new version has been installed

Launching Unity

Once you have successfully installed Unity and its components, we can now launch it through Unity Upon the first time you open Unity, you may need to provide your email address, so that you can receive regular updates from the Unity team. This should be really useful to keep up to date with major announcements for this software. You may also be asked whether you would like to activate the Pro version. However, for the purpose of this tutorial, you only need to use the free version (i.e., personal edition).

After having provided your email details as well as choosing the free version of the software, we can start to enjoy Unity.

You can now open a new project through Unity
In Unity please select the section called Projects from the left menu.

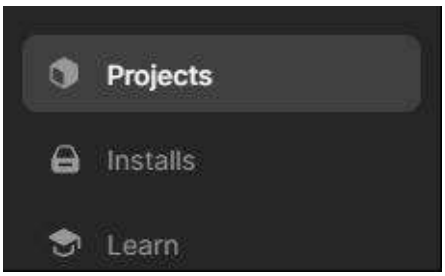
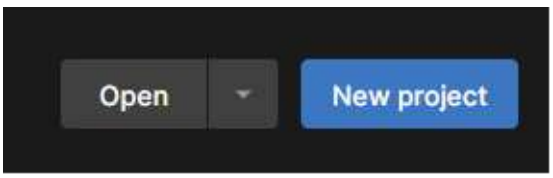


Figure 1-5: Selecting the Project tab

Click on the button labelled



In the new window, give a name to your project, for example

When Unity starts-up, a window labeled Unity Editor Update Check appears. This window, illustrated below, is there to check whether you have the latest version of Unity and to let you know of any recent updates available. If an update is necessary, you can install it. If you would prefer not

to see this message displayed every time you start Unity, you can uncheck the corresponding box labeled Check for Updates accordingly.

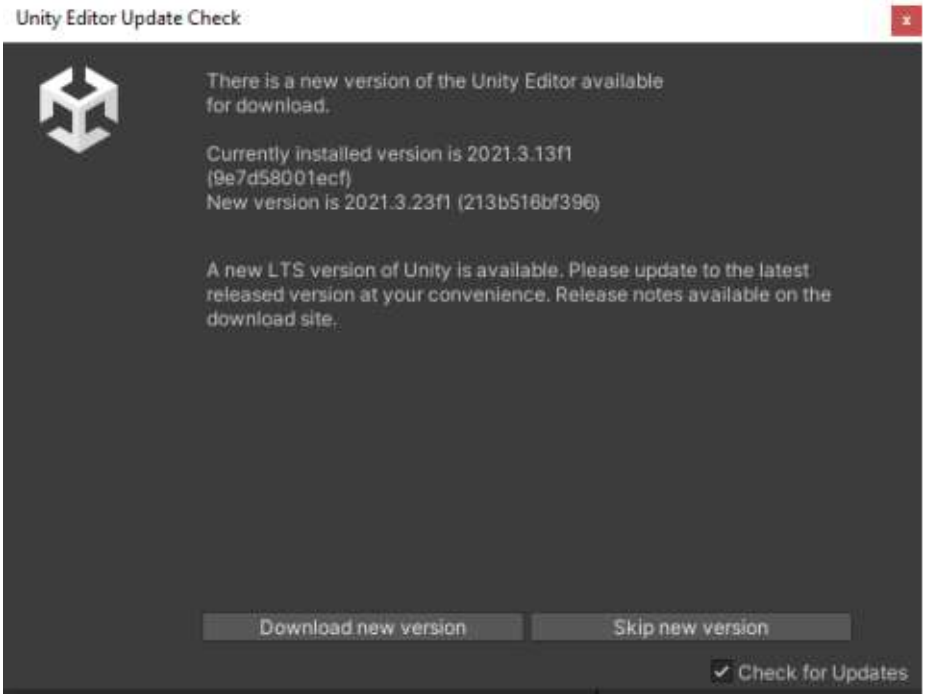


Figure 1-6: Automatically checking for Unity updates

Unity provides links to official forums and documentation from the main (i.e., top) menu: Help | Unity Forums or Help | Unity

Understanding and becoming familiar with the interface

After launching Unity, you will notice that it includes several windows organized in a (default) layout. Each of these windows includes a label in their top-left corner. These windows can be moved around and rearranged, if necessary, by either changing the layout (using the menu Window | Layouts | or by dragging and dropping the corresponding tab for a window to a different location. This will move the view (or window) to where you would like it to appear within the window. In the default layout, the following views appear onscreen (as described in the next screenshot, clockwise from the top left corner):

The Hierarchy window (the corresponding shortcut is this window or view lists all the objects currently present in your scene. These could include, for example, basic shapes, 3D characters, or terrains. This view also makes it

possible to identify a hierarchy between objects, and to identify, for example, whether an object has children or parents (we will explore this concept later).

If you are using Mac OS, then CTRL can be replaced by CMD.

The Scene view this window displays the content of a scene (or the item listed in the Hierarchy view) so that you can visualize and modify them accordingly (e.g., move, scale, etc.) using the mouse.
The Game view this window makes it possible to visualize the scene as it will appear in the game (that is, through the lens of the active camera).

The Inspector view this window displays information (i.e., the properties) on the object currently selected.
The Console window this window displays messages that are printed from the code by the user, or warnings and error messages related to your project or code displayed by Unity.
The Project window this window includes all the assets available and used for your project, such as 3D models, sounds, or textures.

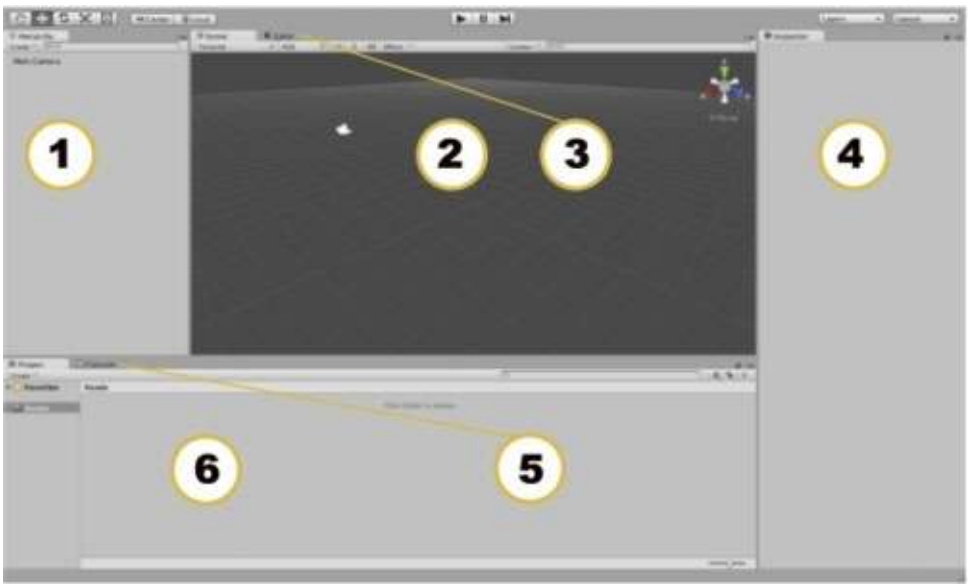
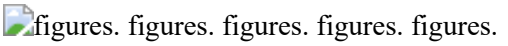


Figure 1-7: Main windows and views in Unity

The scene view

We will use this view to create and visualize the scene for our game. When you create a project, you can include several scenes within. A scene is comparable to a level, and scenes that are included in the same project can share similar resources, so that assets are imported once and shared across (or used in) all scenes. The Scene and Game views are displayed in the same window, and both are represented by a corresponding tab. By default, the Scene view is active. However, it is possible to switch to the Game view by clicking on the tab labeled For example, if we click successively on the Game and Scene tabs, we can see the view from both the perspectives of your eyes (i.e., the Scene view) and the active camera present in the scene (i.e., the Game view) as illustrated in the next figures.


figures. figures. figures. figures. figures.

Note that you can also rearrange the layout to be able to, for example, see both the Scene and Game views simultaneously. We could, for example, drag and drop the Game tab beside the Console tab to obtain the layout described on the next figure.

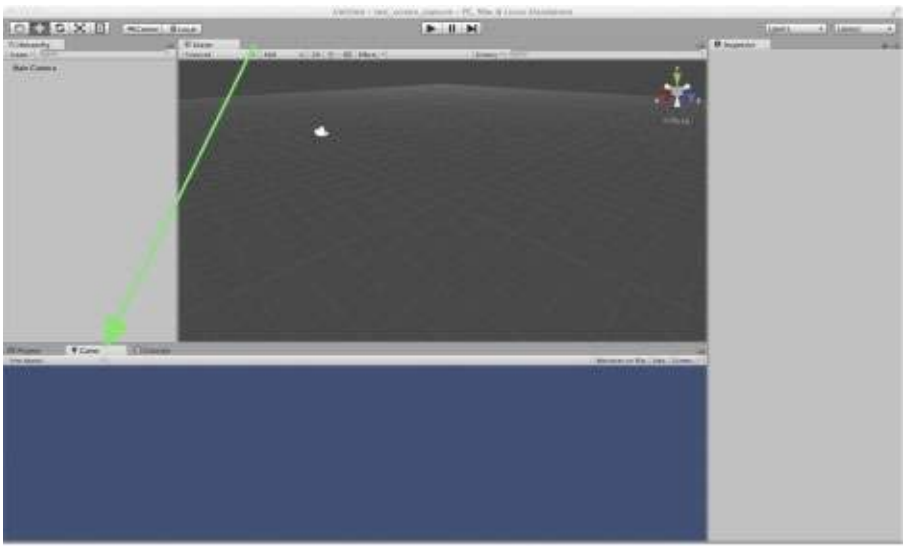


Figure 1-10: Changing the layout to display both Game and Scene views

In the previous figure, the Game tab that is usually to the right of the tab called Scene has been dragged to the right of the tab called This way, you can see both the Scene view and the Game view simultaneously.

Discovering and navigating through the scene

So that you can navigate easily in the current scene, several shortcuts and navigation modes are available. These make it possible to navigate through your scene just as you would in a First-Person Shooter or to literally “fly” through your scene. You can also zoom-in and zoom-out to focus on specific areas or objects, look around (i.e., using mouse look) or pan the view to focus on a specific part of the scene. The main modes of navigation are provided in the next table. However, we will look into these in more detail in the next section as we will be experimenting with them to explore (and modify) an existing scene.

Table 1: Navigation shortcuts

shortcuts shortcuts shortcuts shortcuts
shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts
shortcuts shortcuts shortcuts shortcuts shortcuts
shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts
shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts
shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts
shortcuts shortcuts shortcuts
shortcuts shortcuts shortcuts shortcuts

shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts shortcuts
shortcuts shortcuts shortcuts shortcuts shortcuts

For example:

In the default navigation mode, you can “walk” through the scene using the arrow keys (i.e., up, down, left and right).
In the “flight” mode, which can be activated by pressing and holding the Mouse Right Button (MRB), we can navigate using the S and D keys.

In the “flight mode”, we can also look around us by dragging the mouse or float up and down using the keys Q and

As you can see, both modes are very useful to navigate through your scene and to visualize all its elements. In addition, you can also choose to display the scene along a particular axis (i.e., x, y, or z) using the gizmo that is displayed in the top-right corner of the Scene view as described on the next figure.

Figure 1-11: Gizmo

The gizmo available in the Scene view includes three axes that are color-coded: x (in red), y (in green) and z (in blue). By clicking on any of these axes (or corresponding letters), the scene will be seen accordingly (i.e., through the x-, y-, or z-axis).

If you are not familiar with 3D axes: and z usually refer to the width and depth, while y refers to the height. By default, in Unity, the z-axis is pointing towards the screen if the x-axis is pointing to the right and the y-axis is pointing upwards. This is often referred as a left-handed coordinate system.

Also note that by clicking on the middle of the gizmo (white box), we can switch between isometric and perspective views.

In addition to the navigation tools, Unity also offers ways to focus on a particular object by rotating around a specific point (i.e., by pressing the ALT key and dragging the mouse to the left, right, up or down), or double-clicking on an object (i.e., in the Scene or Hierarchy view), so that the camera in the Scene view is focused on this object (this can also be achieved by selecting the object in either the Scene or Hierarchy view and by then pressing SHIFT+ or by zooming-in and out (i.e., scrolling the mouse wheel forward or back).

While the shortcuts and keys described in this section should get you started with Unity and make it possible for you to navigate through your scene easily, there are, obviously, many more shortcuts that you could use, but that will not be presented in this book. Instead, you may look for and find these in the official documentation that is available both offline (using the top menu: Help | Unity Manual then select the sections Working in Unity | The Main Windows | The Scene View | Scene View and online When using the documentation, you can also search for particular words as illustrated on the next figure.

Figure 1-12: Using Unity's manual

The hierarchy view

As indicated by its name, this view lists and displays the name of all objects included in the scene (in alphabetical order, by default) along with the type of relationship or hierarchy between them. You may notice that before you add any object to the scene, a camera is already present in the scene so that it can be viewed in the Game view through its lenses.

This view offers several advantages when we need to manage all the objects present in the scene quickly and to perform organizational changes. For example, we could use this view to find objects based on their name, to duplicate objects, to amend the name of objects, to amend the properties of several objects simultaneously, or to change the hierarchy between objects.

For example, on the following figure, we can see that the scene includes seven objects including a camera, a directional light, four cubes, and an object called

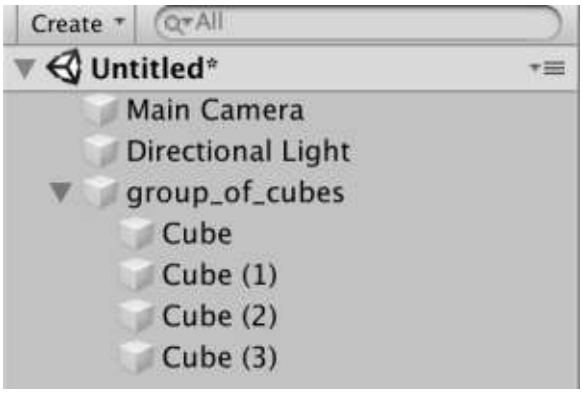


Figure 1-13: Creating a hierarchy between objects

We can also notice that all cubes are grouped under a “folder” (in Unity, this can be created as an empty object in the scene), which means that:

All four cubes are children of the object called
The object group_of_cubes is the parent of the four cubes.
If a transformation (i.e., scale or rotate) is applied to the parent (e.g., group of cubes) it will also be applied to the children (i.e., Cube(3) and

To change the hierarchy of the scene and make some objects children of a particular object, we only need to drag these objects atop the parent object.

The project view

This view includes and displays all the assets employed in your project (and across scenes), including: audio files, textures, scripts (e.g., scripts written in C#), materials, 3D models, scenes, or packages (i.e., zipped resources for Unity). All these assets, once present in the Project view, can be shared across scenes.

In other words, if we create a project and then a scene, and import assets for our game, these assets will be available from any other scene within the same project.

As for the Hierarchy view, built-in folders and search capabilities are included to ease the management of all your assets.

By default, the Project view includes two windows divided vertically (left and right columns). As illustrated on the next figure, the left window includes

a folder called assets and a series of “smart” folders (i.e., the content of these folders varies dynamically) called The right window displays the content of the folder selected on the left-hand side.

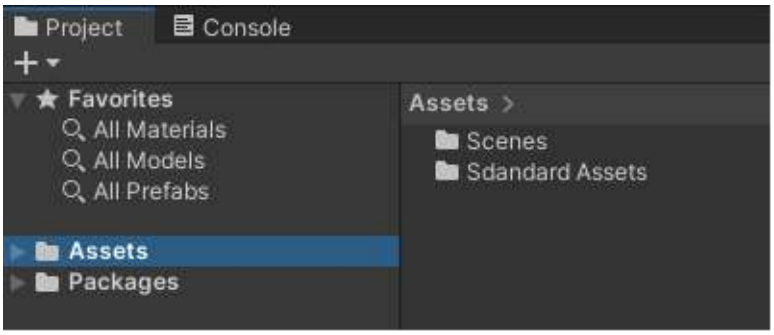


Figure 1-14: The project view

By clicking on any of the smart folders (e.g., All All or All Unity will filter the assets to display only the relevant ones accordingly (e.g., materials, models, or prefabs). This can speed-up the process of accessing specific assets and can be done, as for many of the functionalities present in Unity, in different ways. For example, you may notice a search window to the left of the Project view as illustrated in the next figure.

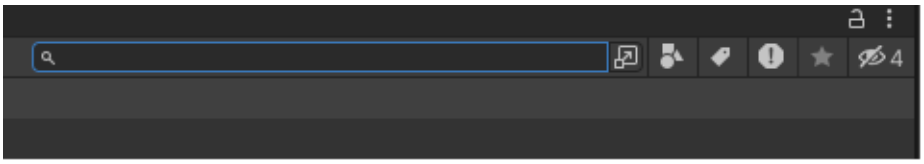


Figure 1-15: Searching for assets in the project

The search window in the Project folder can be used to search assets by their name or by their type, as illustrated in the next figure, by clicking on this icon .

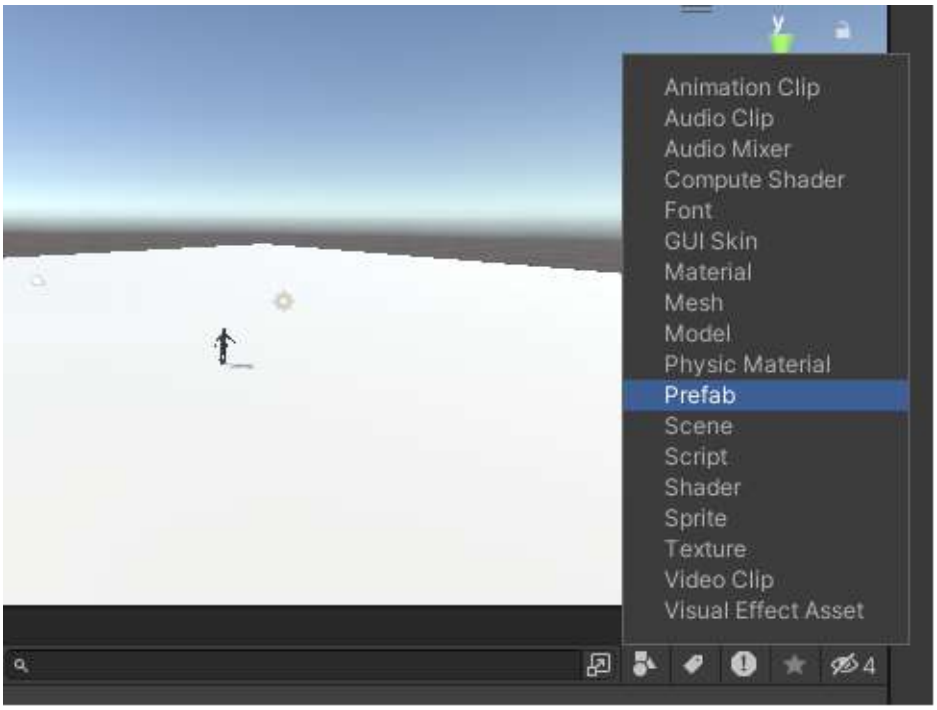


Figure 1-16: Filtering through the assets in the project view

As we can see on the previous figure, we have the option to select the type of assets that we are looking for (e.g., or Note that this option can also be specified by typing t: followed by the type that we are looking for in the search window. For example, by typing t:material in the search window, Unity will only display assets of type

The inspector

This window displays the properties of the object currently selected (i.e., the object selected in the Scene or the Hierarchy view) and it makes it possible to modify the attributes of an object accordingly. All properties are categorized in

By default, all objects present in the scene have a name, a default layer (we will look at this aspect later) and a component called However, it is possible to add components to an object using the button Add Component (see next figure) or the menu called

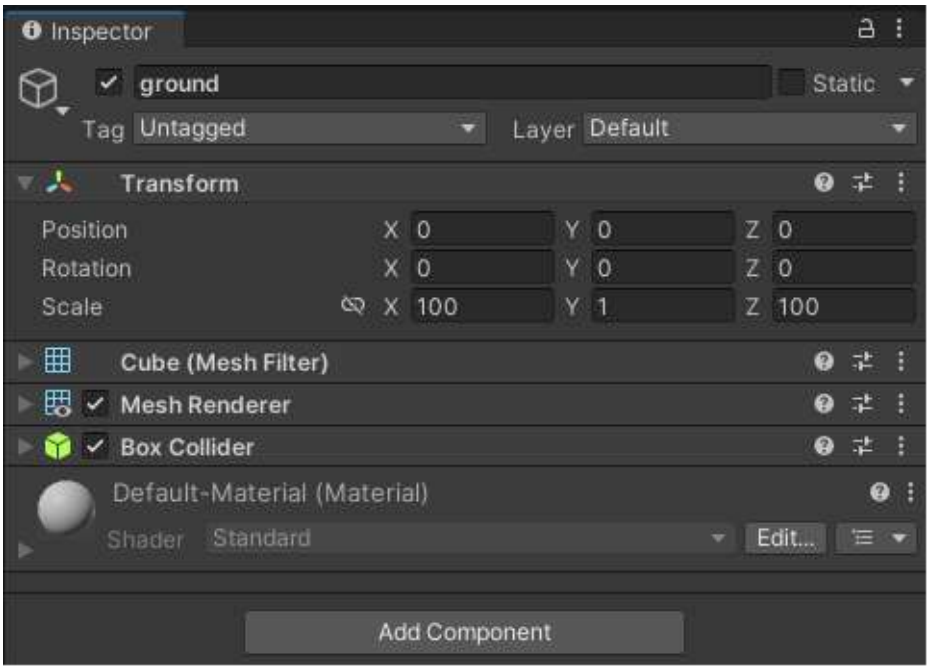


Figure 1-17: The Inspector window

You may also notice a tick box, in the Inspector window, to the left of the name of the object, that can be used to temporarily deactivate (and consequently reactivate) the object. This can be useful when you would like to temporarily remove an object from the scene without having to recreate it.

As we will see later, there are many types of components that can be added to an object to enhance it, including physics properties (to enhance how an object will behave realistically following the laws of physics), rendering (to enhance its appearance), or collision (to refine how it will detect collisions with other objects). For example, the default component Transform includes the position, rotation, and scale attributes of the object selected.

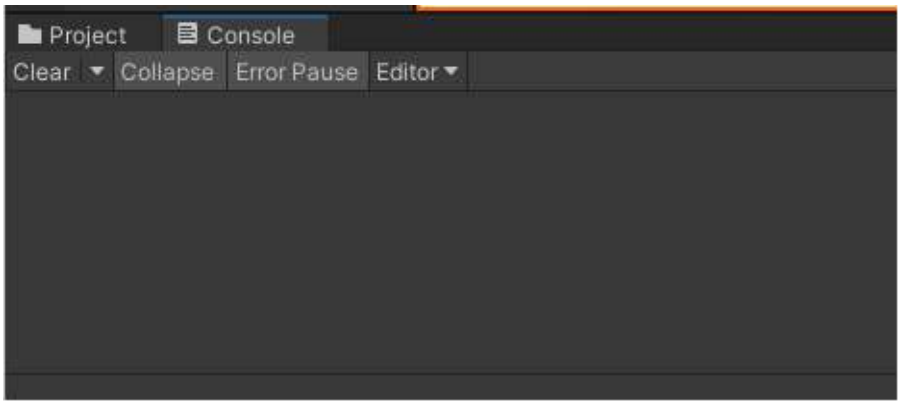
The attributes and while important, will be covered in later sections.

As we will see later, a scene can be edited and played. However, if we try to modify the attributes of an object while the game is playing, these will not be saved. In other words, for modifications to be saved in the scene, they have to be made while the game is stopped (i.e., not played).

The console view

As seen previously, the console window will display messages from Unity, related to possible errors and warnings in your code that may prevent the

game from playing, or messages that you can print through your own code (e.g., for debugging purposes).



The asset store window

This window, which is not displayed by default, connects you to the Asset an online repository and marketplace where you can search for and find free or premium assets for your game. This window can be accessed through the main menu (i.e., Window | Asset or by using the corresponding shortcut and will be displayed in your default browser.

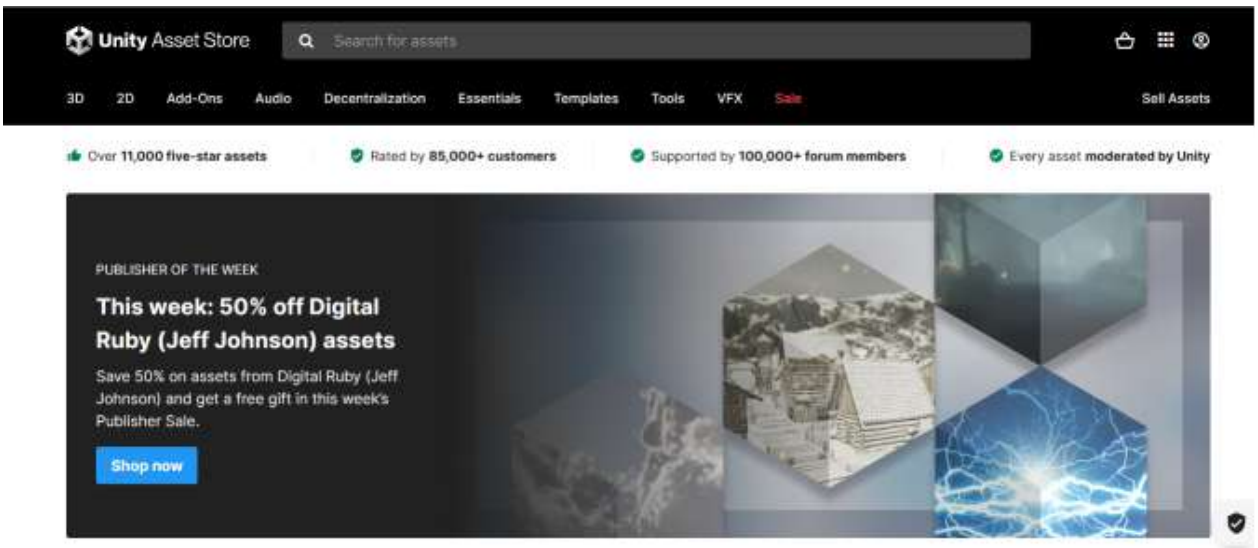


Figure 1-18: The Asset Store view

Level roundup

Summary

In this chapter, we have become familiar with the different views and windows available in Unity. We also looked at how to navigate through scenes and how to change the layout of our working environment. In the next chapter, we will harness these skills to be able to create and navigate through our own 3D environment.

Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist

Checklist Checklist

Quiz

It is now time to test your knowledge. Please specify whether the following statements are TRUE or FALSE. The answers are available at the end of the book.

The shortcut to open the Console window is CTRL + 1.
The shortcut to open the Project window is CTRL + 2.
The shortcut to open the Hierarchy window is CTRL + 4.
The Console window can display all the objects included in your scene.

The Project window can display messages or errors from your code.
Once an asset has been downloaded in the scene, it is not available in other scenes within the same project.
Once an object has been deactivated (i.e., using the tick box in the it will be deleted from your project forever.
To make some objects children of other objects, you can select the option: GameObject | Create

Unity is using a right-hand coordinate system.
Help on Unity is only available online (i.e., you need to be connected to the Internet to access it).

2

Introduction to C# programming

In this section, we will go through an introduction to C# programming and look at key aspects that you will need for your games, including:

C# Syntax.
Variable types and scope.
Useful coding structures (e.g., loops or conditional statements).

So, after completing this chapter, you will be able to:

Understand key concepts related to C# programming.
Understand the concepts of variables and methods.
Create a Flappy Bird game based on the concepts covered in this chapter.

The code solutions for this chapter are included in the resource pack that you can download by following the instructions included in the section entitled and Resources for this

Introduction

When you are using scripting in Unity, you are communicating with the Game Engine and asking it to perform actions. To communicate with the system, you are using a language or a set of words bound by a syntax that the computer and you know. This language consists of keywords, key phrases, and a syntax that ensures that the instructions are written and (more importantly) understood properly.

In computer science, this language needs to be exact, precise, unambiguous, and with a correct syntax. In other words, it needs to be

When writing C# code, you will be following a syntax; this syntax will consist in a set of rules that will make it possible for you to communicate with Unity clearly and unambiguously. In addition to its syntax, C# also uses classes, and your C# scripts will, by default, be saved as classes.

In the next section, we will learn how to use this syntax. If you have already coded in JavaScript, some of the information provided in the rest of this chapter may look familiar and this prior exposure to JavaScript will definitely help you. This being said, UnityScript and C#, despite some relative similarities, are quite different in many aspects; for example, in C#, variables and functions are declared differently.

When scripting in C#, you will be using a specific syntax to communicate with Unity; this syntax will be made of sentences that will be used to convey information on what you would like the computer to do; these sentences or statements will include a combination of keywords, variables, methods, or events; and the next section will explain how you can confidently build these sentences together and consequently program in C#.

Statements

When you code in C#, you need to tell the system to execute your instructions (e.g., print information) using statements. A statement is literally an order or something that you ask the system to do. For example, in the next line of code, the statement will tell Unity to print a message in the Console window:

print (“Hello Word”);

When writing statements, you will need to follow several rules, including the following:

The order of each statement is executed in the same order as it appears in the script. For example, in the next example, the code will print then world; this is because the associated statements are in that particular sequence.

```
print (“hello”);
print (“world”);
```

Statements are separated by semi-colons (i.e., semi-colon at the end of each statement).

Note that several statements can be added on the same line, as long as they are separated by a semi-colon.

For example, the next line of code has a correct syntax, as all of its statements are separated by a semi-colon.

Multiple spaces are ignored for however, it is good practice to add spaces around the operators or % for clarity. For example, in the next code snippet, we say that a is equal to You may notice that spaces have been included both before and after the operator

```
a = b;
```

Statements to be executed together (e.g., based on the same condition) can be grouped using code In C#, code blocks are symbolized by curly

brackets (e.g., { or So, in other words, if you needed to group several statements, you would include all of them within the same set of curly brackets, as follows:

```
{
print (“hello stranger!”);
print (“today, we will learn about scripting”);
}
```

As we have seen earlier, a statement usually employs or starts with a keyword (i.e., a word that the computer knows). Each of these keywords has a specific purpose, and common keywords, at this stage, could be used for the following actions:

Printing a message in the Console window: the keyword is
Declaring a variable: the keyword, in this case, depends on the type of the variable that is declared (e.g., int for integers, string for text, or bool for Boolean variables), and we will see more about these in the next sections.
Declaring a method: the keyword to be used depends on the type of the data returned by the method. For example, in C#, the name of a method is preceded by the keyword int when the method returns an it is preceded by the keyword string when the method returns a or by the keyword void when the method does not return any information.

What is called a method in C# is what used to be called a function in UnityScript; these terms (i.e., function and method) differ in at least two ways: in C# you need to specify the type of the data returned by this method, and the keyword function is not used anymore in C# for this purpose. We will see more about methods in the next sections.

Marking a block of instructions to be executed based on a condition: the keywords are if and
Exiting a function: the keyword is

Comments

In C# (similarly to JavaScript), you can use comments to explain your code and to make it more readable by others. This becomes important as the size of your code increases; and it is also important if you work in a team, so that other team members can understand your code and make amendments in the right places, if and when it is needed.

Code that is commented is usually not executed. There are two ways to comment your code in C# using either single- or multi-line comments.

In single-line comments, a double forward slash is added at the start of a line or after a statement, so that this line (or part thereof) is commented, as illustrated in the next code snippet.

```
//the next line prints Hello in the console window
print (“Hello”);
//the next line declares the variable name
string name;
name = “Hello”;//sets the value of the variable name
```

In multi-line comments, any code between the characters forward slash and star “ /*” and the characters star and forward slash “*/” will be commented, and this code will not be executed. This is also referred as comment

```
/* the next lines after the comments will print the message “hello” in the console window
we then declare the variable name and assign a value
*/
print(“Hello”);
string name;
name = “Hello”;//sets the value of the variable name
```

```
//print (“Hello World”)
/*
string name;
name = “My Name”;
*/
```

Variables

A variable can be compared to a container that includes a value that may change over time. When using variables, we usually need to: (1) declare the variable by specifying its type, (2) assign a value to this variable, and (3) possibly combine this variable with other variables using operators, as illustrated in the next code snippet.

```
int myAge;//we declare the variable myAge
myAge = 20;// we set the variable myAge to 20
myAge = myAge + 1; //we add 1 to the variable myAge
```

In the previous example, we have declared a variable called myAge and its type is int (as in We save the value 20 in this variable, and we then add 1 to it.

Note that, contrary to UnityScript, where the keyword var is used to declare a variable, in C# the variable is declared using its type followed by its name. As we will see later, we will also need to use what is called an access modifier in order to specify how and from where this variable can be accessed.

Also note that in the previous code, we have assigned the value myAge + 1 to the variable the = operator is an assignment operator; in other words, it is there to assign a value to a variable and is not to be understood in a strict algebraic sense (i.e., that the values or variables on both sides of the = sign are equal).

To make C# coding easier and leaner, you can declare several variables of the same type in one statement. For example, in the next code snippet, we declare three variables and v3 in one statement. This is because they are of the same type (i.e., they are

int v1,v2,v3;

int v4=4, v5=5, v6=6;

In the code above, the first line declares the variables and All three variables are In the second line of code, not only do we declare three variables simultaneously, but we also initialize them by setting a value for each of these variables.

When using variables, there are a few things that we need to determine including their name, their type and their scope:

Name of a variable: a variable is usually given a unique name so that it can be identified easily and uniquely. The name of a variable is usually referred to as an When defining an identifier, it can contain letters, digits, a minus, an underscore or a dollar sign, and it usually begins with a letter. Identifiers cannot be keywords, such as the keyword for example.

Type of variable: variables can hold several types of data, including numbers (e.g., integers, doubles or floats), text (e.g., strings or characters), Boolean values (e.g., true or false), arrays, objects (we will see the concept of arrays later in this chapter) or GameObjects (i.e., any object included in your scene), as illustrated in the next code snippet.

string myName = “Patrick”;//the text is declared using double quotes

int currentYear = 2017;//the year needs no decimals and is declared as an integer

float width = 100.45f;//the width is declared as a float (i.e., with decimals)

Variable declaration: variables need to be declared so that the system knows what you are referring to if you use this variable in your code. The first step in using a variable is to declare or define this variable. At the declaration stage, the variable does not have to be assigned a value, as this

can be done later. In the next example, we declare a variable called myName and then assign the value “My Name” to it.

```
string myName;
myName = “My Name”
```

Scope of a variable: a variable can be accessed in specific contexts that depend on where in the script the variable was initially declared. We will look at this concept later.

Accessibility level: as we will see later, a C# program consists of classes; for each of these classes, the methods and variables within can be accessed depending on their accessibility levels and we will look at this principle later.

Common variable types include:

same as text.
integer (1, 2, 3, etc.).
true or false.
with a fractional value (e.g., 1.2f, 3.4f, etc.).
a group of variables of the same type. If this is unclear, not to worry, this concept will be explained further in this chapter.
a game object (any game object in your scene).

Arrays

You can optimize your code with arrays, as they make it easier to apply features and similar behaviors to a wide range of data. When you use arrays, you can manage to declare less variables (for variables storing the same type of information) and to also access them more easily. You can create either single-dimensional arrays or multi-dimensional arrays.

Let’s look at the simplest form of arrays: single-dimensional For this concept, we can take the analogy of a group of 10 people who all have a name. If we wanted to store this information using a string variable, we would need to declare (and to set) ten different variables, as illustrated in the next code snippet.

string name1;string name2; .....

While this code is perfectly fine, it would be great to store this information in only one variable instead. For this purpose, we could use an array. An array is comparable to a list of items that we can access using an index. This index usually starts at 0 for the first element in the array.

So let’s see how we store the names with an array.

First we could declare the array as follows:

string [] names;

You will probably notice the syntax dataType [] The opening and closing square brackets are used to specify that we declare an array that will include string values.

Then we could initialize the array as follows:

names = new string [10];

In the previous code, we just specify that our new array, called will include 10 string variables.

We can then store information in this array as described in the next code snippet.

names [0] = “Paul”;

names [1] = “Mary”;

...

names [9] = “Pat”;

In the previous code, we store the name Paul as the first element in the array (remember the index starts at 0); we store the second element (with the index 1) as as well as the last element (with the index 9),

Note that for an array of size the index of the first element is 0 and the index of the last element is So for an array of size 10, the index for the first element is 0, and the index of the last element is 9 (i.e., 10-1).

If you were to use arrays of integers or floats, or any other type of data, the process would be similar, as illustrated in the next code snippet.

int [] arrayOfInts; arrayOfInts [0] = 1;

float [] arrayOfFloats;arrayOfLoats[0]=2.4f;

Now, one of the cool things that you can do with arrays is that you can initialize your array in one line, saving you the headaches of writing 10 lines of code if you have 10 items in your array, as illustrated in the next example.

string [] names = new string [10] {“Paul”,”Mary”,”John”,“Mark”, “Eva”,”Pat”,”Sinead”,”Elma”,”Flaithri”, “Eleanor”};

This is very handy, as you will see in the next chapters, and this should definitely save you a lot of time coding.

Now that we have looked into single-dimensional arrays, let’s look at multidimensional arrays, which can also be very useful when storing information. This type of array (i.e., multidimensional arrays) can be compared to a building with several floors, each with several apartments. So let’s say that we would like to store the number of tenants for each apartment. We would, in this case, create variables that would store this number for each of these apartments.

The first solution would be to create variables that store the number of tenants for each of these apartments with a variable that makes a reference

to the floor, and the number of the apartment. For example, the variable ap0_1 could be defined to store the number of tenants in the first apartment on the ground floor, could be defined to store the number of tenants in the second apartment on the ground floor, ap1_1 could be defined to store the number of tenants in the second apartment on the first floor, and could be defined to store the number of tenants in the third apartment on the first floor. So in term of coding, we could have the following:

```
int ap0_1 = 0;
int ap0_2 = 0;
...
```

However, we could also use arrays in this case, as illustrated in the next code snippet:

```
int [,] apArray = new int [10,10];
apArray [0,1] = 0;
apArray [0,2] = 0;
print (apArray[0]);
```

In the previous code:

We declare our array. [,] means a two-dimensional array with integers; in other words, we state that any element in this array will be defined and accessed based on two parameters: the floor level and the number of this apartment on that level.

We also specify a size (or maximum) for each of these parameters. The maximum number of floors (or level) will be 10, and the maximum number of apartment per floor will be 10. So, for this example we can define levels, from level 0 to level 9 (i.e., 10 levels), and from apartment 0 to apartment 9 (i.e., 10 apartments).

The last line of code prints the value of the first element of the array in the Console window.

One of the other interesting things with arrays is that, by using a loop, you can write a single line of code to access all the items in this array, and hence, write more efficient code.

Constants

So far we have looked at variables and how you can store and access them seamlessly in your code. The assumption then was that a value may change over time, and that this value would be stored in a variable accordingly. However, there may be times when you know that a value will remain constant throughout your game. For example, you may want to define labels that refer to values that should not change over time, and in this case, you could use constants.

Let's see how this works: let's say that the player has three choices in the first menu of the game, that we will call 0, 1, and 2. Let's assume that you would like an easy way to remember these values so that you can process the corresponding choices. Let's look at the following code that illustrates this idea:

```
int userChoice = 2;
if (userChoice == 0) print ("you have decided to restart");
if (userChoice == 1) print ("you have decided to stop the game");
if (userChoice == 2) print ("you have decided to pause the game");
```

In the previous code:

The variable userChoice is an integer and is set to
We then check the value of the variable userChoice and print a message accordingly in the console window.

Now, as you add more code to your game, you may or may not remember that the value 0 corresponds to restarting the game; the same applies to the other two values defined previously. So instead, we could use constants to make it easier to remember (and to use) these values.

Let’s see how the previous example can be modified to employ constants instead.

```
const int CHOICE_RESTART = 0;
const int CHOICE_STOP = 1;
const int CHOICE_PAUSE = 2;
int userChoice = 2;
if (userChoice == CHOICE_RESTART) print (“you have decided to restart”);
if (userChoice == CHOICE_STOP) print (“you have decided to stop the game”);
if (userChoice == CHOICE_PAUSE) print (“you have decided to pause the game”);
```

In the previous code:

We declare three constant variables.
These variables are then used to check the choice made by the user.

In the next example, we use a constant to calculate a tax rate; this is a good practice as the same value will be used across the program with no or little room for errors when it comes to using the exact same tax rate across your program.

```
const float VAT_RATE = 0.21f;
float priceBeforeVat = 23.0f
float priceAfterVat = pricebeforeVat * VAT_RATE;
```

In the previous code:

We declare a constant float variable for the vat rate.

We declare a float variable for the item’s price before tax.
We calculate the item’s price after adding the tax.

It is a very good coding practice to use constants for values that don't change across your program. Using constants makes your code more readable, it saves work when you need to change a value in your code, and it also decreases possible occurrences of errors (e.g., for calculations).

Operators

Once we have declared and assigned values to variables, we can then combine these variables using operators. There are different types of operators including: arithmetic operators, assignment operators, comparison operators and logical operators. So let's look at each of these operators:

Arithmetic operators are used to perform arithmetic operations including additions, subtractions, multiplications, or divisions. Common arithmetic operators include or % (modulo).

```
int number1 = 1;// the variable number1 is declared
int number2 = 1;// the variable number2 is declared
int sum = number1 + number2;// We add two numbers and store them in the variable sum
int sub = number1 - number2;// We subtract two numbers and store them in the variable sub
```

Assignment operators can be used to assign a value to a variable and include /= or

```
int number1 = 1;

int number2 = 1;
number1+=1; //same as number1 = number1 + 1;
number1-=1; //same as number1 = number1 - 1;
number1*=1; //same as number1 = number1 * 1;
```

number1/=1; //same as number1 = number1 / 1;

number1%=1; //same as number1 = number1 % 1;

Note that the = operator, when used with strings, will concatenate these strings (i.e., add them one after the other to create a new string). When used with a number and a string, the same will apply; for example "Hello"+1 will result in

Comparison operators are often used in conditional statements to compare two values; comparison operators include ==, <= and

if (number1 == number2); //if number1 equals number2

if (number1 != number2); //if number1 and number2 have different values

if (number1 > number2); //if number1 is greater than number2

if (number1 >= number2); //if number1 is greater than or equal to number2

if (number1 < number2); //if number1 is less than number2

if (number1 <= number2); //if number1 is less than or equal to number2

Conditional statements

Statements can be performed based on a condition, and in this case, they are called conditional The syntax is usually as follows:

if (condition) statement;

This means if the condition is verified (or true) then (and only then) the statement is When we assess a condition, we test whether a declaration is true. For example, by typing if (a == we mean "if it is true that a is equal to Similarly, if we type if (a>=b) we mean "if it is true that a is greater than or equal to b"

As we will see later, we can also combine conditions and decide to perform a statement if two (or more) conditions are true. For example, by typing if (a == b && c == 2) we mean "if a is equal to b and c is equal to

In this case, using the operator && means and that both conditions will need to be true. We could compare this to making a decision on whether we will go sailing tomorrow. For example, the weather is sunny and if the wind speed is less than 5km/h then I will go

We could translate this statement as follows.

if (weatherIsSunny windSpeed < 5) IGoSailing

When creating conditions, as for most natural languages, we can use the operator OR noted ||. Taking the previous example, we could translate the following sentence the weather is too hot or if the wind is faster than 5km/h then I will not go sailing ", as follows.

if (weatherIsTooHot || windSpeed >5) IGoSailing

Another example could be as follows.

if (myName == "Patrick") print("Hello Patrick");

else print ("Hello Stranger");

In the previous code:

We assess the value of the variable called

The statement print("Hello Patrick") will be printed if the value of the variable myName is

Otherwise, the message "Hello Stranger" will be displayed instead.

When we deal with combining true or false statements, we are effectively applying what is called Boolean Boolean logic deals with Boolean variables that have two possible values 1 and 0 (or true and false). By evaluating conditions, we are effectively processing Boolean numbers and applying Boolean logic. While you don't need to know about Boolean logic in depth, some operators for Boolean logic are important, including the ! operator. It means NOT (or "the opposite"). This means that if a variable is true, its opposite will be false, and vice versa. For example, if we consider the variable weatherIsGood = the value of

!weatherIsGood will be false (its opposite). So the condition if (weatherIdGood == false) could be also written if (!weatherIsGood) which would literally translate as “if the weather is NOT good”.

Switch Statements

If you have understood the concept of conditional statements, then this section should be pretty much straight forward. Switch statements are a variation on the if/else statements that we have seen earlier. The idea behind the switch statements is that, depending on the value of a particular variable, we will switch to a particular portion of the code and perform one or several actions accordingly. The variable considered for the switch structure is usually of type Let’s look at a simple example:

```
int choice = 1;


switch (choice)
{
case 1:
print ("you chose 1");
break;
case 2:
print ("you chose 2");
break;
case 3:
print ("you chose 3");
break;
default:
print ("Default option");
break;
}
print (“We have exited the switch structure”);
```

In the previous code:

We declare the variable called as an integer and initialize it to

We then create a switch structure whereby, depending on the value of the variable the program will switch to the relevant section (i.e., the portion of code starting with case case etc.). Note that in our code, we look for the values 2 or However, if the variable choice is not equal to 1 or 2 or 3, the program will go to the section called This is because this section is executed if all of the other possible choices (i.e., 1, 2, or 3) have not been fulfilled (or selected).

Note that each choice or branch starts with the keyword case and ends with the keyword The break keyword is there to specify that after executing the commands included in the branch (or the current choice), the program should exit the switch structure. Without any break statement we will remain in the switch structure and the next line of code will be executed.

So let's consider the previous example and see how this would work in practice. In our case, the variable choice is set to so we will enter the switch structure, and then look for the section that deals with a value of 1 for the variable This will be the section that starts with case then the command print ("you chose 1"); will be executed, followed by the command indicating that we should exit the switch structure; finally the command print ("We have exited the switch structure") will be executed.

Switch structures are very useful to structure your code and when dealing with mutually exclusive choices (i.e., only one of the choices can be processed) based on an integer value, especially in the case of menus. In addition, switch structures make for cleaner and easily understandable code.

Loops

There are times when you have to perform repetitive tasks as a programmer; many times, these can be fast forwarded using loops which are structures that will perform the same actions repetitively based on a condition. So, the process is usually as follows when using loops:

Start the loop.
Perform actions.
Check for a condition.

Exit the loop if the condition is fulfilled or keep looping otherwise.

Sometimes the condition is performed at the start of the loop, some other times it is performed at the end of the loop. As we will see in the next paragraph this will be the case for the while and do-while loop structures, respectively.

Let's look at the following example that is using a while loop.

```
int counter =0;
while (counter <=10)
{
counter++;
}
```

In the previous code:

We declare the variable counter and set its value to 0.
We then create a loop that starts with the keyword while and for which the content (which is what is to be executed while we are looping) is delimited by opening and closing curly brackets.
We set the condition to remain in this loop (i.e., counter So we will remain in this loop as long as the variable counter is less than or equal to 10.
Within the loop, we increase the value of the variable counter by 1 and print its value.

So effectively:

The first time we go through the loop: the variable counter is increased to we reach the end of the loop; we go back to the start of the loop and check if counter is less or equal to this is true in this case because counter equals 1.
The second time we go through the loop: counter is increased to we reach the end of the loop; we go back to the start of the loop and check if counter is less or equal to 10; this is true in this case because counter equals
...
The 11th time we go through the loop: counter is increased to we reach the end of the loop; we go back to the start of the loop and check if counter is less or equal to 10; this is now false as counter now equals As a result, we exit the loop.

So, as you can see, using a loop, we have managed to increment the value of the variable counter iteratively, from 0 to 11, but using less code than would be needed otherwise.

Now, we could create a slightly modified version of this loop, using a do-while loop structure instead, as illustrated in the next example:

```
int counter =0;
do
{
counter++;
} while (counter <=10);
```

In the previous example, you may spot two differences, compared to the previous code:

The while keyword is now at the end of the loop. So the condition will be evaluated (or assessed) at the end of the loop.
A do keyword is now featured at the start of the loop.

So here, we perform statements first and then check for the condition at the end of the loop.

Another variations of the code could be as follows:

```
for (int counter = 0; counter <=10; counter ++)
{
print (“Counter = ” + counter);
}
```

In the previous code:

We declare a loop in a slightly different way: we state that we will use an integer variable called counter that will go from 0 to 10.
This variable counter will be incremented by 1 every time we go through the loop.
We remain in the loop as long as the variable counter is less than or equal to 10.
The test for the condition, in this case, is performed at the start of the loop.

Loops are very useful to be able to perform repetitive actions for a finite number of objects, or to perform what is usually referred as recursive actions. For example, you could use loops to create (or instantiate) 100 objects at different locations in your game, or to go through an array of 100 items. So using loops will definitely save you some code and time :-).

Level Roundup

In this chapter, we have learned some concepts related to C# and Object-Oriented Programming. We also learnt how to define and to use classes, along with member variables and methods. Along the way, we also looked at other programming concepts such as variables, loops, and conditional statements. So, we have covered considerable ground to get you started with C#!

Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Quiz

It is now time to test your knowledge. Please specify whether the following statements are TRUE or FALSE. The answers are available at the end of the book.

The value of a variable always remains constant.
A method always returns information.
A method may not return information.
If a method is void, it will return an integer value.
An array can store several variables at a time.
A class usually includes a constructor.

A for loop can be used to go through all the elements of an array.

A public method is accessible from anywhere.
A private variable is accessible only from members of the class.
A protected variable is accessible only from members of the class.

3

Creating your First Script

In this section we will start to create C# scripts in Unity. Some of the objectives of this section will be to:

Introduce C# scripting in Unity.
Explain some basic scripting concepts.
Explain how to display information from the code to the Console window.
Illustrate how to create classes and to employ object-oriented principles.

After completing this chapter, you will be able to:

Understand and apply basic C# concepts.
Understand best coding practices.
Code your first script in Unity.
Create classes, methods and variables.
Instantiate objects based on your own classes.
Use built-in methods.
Use common structures such as conditional statements or loops.

You can skip this chapter if you are already familiar with C#, or if you have already created and used C# scripts within Unity.

Workflow to create a script

There are many ways to create and use scripts in Unity, but generally the process is as follows:

Create a new script using the Project view | C# the main menu | Create | C# or from an object via the Inspector (using the option Add
Attach the script to an object (for example, drag and drop the script on the object).
Check in the Console window that there are no errors in the script.
Play the scene.

By default, when you create your script, the name of the class within the script will be the same name as your script. So let's say that you created a new script called then the following code will be automatically generated.

```
using UnityEngine;
using System.Collections;
public class TestCode : MonoBehaviour
{
public void Start ()
{
}
public void Update ()
{
}
}
```

In the previous code, the class TestCode has been created; it inherits from the class MonoBehaviour and it includes two methods that can be modified: the methods Start and

You will also notice the two namespaces UnityEngine and As we have seen earlier, the keywords using is called a directive; in this particular case it is used to specify the namespaces and classes that will be used in our script.

If you would like to know more about namespaces and directives, you may look at the section called NameSpaces in the previous chapter.

How scripts are compiled

Whenever you create and save a script, it is compiled, and Unity will notify you (using the Console window) of any error. This being said, the order in which a script is compiled depends on its location.

First, the scripts located in the folders Standard Pro Standard Assets or Plugins are compiled, then the scripts located in the folders Standard Pro Assets/Editor or Plugins/Editor are compiled, then the scripts outside the Editor folder are compiled, followed by the scripts in the Editor folder. For more information on script compilation, you can check the official

Coding conventions

Quite often, coding conventions can provide increased clarity for your code and can be applied depending on the language that is being used.

Naming conventions usually employ a combination of Camel casing and Pascal

In Camel casing all words included in a name, except for the first one, are capitalized (e.g.,
In Pascal casing all words included in a name are capitalized (e.g., MyVariable).

When coding in C#, for example, naming conventions will use a combination of Camel and Pascal casing depending on whether you are naming a class, an interface, a variable or a resource.

However, as a C# beginner, in addition to learning about classes, methods, or inheritance, it may not be necessary to completely adhere to this naming convention at the start, at least as long as you use a consistent naming scheme throughout your code.

So, it is good to acknowledge different naming conventions linked to programming languages and to understand why they are in place; however, to keep things simple, this book will use a simplified naming convention, as follows:

Pascal casing for classes.
Camel casing for all methods and variables.

Once you feel comfortable with C# and when you want to know more about the official naming scheme for C#, you may look at Microsoft official naming

---

A few things to remember when you create a script (checklist)

As you create your first scripts, there will be, without a doubt, errors and possibly hair pulling :-). You see, when you start coding, you will, as for any new activity, make small mistakes, learn what they are, improve your coding, and ultimately get better at writing your scripts. As for my previous students, you will make mistakes; these don't make you a bad programmer; on the contrary, it is part of the learning process because we all learn by trial and error, and making mistakes is part of the learning process.

So, as you create your first script, please set any fear aside and try to experiment. Be curious and get to learn the language by practicing. It is like learning a new foreign language: when someone from a foreign country understands your first sentences, you feel so empowered! So, it will be the same with C#, and to ease the learning process, I have included a few tips and things to keep in mind when writing your scripts, so that you progress even faster.

You don’t need to know all of these by now (as I will refer to these tips later, in the next chapter), but just be aware of it and also use this list if any error occurs. This list is also available as a pdf file in the resource pack, so that you can print it and keep it close by. So, watch out for these tips :-).

- Each opening bracket has a corresponding closing bracket.

- All variables are written consistently (e.g., spelling and case). The name of each variable is case-sensitive; this means that if you declare a variable myvariable but then refer to it as myVariable later on in the code, this may trigger an error, as the variables myVariable and because they have a different case (upper- or lower-case are seen as two different variables.

- None of the local variables have the same name as some of the member variables.

- All variables are declared (type and name) prior to being used (e.g.,

- The type of the argument passed to a method is the type that is required by this method.

- The type of the argument returned by a method is the type that is required to be returned by this method.

- Built-in functions are spelt with the proper case (e.g., upper-case U for

• Use Camel casing (i.e., capitalize the first character of each word except for the first word) or Pascal casing (i.e., capitalize the first character of each word) consistently.

• All statements end with a semi-colon “;”.

• For if statements the condition is within round brackets.

• For if statements the condition uses the syntax rather than

• When calling a method, the exact name of this method (i.e., case-sensitive) is used.

• When referring to a variable, it is done with regards to (and awareness of) the access type of the variable (e.g., public or private).

• Local variables are declared and can be used within the same method.

• Member variables are declared outside methods and can be used anywhere within the class.

Common errors and their meaning

As you will start your journey through C# coding, you may sometimes find it difficult to interpret the errors produced by Unity in the console. However, after some practice, you will manage to recognize them, to understand (and also avoid) them, and to fix them accordingly. The next

list identifies the errors that my students often come across when they start coding in C#.

When an error occurs, Unity usually provides you with enough information to check the location of this error in your code, so that you can fix it. While many are relatively obvious to spot, some others are trickier to find. In the following paragraphs, I have listed some of the most common errors that you may come across as you start with C#. The trick is to recognize the error message so that you can understand what Unity is trying to tell you.

Again, this is part of the learning process, and you WILL make these mistakes, but as you recognize these errors, you will learn to understand them (and avoid them too :-)).

Again, Unity is trying to help you by communicating, to the best that it can, where the issue is with your code; so by understanding the error messages, we can get to fix these bugs easily. To make it easier to fix errors, Unity usually provides the following information when an error occurs:

The name of the script where the error was found.
The location of the error (i.e., row and column).
A description of the error.

So, if Unity was to generate the following error message...

“Assets/Scripts/MyFirstScript.cs (23,34) BCE0085: Unknown identifier: ‘localVariable’”

...it is telling us that an error has occurred in the script called at the line and around the 34th character (or the 34th column) on this line. In this particular message, it is telling us that it can’t recognize the variable

So, you may come across the following errors; this list is also available in the resource pack as a pdf file, so that you can print it and keep it close

by:

“;” This error could mean that you have forgotten to add a semi-colon at the end of a statement. To fix this error, just go to the line mentioned in the error message and ensure that you add a semi-colon at the end of the statement.
Unknown This error could mean that Unity does not know the variable that you are mentioning. It can be due to at least three reasons: (1) the variable has not been declared yet, (2) the variable has been declared but outside the scope of the method (e.g., declared locally in a different function), or (3) the name of the variable that you are using is incorrect (i.e., spelling or case). Remember, the names of all variables and functions are case-sensitive; so by just using an incorrect case, Unity will assume that you refer to another variable.

The best method overload for function ... is not This error is probably due to the fact that you are trying to call a function and to pass a list of parameters (which means the number and the types of parameters) that is not compatible with what the function is expecting. For example, the method described in the next code snippet, is expecting a String value for its parameter; so, if you pass an integer value instead, an error will be generated.

```
void mySecondFunction(string name)
{
print (“Hello, your name is” +name);
}
mySecondFunction(“John”);//this is correct
mySecondFunction(10);//this will trigger an error
```

Expecting } found ...: This error is due to the fact that you may have forgotten to either close or open curly brackets for conditional statements or functions, for example. To avoid this issue, there is a trick (or best practice) that you can use: you can ensure that you indent your code so that corresponding opening and closing brackets are at the same level. In the next example, you can see that the brackets corresponding to the start and the end of the method testBrackets are indented at the same level, and so are the brackets for each of the conditional statements within this function. By indenting your code, using several spaces or tabulation, you can make sure that your code is clear and that missing curly brackets are easily spotted.

```
void testBrackets()
{
if (myVar == 2)
{
print ("Hello World”);


myVar = 4;
}
else
{
}
}
```

Sometimes, although the syntax of your code is correct and does not yield any error in the Console window, it looks like nothing is happening; in other words, it looks like the code, and especially the methods that you have created do not work. This is bound to happen as you create your first scripts. It can be quite frustrating (and I have been there :-)) because, in this case, Unity will not let you know where the error is. However, there is a succession of checks that you can perform to ensure that this does not happen; so you could check the following:

The script that you have written has been saved.
The script contains no errors.
The script is attached to an object.
If the script is indeed attached to an object and you are using a built-in method that depends on the type of object it is attached to, make sure that the script is linked to the correct object. For example, if your script is using the built-in method which is used to detect collision between the FPSController and other objects, but you don't drag and drop the script on the FPSController object, the method OnControllerColliderHit will not be called if you collide with an object.

If the script is indeed attached to the right object and is using a built-in method such as or make sure that these functions are spelt properly (i.e., exact spelling and case). For example, for the method the system will call the method Update every frame, and no other function. So if you write a method spelt the system will look for the function instead, and since it has not been defined (or overwritten), nothing will happen, unless you specifically call this function from your code. The same would happen for the method In both cases, the system will assume that you have created two new functions update and

Best practices

To ensure that your code is easy to understand and that it does not generate countless headaches when trying to modify it, there are a few good practices that you can start applying as you begin with coding; these should save you some time along the line.

Variable naming

Use meaningful names that you can understand, especially to those not familiar with your code.

string myName =
    string b = “Patrick”;//NOT SO GOOD

Capitalize words within a name consistently (e.g., Camel or Pascal casing).

    bool testIfTheNameIsCorrect;// GOOD
    bool testifthenameiscorrect; // NOT SO GOOD

Methods

Check that all opening brackets have a corresponding closing bracket.
Indent your code.
Comment your code as much as possible to explain how it works.
Use the Start method if something just needs to be done once at the start of the scene.
If something needs to be done repeatedly, then the method Update might be a better option.

Debugging using dichotomy

In addition to providing explanations about your code, you can also use comments to prevent part of your code to be executed. This is very useful when you would like to debug your code and to find where the error or bug might be, using a very simple method called

By commenting sections of your code, and by using a process of elimination, you can usually find the issue quickly. For example, you can

comment all the code and run the script, then comment half the code, and run the script.

If the code works after this modification, it means that the bug that you are looking for is within the code that has just been commented; otherwise the error is probably in the code that has not yet been commented. Following this modification, we just need to comment half of the portion of the code where you think the error is.

So, by successively commenting more specific areas of our code, you locate the bug relatively quickly. This process is often called dichotomy (as we successively divide a code section into two). It is usually very effective to debug your code because the number of iterations (iteratively dividing parts of the code in two) is more predictable and also potentially less time-consuming. For example, for 100 lines of codes, we can successively narrow down the issue to 50, 25, 12, 6, and 3 lines. Between five and six iterations would be necessary in this case compared to checking 100 lines individually.

Creating a script

Now that we have gone through an overview of the best coding practices, it is time to create your first script; so let's get started!

Please launch Unity.
Create a new Project | New
Create a new scene | New

Let's create a new script:

In the Project window, click once on the Assets folder.
Create a new folder where you will be able to store your scripts (this is not compulsory but it will help to organize your scripts) by selecting Create | Folder from the Project window.

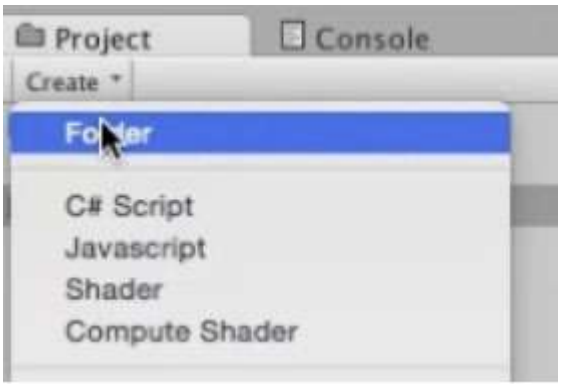


Figure 3-1: Creating a new folder

This will create a new folder labeled
Rename this folder
Double click on this folder to display its content and so that the script that we are about to create is added to this folder.
In the Project window, select Create | C#
This should create a new C# script.

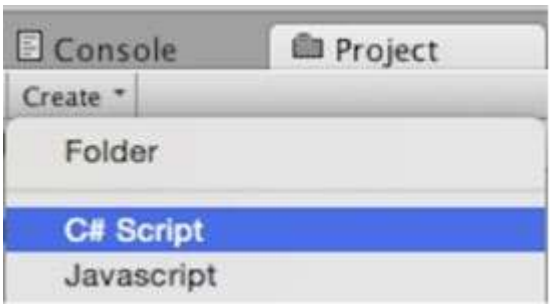


Figure 3-2: Creating a new C# script

By default, this script will be named However, the name of the script will be highlighted in blue so that you can modify it before its content is created. Please rename this script

Note that the name of the script should always match the name of the main class within the script; so if you want to rename this script later, you will also need to modify the name of the main class within the script.

Click once on the script and look at the Inspector window, and you will see the content of the script. By default, you will notice that it includes a definition for the class along with namespaces, as well as two different member methods Start and
Double-click on the script (within the Project window), this will open the script in MonoDevelop or Visual depending on the code editor that you have chosen for Unity.

Note that you can select the default code editor used in Unity by modifying Unity's preferences (i.e., Unity | Preferences | External Tools for Mac OS or Edit | Preferences...| External Tools for Windows).

The remaining of the examples provided in this book will be using Mono Develop; however, the actions and steps required should be similar if you are using another code editor (e.g., Visual Studio).

As you open the script, you should see the code that has been generated automatically by Unity. Again, the Start method is called at the start of each scene and the method Update is called every time the screen is refreshed (that is, every frame).
These functions are case-sensitive. Because they are built-in functions (i.e., functions made available by Unity for your use), Unity is expecting them to be spelt correctly, otherwise, it will assume that the method that you write serves a different purpose.

Writing your first statement
In this section, we will start by coding a simple statement.

So let’s start:

Please open the script called Game that you have just created.
Type the following code inside the Start function (new code in bold).

```
void Start()
{
print (“Hello World”);
}
```

In the previous code, we just create a statement that will print the sentence in the Console window. As the Start function is launched at the start of the scene, this statement should then be executed.
You can now save your code.
As your code is saved, you can switch back to Unity and check in the Console window for any error messages.

At this stage, when you have checked that the script is error-free, we just need to attach it to an empty object for the code to be executed.

Please create a new empty object | Create
Rename this object
Drag and drop the C# file Game from the Project window, to the object called game in the
If you click once on the object you should now see in the Inspector that it includes a component called Game which is the script that we have just linked to this object.

Now that the script and the object are linked, you can play the scene + P or Apple + and you should see the message in the Console window.

Using variables

In this section, we will start to use variables and combine them using statements and operators. So let's get started:

Please add the following code to the Start method.

```
//print ("Hello World");
int dateOfBirth = 1990;
int currentYear = 2017;
int age = currentYear - dateOfBirth;
print ("My age is " + age);
```

In the previous code:

We have commented the previous code using the two forward slashes; so this code will not be executed.
We declare two variables dateOfBirth and
We then declare a variable called

We perform a subtraction between the variable currentYear and the variable and we then save the result in the variable called
Finally, we print the text age is ", followed by the value of the variable called

Note that adding a number and a string is equivalent to concatenating these two, so the result of this addition will be a string.

You can now save your code, switch to Unity and play the scene; you should see the message age is in the Console window.

Creating methods

So at this stage we have managed to create two variables and to combine them using the operators + and in order to calculate and display the age based on a date of birth and the current year.

In this section, we will get to experiment with methods. So we will create two types of methods: a method that returns no data and that will be used to just display information onscreen, based on a parameter, and a second method that returns the age of an individual calculated using a date of birth and the current year.

So let’s get started.

Please open the script called
Add the following code just after the method Start (new code is in bold)

```
void Start ()
{

//print ("Hello World");
int dateOfBirth = 1990;
int currentYear = 2017;
int age = currentYear - dateOfBirth;
print ("My age is" + age);
}
void sayHello(string name)
{
print ("Hello " + name);
}
```

In the previous code:

We create a method called
This method is of type which means that it does not return any data.

This method takes one string parameter that will be referred to as name within the function
Within the function we define what should be done when the function is called: it will print a message onscreen that concatenates the string ” and the name passed as a parameter.

The last thing we need to do is to call this method from the Start function:

Please, add the following code to the Start function, just after the last print statement, as follows (new code in bold).

```
int age = currentYear - dateOfBirth;
print ("My age is" + age);

sayHello(“John”);
```

In the previous code:

We call the function sayHello and we pass the string John as a parameter; so this string will be referred as name within the function

You can now save your code, switch to Unity, and play the scene, and you should see the message in the Console window.

Modifying the scope of variables

In the previous section, we have created variables that were declared within the function This means that they are only accessible within this function. As such, they can be called local variables. However, you may want to make these variables accessible throughout your class. For example, we could create a variable called lastName that will be seen and used throughout the class as follows:

Please add the following code at the beginning of the class:

```
string lastName = “Blog”;
```

Modify the function called sayHello as follows:

```
void sayHello(string name)
{
//print ("Hello " + name);
print ("Hello " + name + " "+ lastName);
}
```

In the previous code, the variable lastName is added at the end of the new message.

If you save your code and play the scene, you should see the message appear in the Console window.

Note that, in this case, the variable because it is declared outside any method and therefore accessible throughout the class, is called a member variable.

Creating your first class

In the previous section, we created variables and methods. So, at this stage, you should be comfortable with both of these concepts.

We will now start to look at some simple Object-Oriented principles and create our own class. The idea here will be to create a class called Cube that will include most of the Object-Oriented concepts that we have seen so far, such as member variables, methods, constructors, and destructors. We will also consider and apply access levels to variables, and we will use other useful structures such as arrays and loops to make our code more efficient.

So let's get started:

First, from the Project view in Unity, create a new C# script and rename it Cube (i.e., Create | C#

This class will be used as a template to create objects of type

Open this script.

Replace the line

public class Cube : Monobehaviour

... with this line ...

public class Cube

This is because our new class will not extent the MonoBehaviour class.

Add the following lines of code at the beginning of the class (new code in bold).

```
public class Cube
{
private float speed;
private int color;
protected string name;
private float xPos, yPos, zPos;
In the previous code:
```

We declare several member variables name along with three float variables that will be used for the coordinates of the cube.
Next, we will create a constructor for this class. As we have seen previously, the constructor will be used when an object of the class Cube is instantiated (for example, when a new cube is created).
Please add the following code within the class

```
public Cube()
{
speed = 1.0f;
color = 2;
name = "Cube";
xPos = yPos = zPos = 0.0f;
Debug.Log("I am a new cube and my name is" + name);
}
In the previous code:
```

We set the values of the member variables and

We also set the values of the three float variables that will define the position of the
Lastly, we write the name of the new Cube object in the Console window.

So at this stage, we have defined the class We have also defined member variables and methods, as well as a constructor for this class. The latter will make it possible to create new Cubes from other scripts.

So now, we can start to create a new cube from the script called Game as follows:

Please save the code in the class called
Open the script called Game
At this stage this script (the script should already be linked to an empty object called
For clarity, you can, if you wish, comment any of the code that is already present in the functions Start or
Add the following code to the Start function.

```
Debug.Log ("Creating a new cube");
Cube cube = new Cube ();
```

In the previous code:

We write the message a new in the Console window.
We also create a new cube by calling its constructor.
You may notice the syntax:

```
className nameOfVariable = new className ();
```

You can now save your code and play the scene, and you should see the messages a new and am a new cube and my name is in the Console

window.

The reason for the second message is that by writing the following statement ...

Cube cube = new Cube ();

... we effectively call the default constructor from the class and we know that, amongst other things, this constructor will set and display the name of the new

Overloading our constructor

Now that we have created and instantiated a simple class, we will look at the concept of overloading. As you may recall, overloading a method means defining several methods with the same name but with different types of parameters; in our case, we will overload the constructor, so that, at instantiation time (that is, when we create a new object based on our class Cube), we have the choice to call different constructors, each of using different parameters to create a new object.

Overloading is part of the wider concept of polymorphism, where in our case, a method can take several (i.e., “poly”) shapes (i.e., “morph”).

Please add the following code to the class called Cube (i.e., just before the last curly bracket).

```
public Cube(int newColor)
{
speed = 1.0f;
color = newColor;


name = "Cube";
xPos = yPos = zPos = 0.0f;
Debug.Log("I am a new cube and my name is" + name);
Debug.Log("My Color Number is " + color);
}
```

In the previous code:

We define a new method called Cube that takes an integer parameter that will be referred as newColor within this method.
The member variable color is set with the value of the parameter called
As for the other Cube constructor, the member variables and zPos are set.

The last thing we need to do now is to modify the call to the constructor in the script called so that we instantiate a new Cube but using the constructor that we have just defined.

Please open the script called
Modify it as follows:

```
//Cube cube = new Cube ();
Cube cube2 = new Cube (2);
```

In the previous code:

We create a new cube called
We call a constructor and pass the value
Because a constructor is called and that an int parameter is passed, then the second constructor (which takes one integer parameter) will be called accordingly.

As a result, the second constructor is called, and the value 2 should be used to define the color number for the new cube.

Once this is done, you can save both scripts and run the scene. You should see a message saying “I am a new cube and my name is cube” followed by “My Color Number is 2” in the Console window.

Using constant variables

In this section, we will be using constant variables; if you remember, constant variables have a constant value; in our case, this will be used to associate a color number to the name of a color so that it is easier to remember and use; so we will be doing the following:

Define constant variables that will correspond to colors to be used for each of the new cubes.
Associate each variable with a name.
Use the constant variable when a cube is created to define its color.
Display the color of the cube in the Console window.

So let's get started:

Please add the following code at the beginning of the class

public const int COLOR_BLUE = 0, COLOR_RED=1, COLOR_YELLOW = 2, COLOR_GREEN = 3;

In the previous code:

We define four constant variables called and
Each of these colors is associated to a number, respectively and
These variables are public which means that they can be accessed from anywhere.

Next, we will create a function that will display the color of our cube; along the way, we will also use a conditional statement.

Please add the following method to the class

```
public void displayColor()
{
if (color == COLOR_BLUE)
Debug.Log ("My Color is blue");
else Debug.Log ("My color is not blue");
}
```

In the previous code:

We define a new method called
We check whether the color of the cube is blue (that its value is
If this is the case, we then print the message Color is
Otherwise, we print the message color is not

At this stage we just need to modify the script called Game so that we can create a new cube based on a constant variable (for the color of the cube) and display this color also.

Please open the script called

Add the following code to the Start function.

```
Cube cube3 = new Cube (Cube.COLOR_BLUE);
cube3.displayColor ();
```

In the previous code:

We create a new Cube variable called
We call the second constructor, passing an integer variable; however, this time, we use a constant variable (as it is easier to remember) to define its color.

We use the constant variable Cube.COLOR_BLUE to define the color of this cube. This is made possible because the variable although it belongs to the class is public, which means that we can access it from anywhere. We use the constant variable Cube.COLOR_BLUE as a parameter; you may note that the variable COLOR.BLUE could have been used without the need to instantiate a new cube.
We then call the method displayColor (which is public) to display the color of the cube.

You can now save your code and play the scene, and the console window should display the message saying “I am a new cube and my name is cube” followed by “My Color Number is

Constant and static variables

In the previous section, we have used constant variables to define the colors of the cubes that we have created. We could also have used static variables instead as these variables share similarities; however, they also differ in several ways:

Both static and constant variables are shared across instances of the same class.
Constant variables cannot be changed after they have been declared whereas static variable can be modified afterwards.
Constant variables cannot be set in a function, whereas static variables can.

Using the switch case structure

In this section we will use some interesting structures to make our code more efficient.

As we have seen in the previous section, we used the function displayColor to display the color of a cube; to do so, we used a conditional

statement to test the value of the color before displaying it in the Console window.

You may notice that if we wanted to test the four different colors, we would use four different conditional statements, but that may be unnecessary. Instead, we could use the switch case structure; if you remember, this structure allows to go to a specific part of the code based on the value of an integer; so we could use a switch structure to display a message that depends on the color of a cube.

So let's get started:

Please add the following code to the script

```
public void displayColor2()
{
switch (color)
{

case COLOR_BLUE: {Debug.Log ("My Color is blue");break;}
case COLOR_RED: {Debug.Log ("My COlor is red");break;}
case COLOR_YELLOW: {Debug.Log ("My COlor is yellow");break;}
case COLOR_GREEN: {Debug.Log ("My COlor is green");break;}
default: break;
}
}
```

In the previous code:

We define a public method called
We then create a switch structure based on the variable color; in other words: we will switch to a particular branch of our code based on the value of the variable

In our case, if the variable color equals to the value of the variable we write the message “My Color is In this particular case, all the instructions are written within curly brackets (although this is not compulsory), and the last instruction is this means that after displaying the message in the Console window, we will exit the switch structure; this ensures that all branches in the switch structure are mutually exclusive; in other words, we will use branch branch 2 or branch but we won’t be going through more than one branch at a time.
We proceed in a similar way for the colors red, yellow, and green.
The switch structure ends with the code:

```
default:break;
```

This means that if none of the branches have been executed, this default branch will be executed and we will then exit the switch loop.

You can now save your code. We will be testing it in the next section; however, if you’d like to perform a test now, you could just to modify the Game class as follows:

```
//cube3.displayColor().;
cube3.displayCOlor2();
```

Using arrays and loops

To be able to test the switch structure that we have created in the previous section, we will start to create four different boxes, each with a different color.

Please add the following code to the script called Game in the Start function.

```
Cube[] cubes = new Cube[4];
```

```
for (int i = 0; i <= 3; i++)
{
cubes [i] = new Cube (i);
cubes [i].displayColor2 ();
}
```

In the previous code:

We create an array of cubes; this array will include four
We then create a loop that goes from 0 to 3 using an increment of 1 (that is: 0, 1, 2, and 3).
Within this loop, we define each Cube that belong to the array called

For each of these Cubes we call the second constructor of the class and pass the value of the variable called i (i.e., 0, 1, 2, or 3) to define the color of the cube.
When this is done, we call the member method displayColor2 for each of these cubes, so that the corresponding color is displayed.

Now that this is done, and to be able to see the corresponding messages in the Console window, you can comment the following code in the Game script.

```
//Cube cube = new Cube ();
//Cube cube2 = new Cube (2);
//cube.displayColor ();
//Cube cube3 = new Cube (Cube.COLOR_BLUE);
//cube3.displayColor ();
```

You can also comment the following line in the first constructor of the class

```
//Debug.Log("My Color Number is " + color);
```

You can now save your code and play the scene; you should see the following messages in the Console window: “My Color is “My Color is

“My Color is and “My Color is

Now that this is working, there is a last thing that we could do to simplify our code: instead of using the switch loop, we could, instead, use an array of string variables that define the color of each of the cubes.

Let’s see how this can be done:

Please add the following code at the start of the class

```
private string [] colors = {"BLUE", "RED", "YELLOW",
"GREEEN"};
```

In the previous code, we define an array of string values called colors and we also initialize this array with the corresponding strings values.

Please add the following method to the class called

```
public void displayColor3()
{
Debug.Log("My Color is "+colors[color]);
}
```

In the previous code:

We define a public method called
In this method we display the color of the current cube.
To do so, we use the member variable color as an index to select the corresponding string in the array called For for example, if we have created a blue cube, its member variable color should be equal to so, the corresponding string would be located at the index 0 in the array called which corresponds to the string

The same applies to the other colors; the key here is to ensure that, when you define the array called you check that the order of the strings in the array (for example BLUE or corresponds to the name of the colors defined for the constant variables COLOR_BLUE (i.e., 0), COLOR_RED (i.e., 1) and so on and so forth.

Last but not least, you can modify the script Game as follows (new code in bold) so that the new function called displayColor3 is called instead of

```
for (int i = 0; i <= 3; i++)
{
cubes [i] = new Cube (i);
//cubes [i].displayColor2 ();
cubes [i].displayColor3 ();
}
```

After making these changes, you can save your code and play the scene; you should see the following messages in the Console window “My Color is “My Color is “My Color is and “My Color is

So as you can see, there are many ways to combine conditional statements, loops, and arrays to be able to optimize your code.

Instantiating visual objects in your scene

So far, the objects that we have created did not have any corresponding visual object in the scene view. So in this section, we will instantiate a new cube onscreen that corresponds to the cube that we have created in our script. This will be an interesting way to tie-in the concepts that we have looked at so far with Unity objects and primitives; we will also use the concept of inheritance.

So we will do the following:

Create a new constructor for the class Cube that accounts for the name, the position and the color of the cube; this is so that new cubes can be easily identified in the Scene view.
Create a new class called VisualCube that inherits from the class
Modify this new class so that, in addition to the member methods and variables inherited from its parent, this class creates a visual representation of the cube in the Scene view, based on its position, its color, and its name.

So let’s get started:

Please open the script called
Add the following code at the beginning of the class.

protected Vector3 position;
In the previous code:

We declare a Vector3 variable called
This variable is protected, which means that it will be accessible from the class Cube and its children.

Once this is done, we will create a new class called VisualCube as follows:

Please create a new C# script called
Open this script.
Modify the first line as follows:

public class VisualCube : Cube {
In the previous code:

We declare a new class called and we specify that it inherits from the class called

This means that it will inherit all its member methods (including the constructors) and member variables (for example, the variable

Next we will create a new constructor for this class. This constructor will be partially based on its parent, but it will also include some additional and more interesting features.

Please add the following code to the class.

public VisualCube(int newColor, Vector3 newPosition, string newName): base(newColor)
{
name = newName;
position = newPosition;
GameObject g = GameObject.CreatePrimitive(PrimitiveType.Cube);
g.name = name;
g.transform.position = position;
}
In the previous code:

We define a constructor for our new class.
The constructor will take three parameters: an int (the variable a Vector3 (the variable and a string (the variable

When the constructor is called, it will first call the constructor from the parent class (or the base class) and pass the parameter newColor to this constructor. This is performed thanks to the following code:

: base(newColor)

So by using this syntax, we use the constructor from the base class and also include additional features.

Then, we define the name of the cube as well as its position.
We create a new primitive (which is a cube) in the Scene view with the corresponding name and at the corresponding position.

Next, we just need to use this class in the script called

Please add the following code at the end of the Start function in the script called

```
VisualCube [] vCubes = new VisualCube [4];
for (int i = 0; i <= 3; i++)
{
vCubes [i] = new VisualCube (i, new Vector3(0.0f,i*1.0f,0.0f),
"Cube"+i);
}
```

In the previous code:

We create an array of VisualCube objects.
We then create a loop.
In this loop, each element of the array vCubes is instantiated, and we specify a new position and a new name for each of these objects through

the constructor.

As you play the scene, you will see that you have created four cubes in the Scene view, named and as per the next figures.

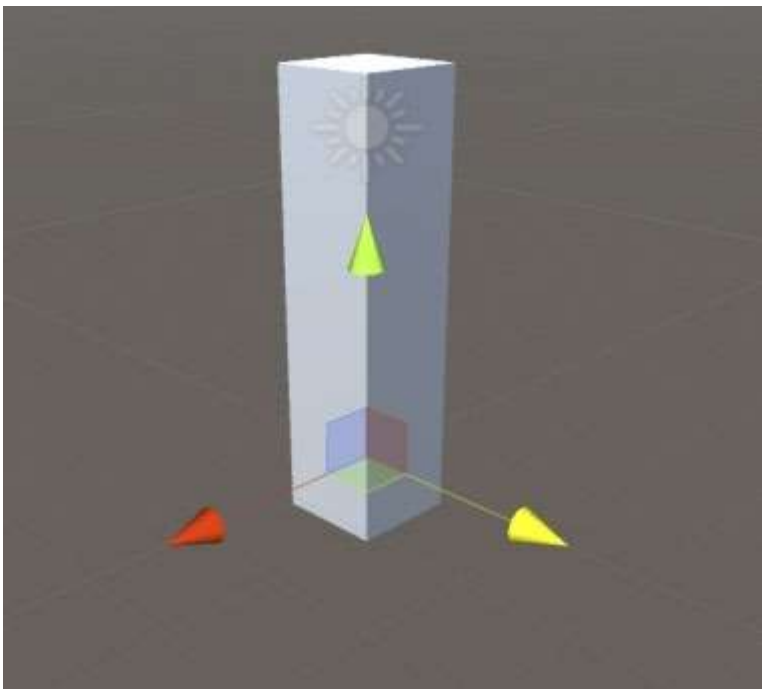


Figure 3-3: Creating VisualCube objects

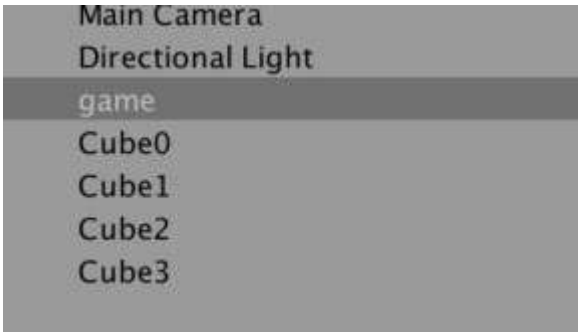


Figure 3-4: The VisualCube objects in the Hierarchy window

The last thing we could do is to allocate a color to all of these cubes.

Please open the script called

Add the following code at the end of the constructor.

```
    switch (newColor)
    {
    case Cube.COLOR_BLUE:  {g.GetComponent
().material.SetColor("_Color", Color.blue);break;}
    case Cube.COLOR_RED:  {g.GetComponent
().material.SetColor("_Color", Color.red);break;}
    case Cube.COLOR_YELLOW:  {g.GetComponent
().material.SetColor("_Color", Color.yellow);break;}
    case Cube.COLOR_GREEN:  {g.GetComponent
().material.SetColor("_Color", Color.green);break;}
    default: break;
    }
```

In the previous code:

As we have done in the past sections, we use a switch structure where we go to a specific branch based on the value of the variable
In each case, we access the Renderer component of the new object, and we then access its material attribute.
We then set the color of the Cube depending on the variable

If you save your code and run the scene, you should see, as illustrated in the next figure, four boxes, each with a different color, in the Game and Scene views.

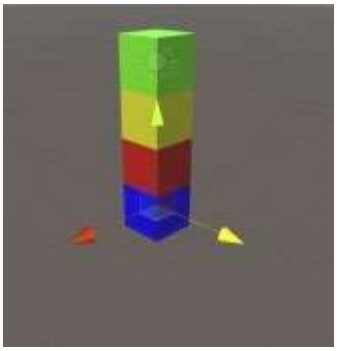


Figure 3-5: Four boxes created from code

Level Roundup
Well, this is it!

In this chapter, we have learned about some interesting Object Oriented concepts. We have created a class, along with a combination of variables, methods, loops and conditional statements. Along the way, we also used the concepts of constructors, overloading, overriding, and inheritance to create a new class called VirtualCube that inherited from the class Cube and that made it possible to display a cube onscreen based on its color. So, we have, again, covered some significant ground compared to the last chapter.

Checklist

Checklist Checklist

Quiz

It is now time to test your knowledge. Please specify whether the following statements are TRUE or FALSE. The answers are available at the end of the book.

In C# two member methods can have the same name.
When a variable is protected it can only be accesses from the base class.
When a variable is private it can only be accesses from the base class.
A switch case structure is similar to using if statements.
An array is better suited for data sets that don't expand overtime.
Lists are better used for data sets that expand overtime.
In each item can be accessed through its key.
A class can have several constructors.
Constructors have the same name as their class.
Overloading a constructor makes it possible to pass different parameters when a class is instantiated.

Challenge

Now that you have managed to complete this chapter and that you have created your first class, you can do the following:

Create a class called student based on the following code.

```
public class Student


{
public string firstName;
public string lastName;
public Student(string fName, string lName)
{
firstName = fName;
lastName = lName;
}
}
```

In the script called

Create a dictionary of students
Use their id as a key.
Display the name of one of the students based on its id.

4 Creating an Infinite Runner

In this section, we will start by creating an infinite runner game, a classic game genre on mobile devices; this game will have the following features:

The player will have to avoid obstacles by jumping above them.
Jumps will be performed when the player presses a key (for the web versions) or when s/he taps once on the right side of the device's screen (for the Android version).
The obstacles will come from the right side of the screen.
If the player hits one of the obstacles, the game will be restarted.
The player will have the option to pause the game.
The score will increase with time.

So, after completing this chapter, you will be able to:

Create a simple infinite runner.
Add simple controls to the main character.
Generate obstacles randomly.
Detect collisions.
Create a simple environment with basic shapes.

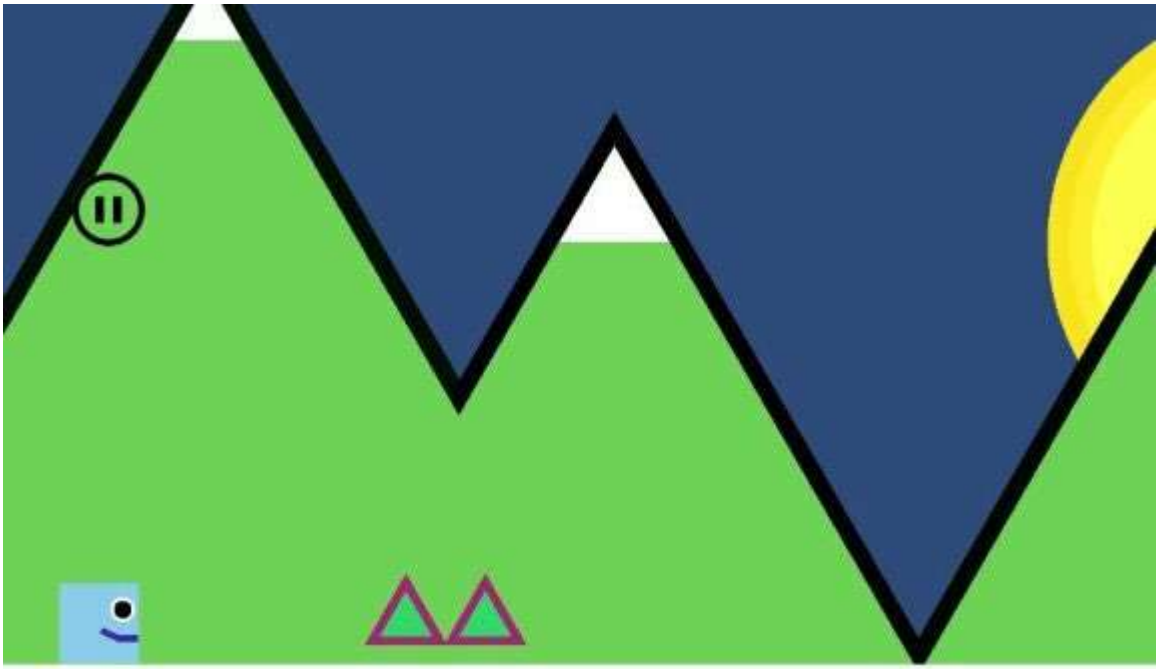


Figure 4-1: The final game

Adding movement to the character

So, in this section, we will start to create the core mechanics of the game; the environment will consist of a box for the player in addition to the ground.

So, let’s get started:

Please create a new scene. You can rename it infiniteRunner or any other name of your choice.

Once this is done, you can check that the 2D mode is activated, based on the 2D logo located in the top-left corner of the Scene view, as illustrated in the next figure.

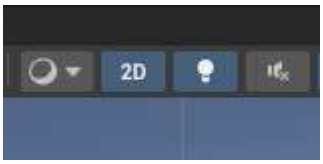


Figure 4-2: Activating the 2D mode

First, we will remove the background image for our If you look at your Game view, it may look like the following figure.

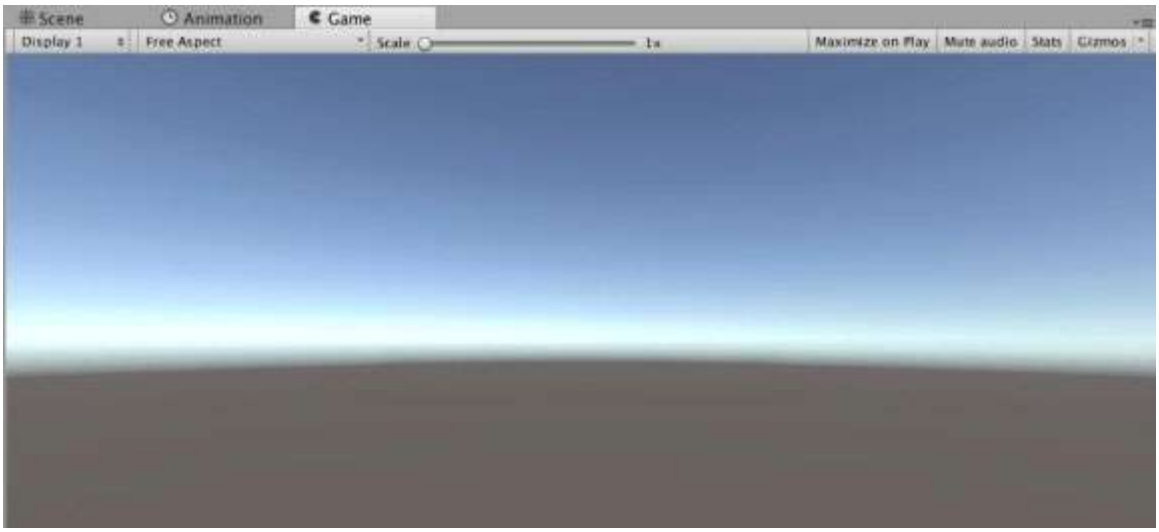


Figure 4-3: The initial background

If it is the case, then please do the following:

From the top menu, select: Window | Rendering | Lighting.
Then In the section delete the Default Skybox that is set for the attribute called SkyBox (i.e., click on the attribute to the right of the label Skybox and press DELETE on your keyboard).

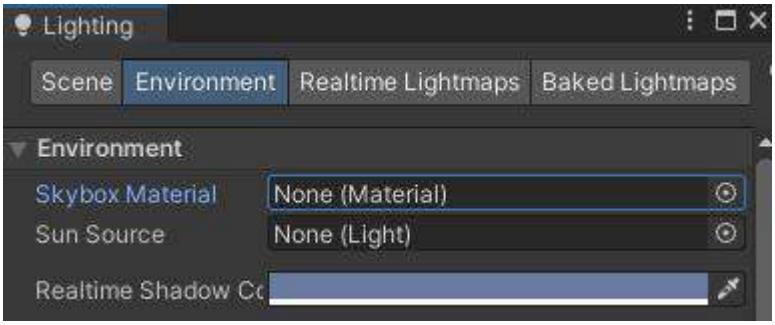


Figure 4-4: Lighting properties

Once this is done, your Game view should look like the following.

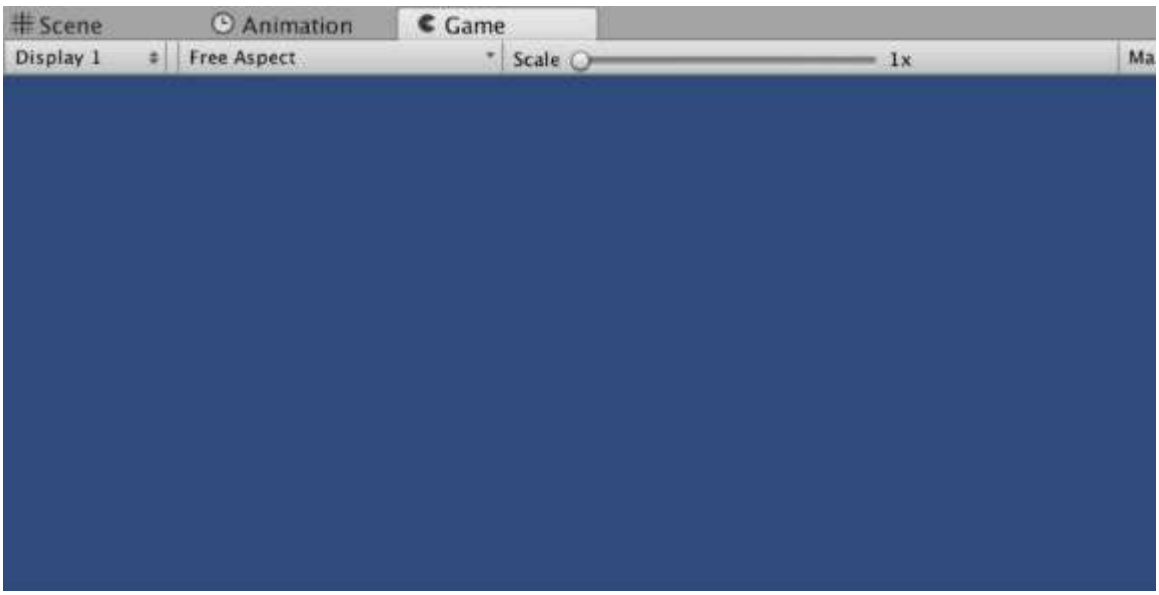
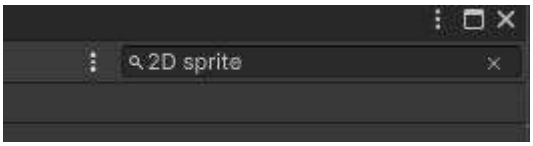


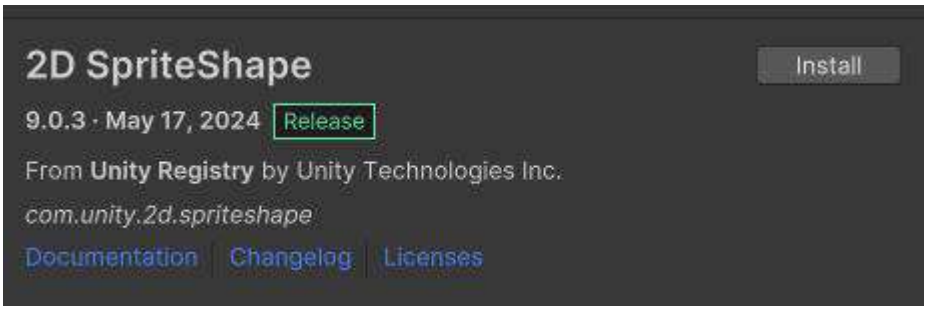
Figure 4-5: The Game view after deleting the SkyBox

Before we can create the items that will be part of our game, we need to import a 2D library:

Please open the Package Select Window | Package
Type “2D Sprite” in the search field



Select the package 2D Sprite Shape and click on



We will now create sprites that will be used for the ground and the player.

From the Project window, select Create | 2D | Sprites | This will create a sprite object called Square in the Project window.

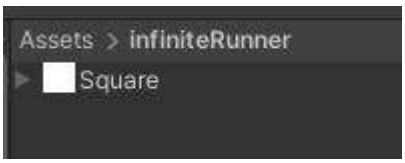


Figure 4-6: Creating a new sprite

Drag and drop this asset from the Project window to the Scene view.
This will create a new object in the Hierarchy window.
Please rename this object ground (i.e., right-click +
Select this new object in the and, using the change its scale attribute to (100, 1, 100) and its position to (0, -4,
Add a BoxCollider2D component to the object called ground (i.e., select Component | Physics2D | BoxCollider2D from the top menu); this is so that we can detect collisions between the player and the ground later-on.

Once this is done, we can then create the player.

Please duplicate the object called ground in the Hierarchy (i.e., select the object + press the keys CTRL and
Rename the duplicate
Using the change the scale of this new object to (1, 1, 1) and its position to (0, -3,
Add a RigidBody2D component to this object (i.e., select Component | Physics2D | RigidBody2D from the top menu).
Using the Sprite Renderer component of the object called change its color to a light blue.

By now, your scene should look like the following figure:

Figure 4-7: Drafting the scene

At this stage, we can start to add some code to be able to control our character. To do so we will do the following:

Check when the player is on the ground.
If this is the case and if the player presses the jump key (this key will be defined later), then the character will jump.
It will be necessary to check whether the player is on the ground so that the character can't jump while in the air.

So, let's proceed:

Please create a new C# script called from the Project window, select Create | C#
Please add the following code at the beginning of the class.

```
bool isOnGround;
```

This variable will be used to check whether the player character is on the ground.

Please add the following function to the class:

```
void OnCollisionEnter2D(Collision2D coll)
{
if (coll.collider.name == “ground”)
isOnGround = true;
else
isOnGround = false;
}
```

In the previous code:

We check whether a collision has been detected between the player and other objects using the built-in function
If the other object has a tag called ground then the variable called isOnground is set to otherwise, this variable is set to
Note that we still have to create a tag for the ground and this will be done in the next sections.

We can now deal with the key inputs from the player:

Please add the following function to the class:

```
void Jump()
{
GetComponent ().AddForce (new Vector2 (0, 400.0f));
isOnGround = false;
```

}

In the previous code:

We declare a function called

This function, when called, adds a vertical force to the player to simulate a jump.

It also sets the variable isOnGround to false.

Last but not least, please add the following code to the Update function (new code in bold):

void Update ()

{

if (Input.GetKeyDown (KeyCode.Space) && isOnGround) Jump();

}

In the previous code, we detect when the player has pressed the Space in this case, we check that the player is on the ground and we then call the function jump accordingly.

Please save your script, check that it is error-free, and drag and drop it on the object called player in the Hierarchy window.

At this stage, we just need to create a tag for the ground.

Please select the object called ground in the

In the Inspector window, click on the drop-down menu called to the right of the label called as illustrated in the next figure.

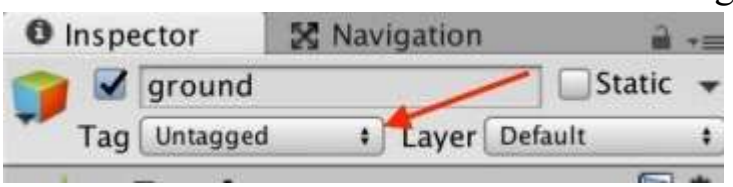


Figure 4-8: Creating a tag

Please select the tag called ground from the drop-down menu; if it is not present, then you can follow the next steps to create it:

Select the option Add Tag from the drop-down menu.

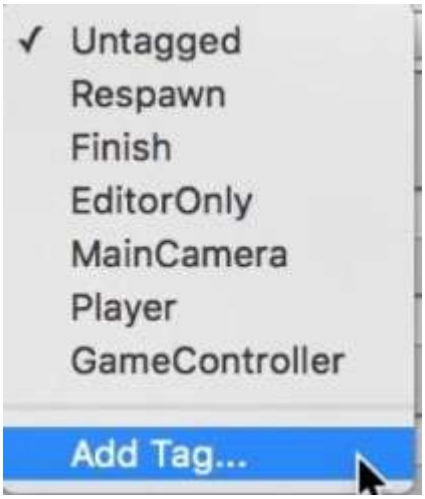


Figure 4-9: Selecting a new tag

In the new window, press on the + button that is just below all the other tags that you have already created, as illustrated in the next figure.

Figure 4-10: Creating a new tag (part 1)

This will create a placeholder for the new tag that we want to create. Create a new tag by entering ground to the right of the label called New Tag Name as illustrated in the next figure.

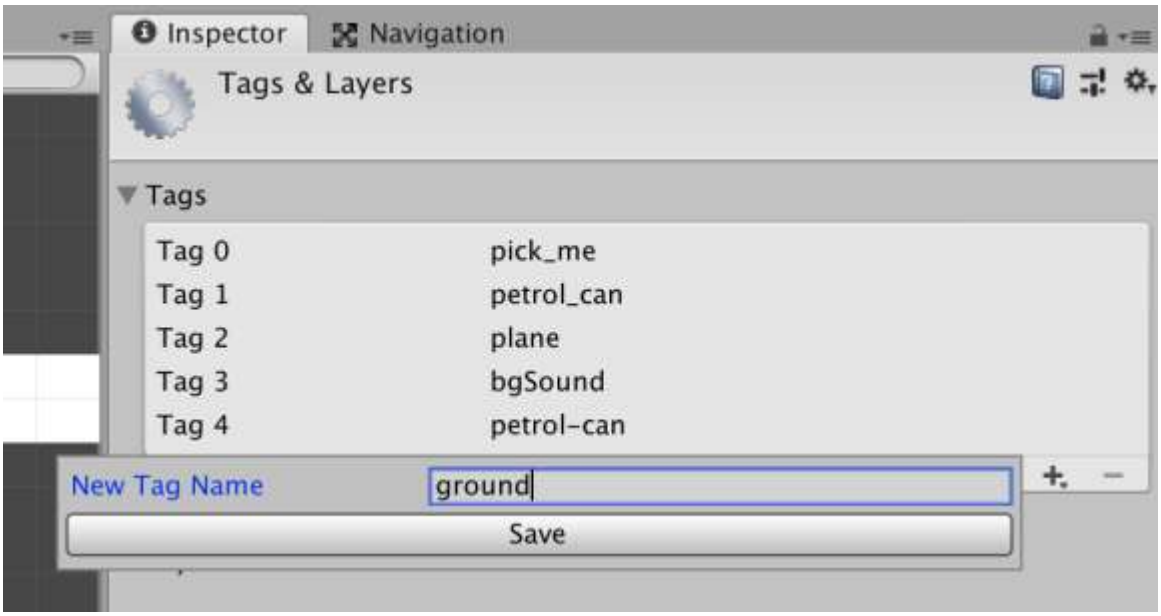


Figure 4-11: Creating a new tag (part 2)

Next, once this tag has been created, we can apply it to the ground object.

Please select the object called ground in the
Using the Inspector window, click to the right of the label called Tag and select the tag called ground from the drop-down menu, as illustrated on the next figure.

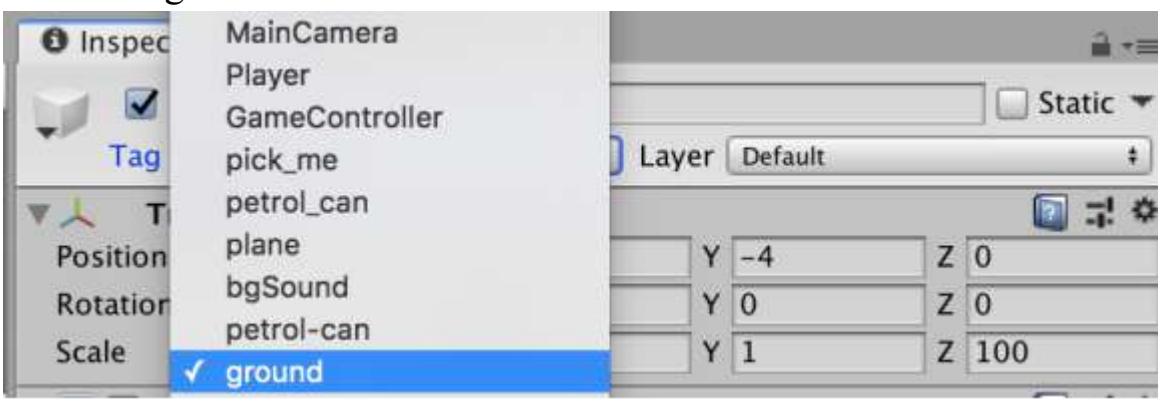


Figure 4-12: Selecting a tag for the ground (part 1)

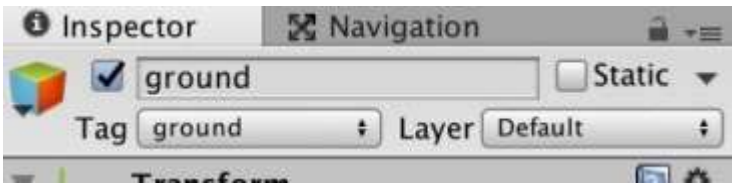


Figure 4-13: Selecting a tag for the ground (part 2)

Before we can test our cene, and so that the player is in te hcamera's field of view, we will modify the camera's position; so please select the object Main Camera and change its position to (0, 0,

You can now test the scene (i.e., CTRL + as you press the Space you should see that the player is jumping.

Figure 4-14: The first jump

Adding random obstacles to the scene

At this stage, we can make the player character jump, and we just need to add obstacles that will be generated at regular intervals; this will consist in:

Creating an obstacle from a basic shape.
Creating a prefab from this object.
Creating an empty object, with an associated script, that will generate these prefabs at frequent intervals.

Each new obstacle will be instantiated to the right of the player (i.e., outside the field of view) and will then start to move to the left (i.e., towards the player).

First let’s create this obstacle:

Please duplicate the object called ground (i.e., CTRL +
Rename the duplicate
Change its position to (3, -3, 0) and its scale attribute to (1, 1,
Using this object’s Sprite Renderer component, change its color to red.
Finally, as we have done earlier, please create a tag called and apply it to this object.

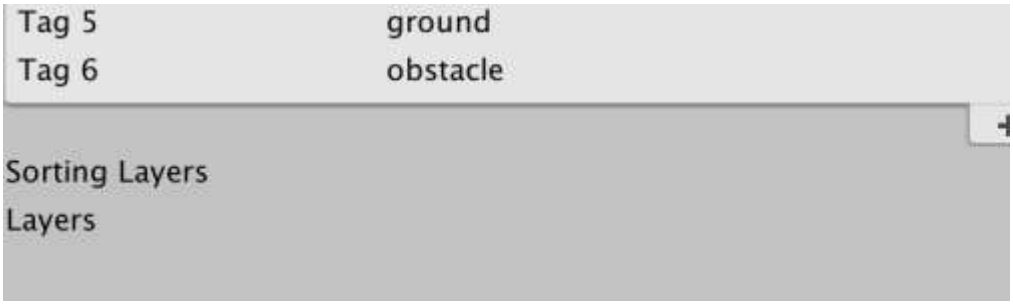


Figure 4-15: Adding a new tag for obstacles (part 1)

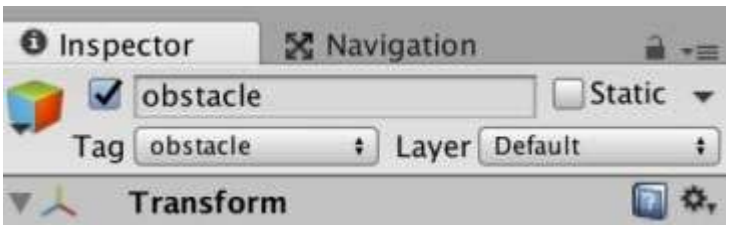


Figure 4-16: Adding a new tag for obstacles (part 2)

We can now create a prefab from this object by dragging and dropping it to the Project window, as illustrated in the next figure.

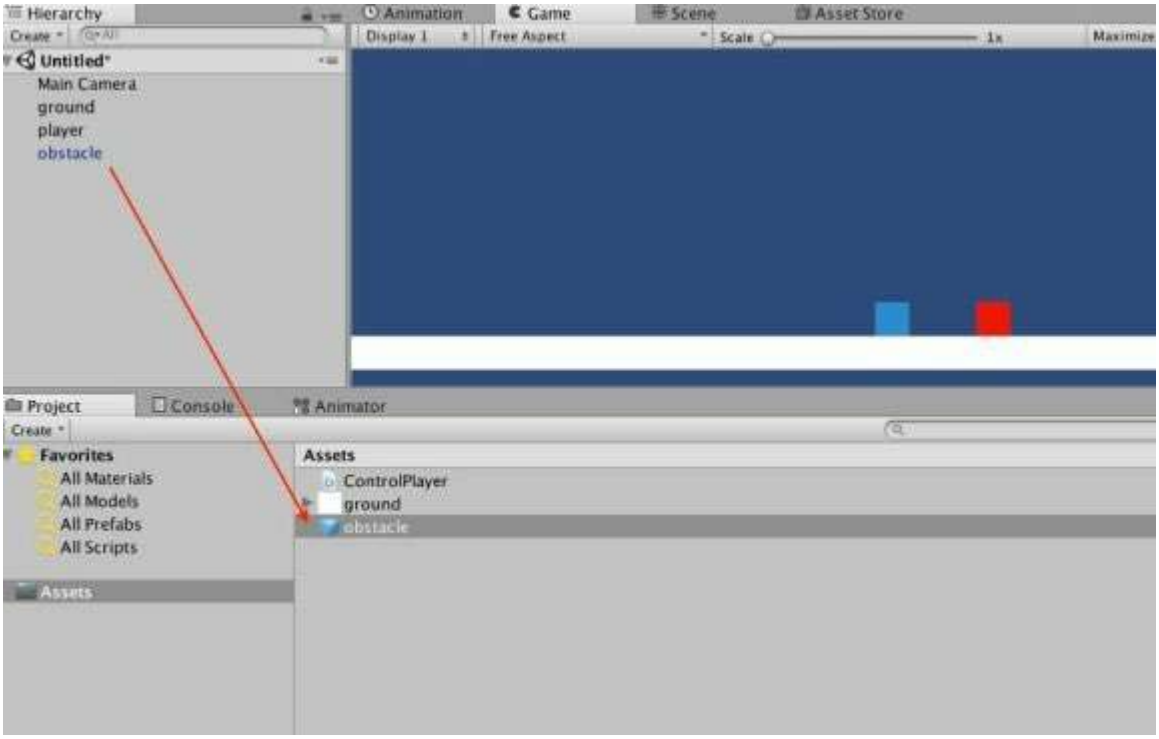


Figure 4-17: Creating a new prefab

Now that the prefab has been created, you can delete (or deactivate) the object called obstacle in the

We will now create a new script for this object (i.e., so that this object starts to move to the left every time it has been instantiated.

Please create a new C# script called Obstacle (i.e., from the Project window select: Create | C#
Add the following code to the Update function (new code in bold).

```
void Update ()
{
transform.Translate (Vector2.left * 4*Time.deltaTime);
if (transform.position.y < -5) Destroy (gameObject);
}
```

In the previous code:

We move the obstacle at the speed of 4 units per second to the left.
We also destroy this object once its y coordinate is less than note that we could also use the function Destroy in the Start function, to delay the destruction of this object.

Please add this script to the obstacle prefab:

Select the obstacle prefab.
Click on the button labelled Open Prefab in the
Drop the script Obstacle to the Inspector so that the script effectively becomes a component of the prefab
Click on the left arrow to the right of the label obstacle for thisprefab, so that you can go back to the regular Hierarchy view.

Figure 4-18: Closing the prefab

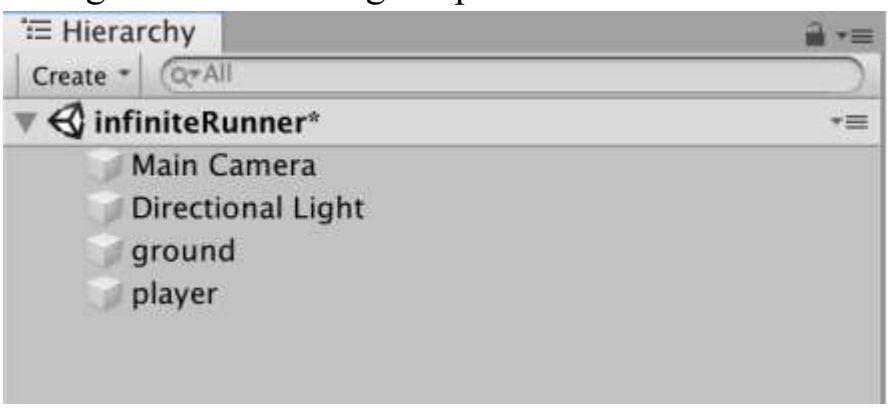


Figure 4-19: The regular Hierarchy view

Next, we need to create an object (and its associated script) that will generate the obstacle(s) at regular intervals.

Before creating the next object, please make sure that you have exited the prefab editing mode, otherwise the new object will be added to the prefab and not to the scene.
Please create a new empty object called generateObjects (i.e., select GameObject | Create
Create a new C# script called GenerateObjects (i.e., from the Project window, select: Create | C#
Open this script.
Add the following code at the beginning of the class.

```
public GameObject obstacle;
float timer;
```

In the previous code, we declare a GameObject variable called obstacle that is public; so that it will be accessible from the It will be set with the prefab called obstacle later-on, using the We also create a variable called timer that will be employed to time the instantiation of the obstacles.

Please add the following code to the Update function (new code in bold).

```
void Update ()
{
ManageTimer ();
}
```

In the previous code, we call a function ManageTimer that we yet have to create, and that will be in charge of checking when a new obstacle should be instantiated.

Please add the following function to the class:

```
void ManageTimer()
{
timer += Time.deltaTime;
if (timer >= 2)
{
AddObstacle ();
timer = 0;
}
}
```

In the previous code:

We define a function called
In this function, the timer's value is increased by one every seconds.
If the timer reaches two seconds, then the function AddObstacle (that we yet have to create) is called;
The timer is also reset to

Please add the following function to the class:

```
void addObstacle()
{
Vector3 positionOfPlayer = GameObject.Find
("player").GetComponent().initialPosition;
GameObject t1;
t1 = (GameObject)(GameObject.Instantiate (obstacle, positionOfPlayer
+ Vector3.right * 20, Quaternion.identity));
}
```

In the previous code:

We define a function called
This function initially detects the position of the player at the start of the game; since the player is initially on the ground, this will ensure that the obstacle will be created just above the ground.
Note that we access a variable called initialPosition that we yet have to create in the script
This position, stored in the variable is then used to determine the position of the obstacle that will be located to the right of the player.

Please save your script.

Last but not least, we need to initialize the variable initialPosition for the script

Please open the script
Add the following code to it (new code in bold):

```
public Vector3 initialPosition;
void Start ()
{
initialPosition = transform.position;
}
```

In the previous code:

We declare a variable called
We then set this variable to the position of the player at the start of the game.
Note that the variable initialPosition is public, so it will be accessible from outside the script, including from the script called

Please save your scripts; check that they are error-free and drag and drop the script GenerateObjects from the Project window to the object called generateObjects in the

Once this is done, you can select the object called generateObjects in the if you look at the Inspector window, you should see that this object has a component called along with an empty placeholder (i.e., a public variable) called

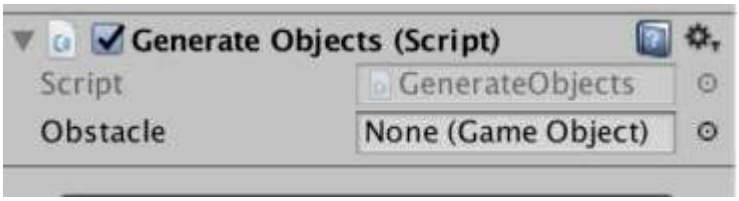


Figure 4-20: Initializing the obstacle (part 1)

Please drag and drop the prefab called obstacle from the Project window to this empty field, as illustrated in the next figure.

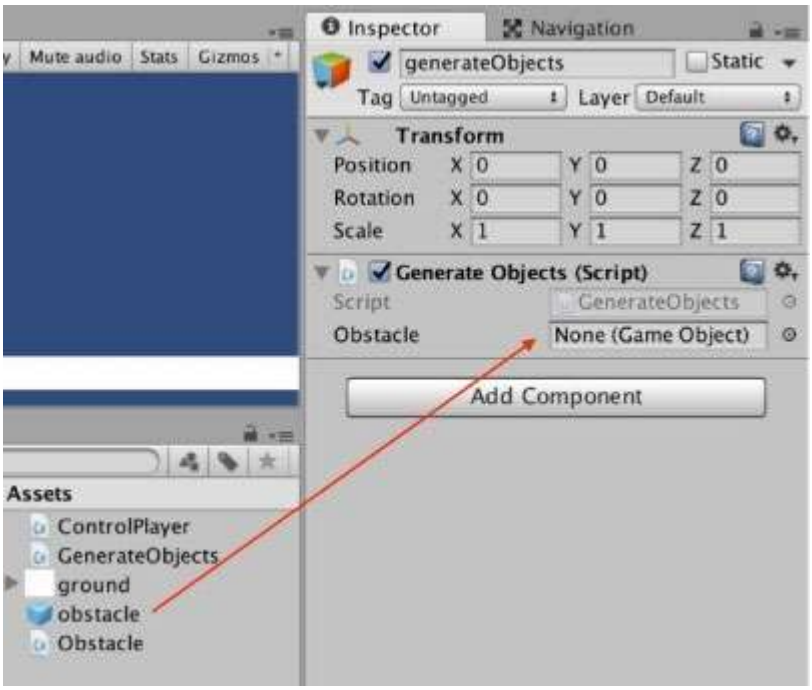


Figure 4-21: Initializing the obstacle (part 2)

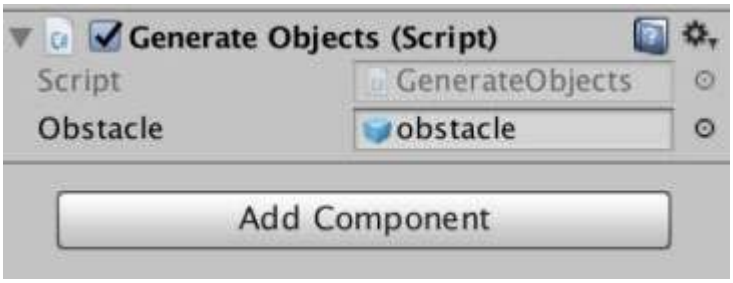


Figure 4-22: Initializing the obstacle (part 3)

You can now test your scene, and you should see that new obstacles are being instantiated every two seconds, as in the next figure.

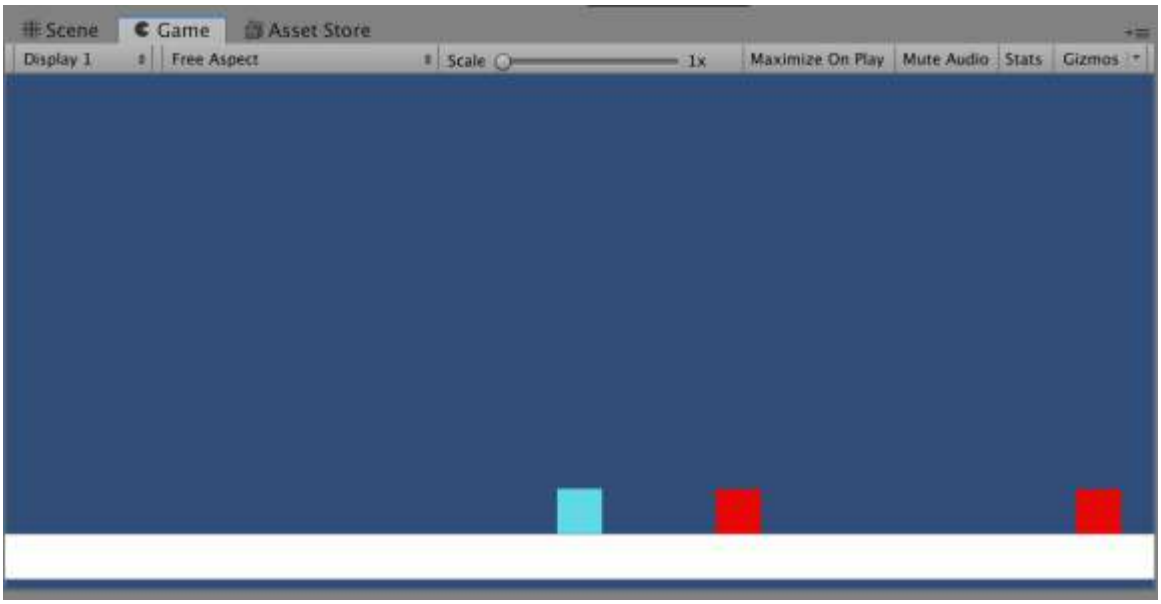


Figure 4-23: Instantiating obstacles

At this stage, we can instantiate the obstacles easily; however, we could improve the game by adding the following features:

Instantiate double or triple obstacles so that the player has to adjust its jump, hence adding more challenge to the game.

Instantiate the objects randomly so that the player never knows whether s/he will have to jump over one, two or three obstacles.

So let’s go ahead:

Please modify the function addObstacle as follows:

```
    void AddObstacle()
    {
    Vector3 positionOfPlayer = GameObject.Find
(“player”).GetComponent().initialPosition;
    float randomNumber = Random.Range (1, 5);
    GameObject t1, t2, t3, t4;
    t1 = (GameObject)(GameObject.Instantiate (obstacle, positionOfPlayer
+ Vector3.right * 20, Quaternion.identity));
    if (randomNumber > 1) t2 = (GameObject)(GameObject.Instantiate
(obstacle, positionOfPlayer + Vector3.right * 21, Quaternion.identity));
    if (randomNumber > 2) t3 = (GameObject)(GameObject.Instantiate
(obstacle, positionOfPlayer + Vector3.right * 22, Quaternion.identity));
    if (randomNumber > 3) t4 = (GameObject)(GameObject.Instantiate
(obstacle, positionOfPlayer + Vector3.right * 21 + Vector3.up,
Quaternion.identity));
    }
```

In the previous code:

We create a random number that will range between 1 and
In all cases, we add a first obstacle.
If the random number is more than one, then we add a second obstacle.
If the random number is more than two, then we add a third obstacle.

If the random number is more than three, then we add a fourth obstacle.

You can test your scene, and you should see that the game instantiates different types of obstacles.

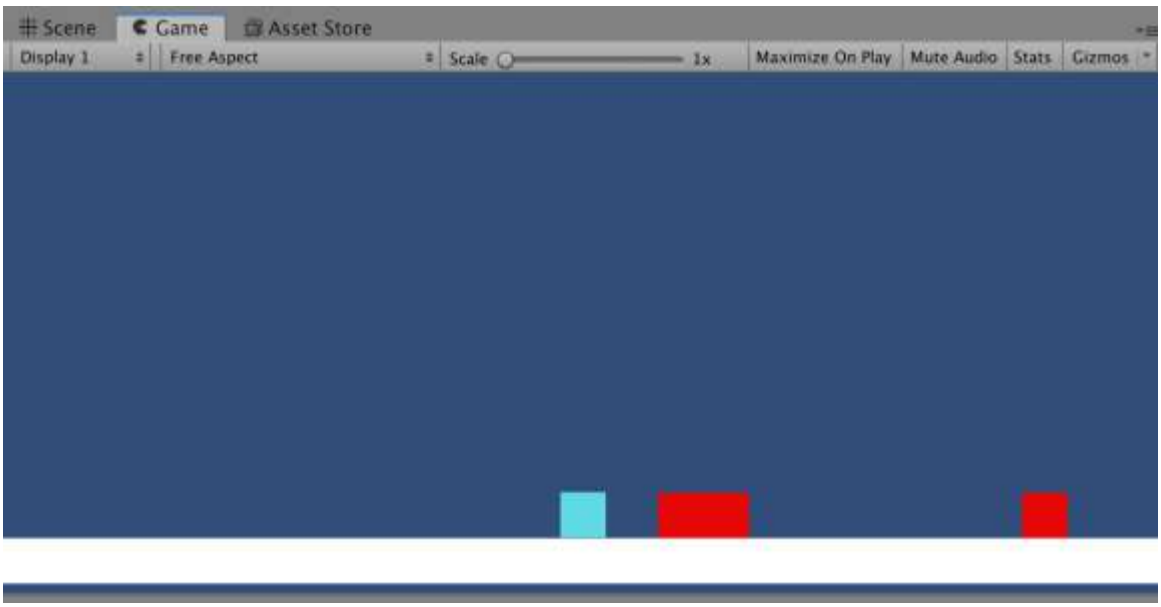


Figure 4-24: Instantiating objects at random

Next, we need to detect collisions between the player and these obstacles; if the player collides with any of the red boxes, we should restart the level. For this purpose, we will modify the script called ControlPlayer as follows:

Please open the script
Add the following code at the beginning of the file.

using UnityEngine.SceneManagement;

Add the following code to the function

if (coll.collider.tag == “obstacle”) SceneManager.LoadScene (SceneManager.GetActiveScene ().name);

In the previous code, we detect whether we collide with an object that has a tag called in this case, the current level (or the active scene) is restarted.

For this to work, we also need to add our current scene to the Build

Please open the Build Settings | Build Settings or CTRL + SHIFT +
Click on the button called Add Open
You can also deselect any of the scenes that were created in the previous chapters.

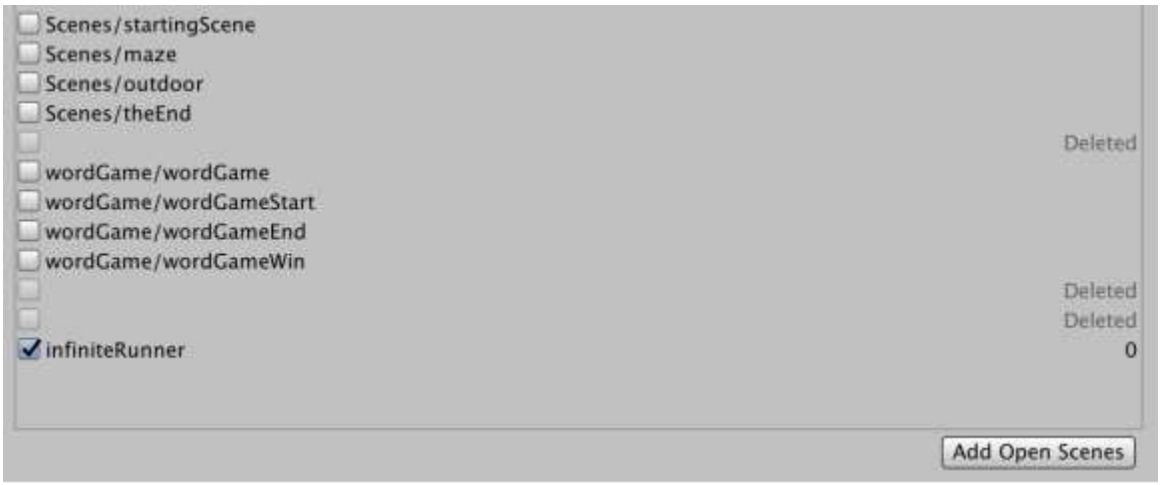


Figure 4-25: Updating the Build Settings

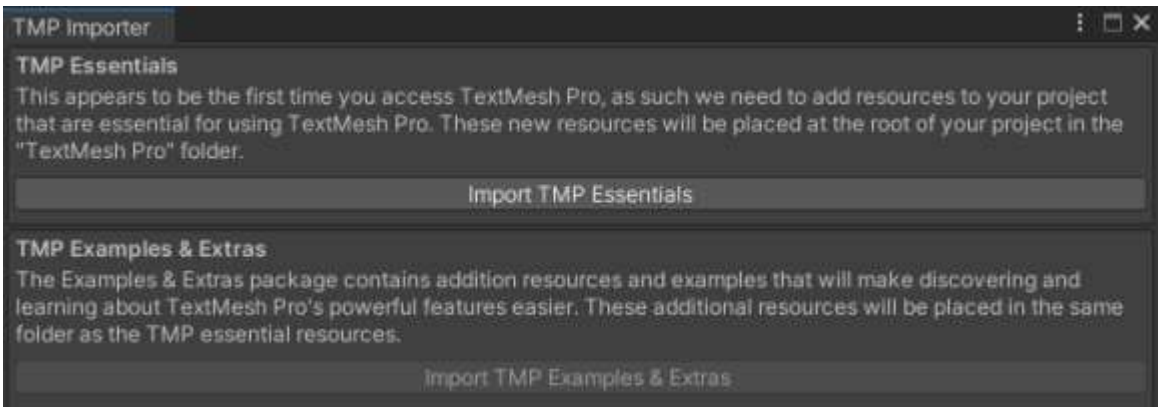
This should add the current scene to the settings, as illustrated in the previous figure.
You can then close the Build Settings window.

Displaying the score

So at this stage, the core mechanics work relatively well, as our character can jump over boxes created at random; another interesting feature for this game could be to add a score based on the time; so the longer the player manages to play the game and avoid obstacles, the higher the score. So we will proceed as follows:

Please create a new UI Text object i.e., | UI |

If the window TMP Importer appears, please click on TMP and close the window after the import is complete.



This will create a new object named Text

Rename this object

Select this object and, using the modify its attributes as follows:

PosX = 0; PosY = 228

Font color =

Width = 400; height =

Alignment TextMesh centered both horizontally and vertically (see next figure).

Figure 4-26: Aligning the text

Font size =

Text: empty

After making these changes, your text field should be located as illustrated in the next figure.

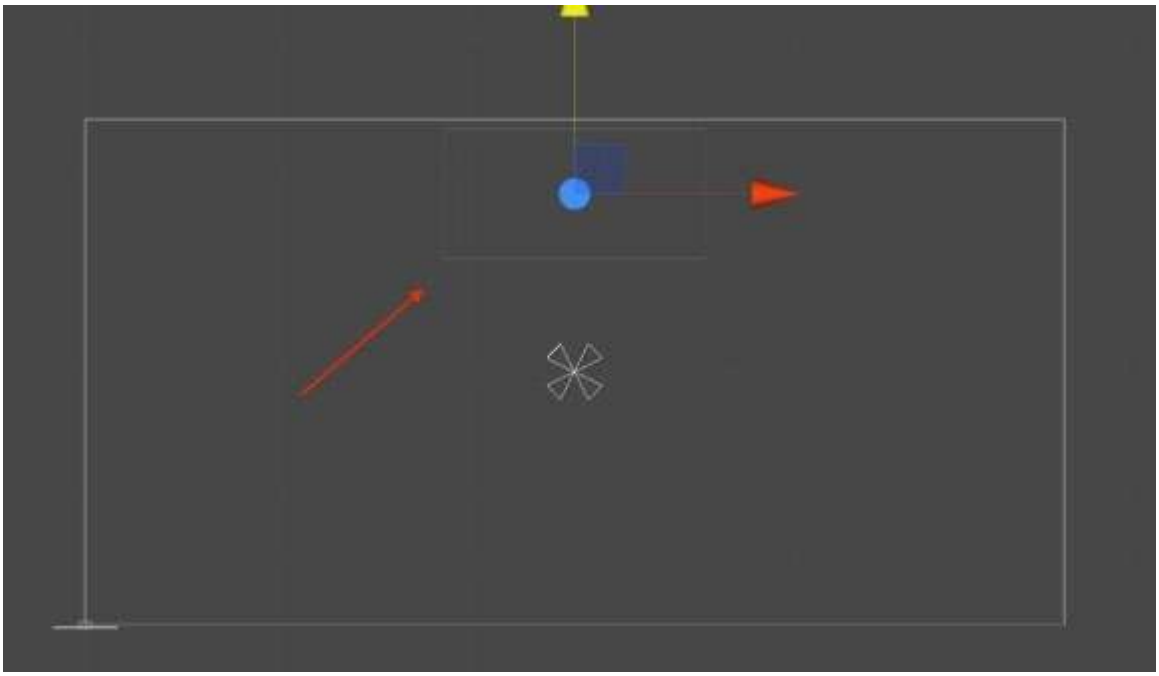


Figure 4-27: Placing the text field for the score

Next, we will create the code to update this UI element.

Please open the script

Add the following code at the beginning of the script.

```
using UnityEngine.UI;
using TMPro;
```

Add the following code at the beginning of the class.

```
float score;
```

Add the following code to the Update function:

```
score += Time.deltaTime;
DisplayScore ();
```

Add the following function just before the end of the class (i.e., before the last closing curly bracket).

```
void DisplayScore()

{
GameObject.Find("scoreUI").GetComponent().text = "" +(int)score;
}
```

You can now test your scene and check that the time is updated accordingly, as illustrated in the next figure.

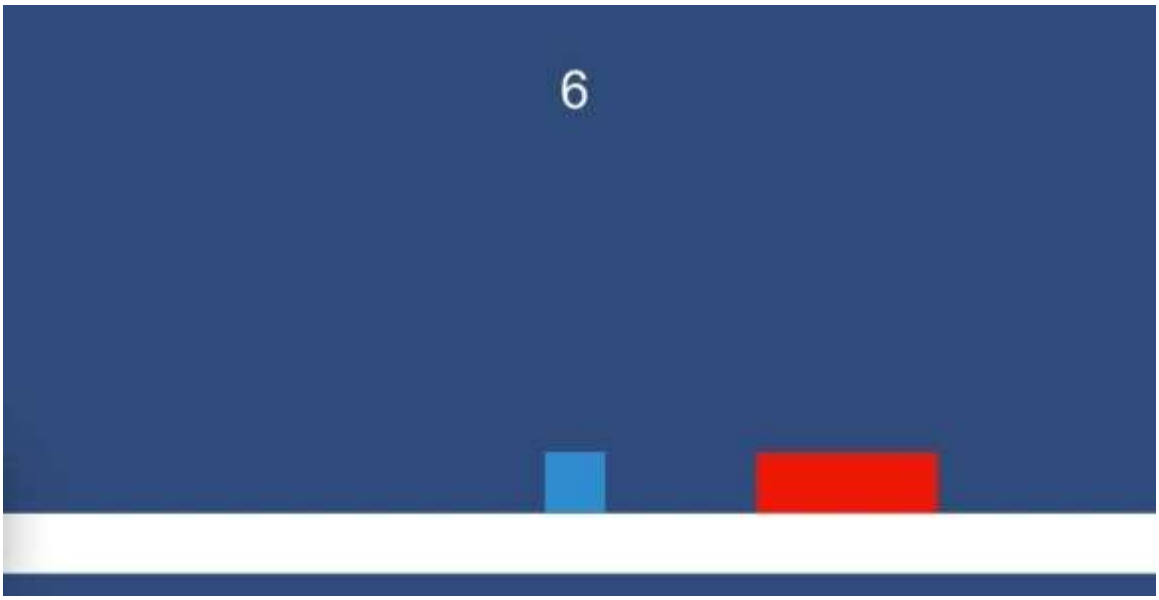


Figure 4-28: Updating the time

Improving the appearance of the game

In the next section chapter, we will improve the look & feel of our game by adding the following features:

Some more distinctive attributes for our character (e.g., eye, mouth, etc.)

Mountains made of basic shapes.

A sun.

Clouds that are instantiated randomly.

Triangular obstacles.

Trees instantiated along the way.

A pause button.

Our complete scene will look as follows:

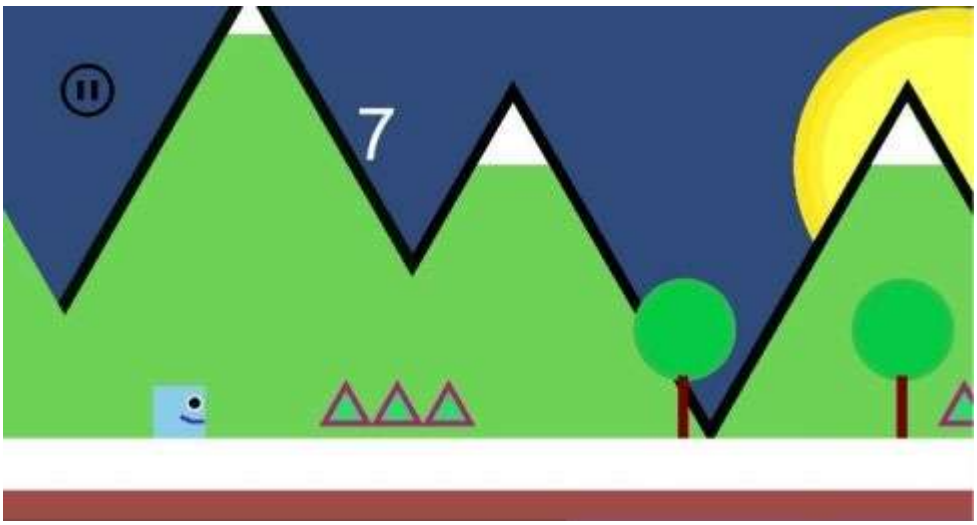


Figure 4-29: An overview of the game (part 1)

The trees will be instantiated frequently.
Objects will have a different depth and we will apply a parallax effect, a visual effect that mimics depth whereby objects closer to the user will move faster onscreen.
Whenever the game is paused, the player will have the choice to resume or to quit the game.

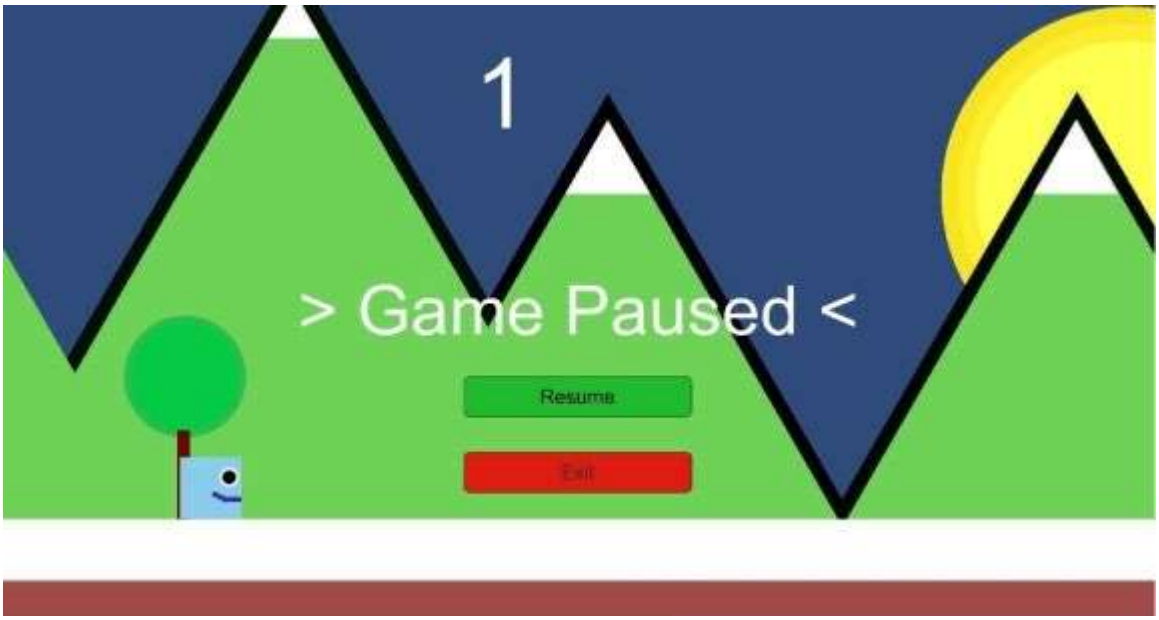


Figure 4-30: An overview of the game (part 2)

Creating the static environment

So let's start to create the static environment that is made of the mountains, the ground and the sun.

In Unity, using the please duplicate the object called and call the duplicate Chang its color to a light brown, and move it down slightly, for example to the position (0,

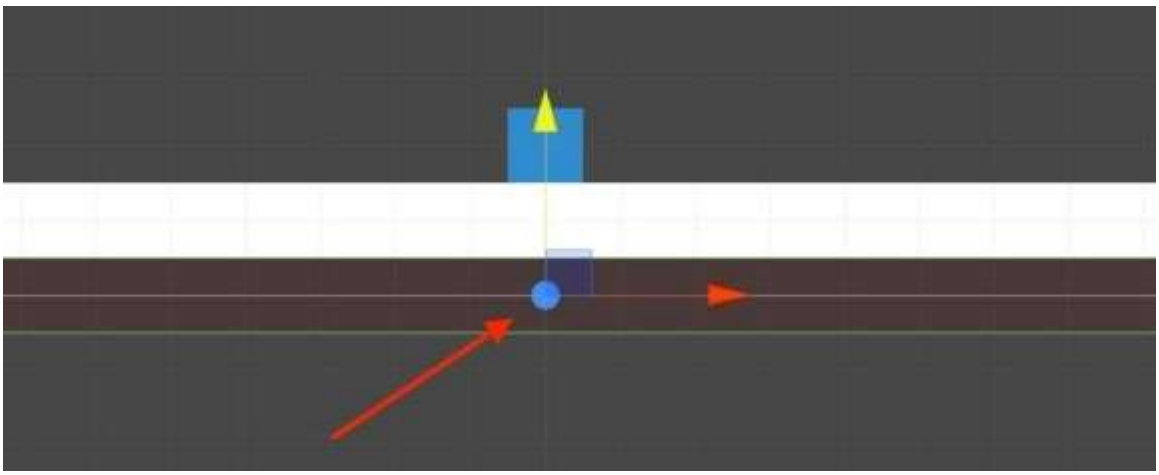


Figure 4-31: Duplicating the ground

We can then start to create mountains:

From the Project window, create a new triangular sprite (i.e., select: Create | 2D | Sprites |
Drag and drop this asset to the Scene view (this will create a new object) and rename the new object
Change its scale to (10, 10, its position to (-10, -1, its color to and the attribute order in layer to as per the next figure.

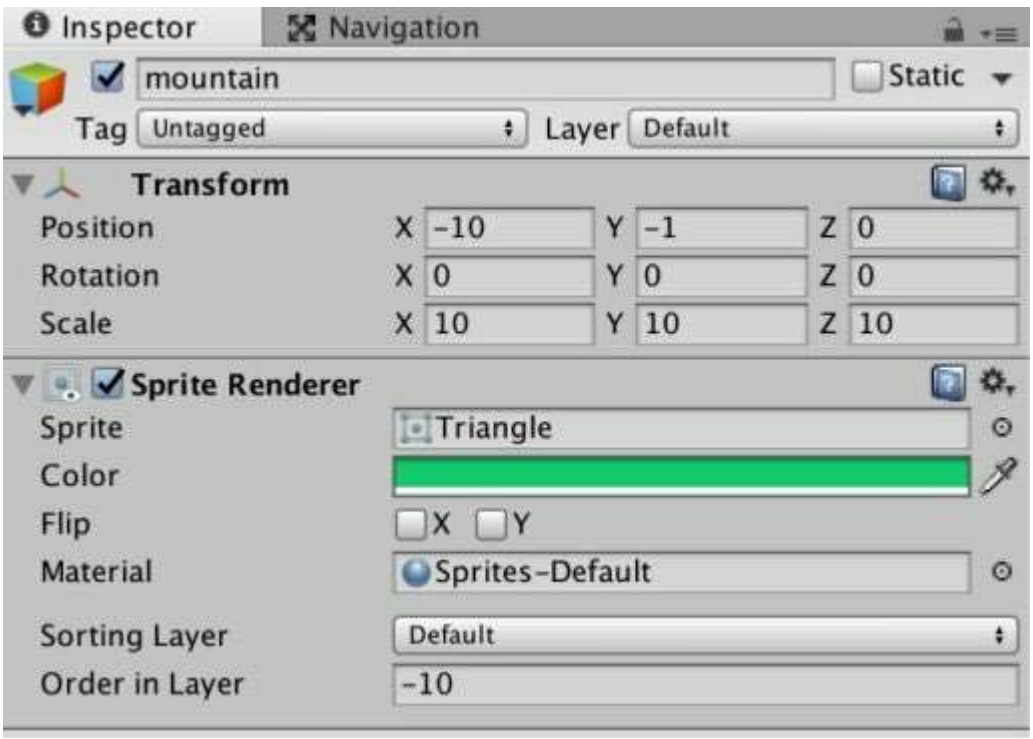


Figure 4-32: Creating a mountain (part1)

The variable called in defines a depth for an object; the objects with the highest value for the attribute in will be displayed atop the other objects.

The mountain should look as in the next figure:

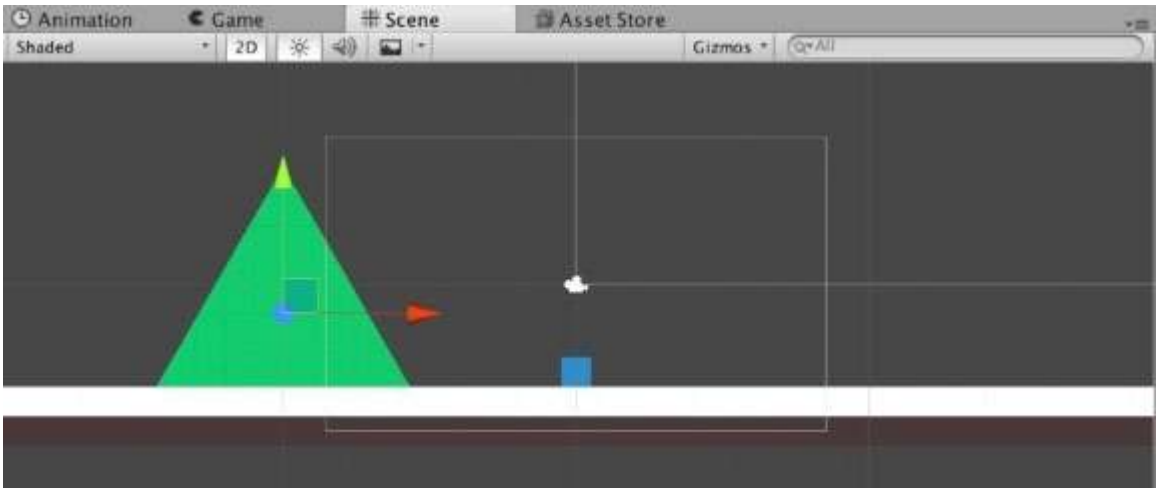


Figure 4-33: Creating a mountain (part2)

Please duplicate this object twice (i.e., the rename the duplicates mountain1 and and place the duplicates at the position (-3, -1, 0) and (5, -1. respectively, so that they look like the following figure.

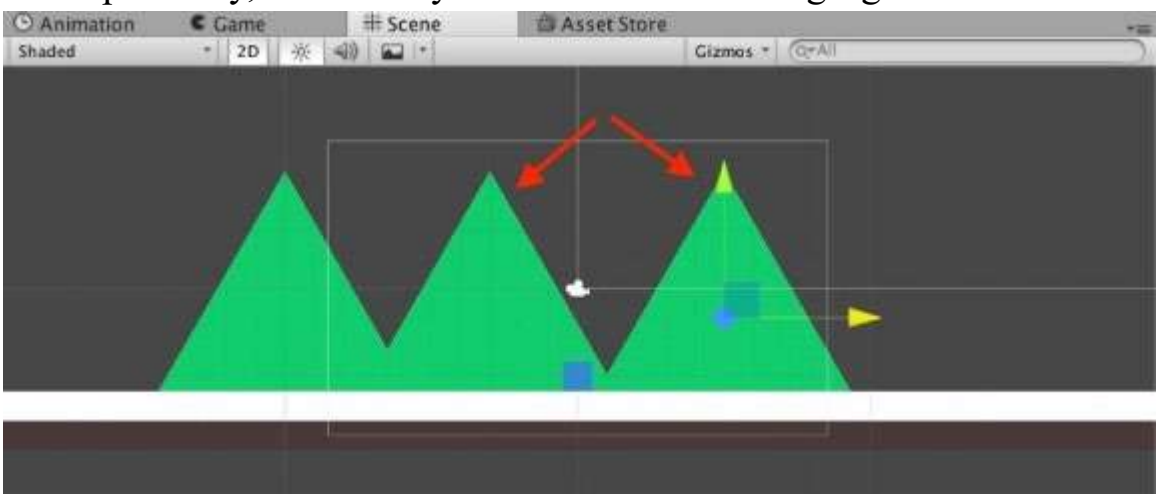


Figure 4-34: Creating additional mountains

After this, we can create the sky:

Please duplicate the object called ground (i.e., CTRL + and rename the duplicate

Change its color to a light blue, its position to (0, 0, 0) its scale to (30, 30, and its attribute Order in Layer to -20 (so that any object is displayed atop the sky).

Using the deactivate its component This is because we do not need to detect any collisions with this object.

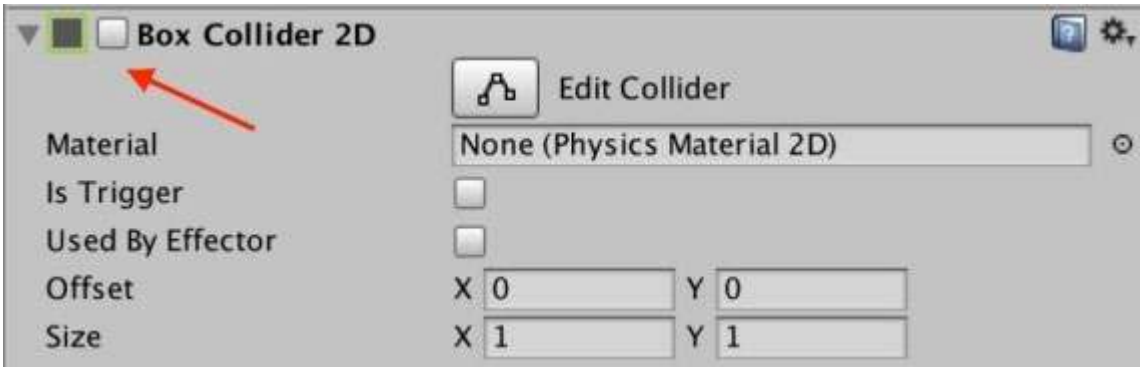


Figure 4-35: Deactivating the collider

The sky object should look as follows, from the Scene view:

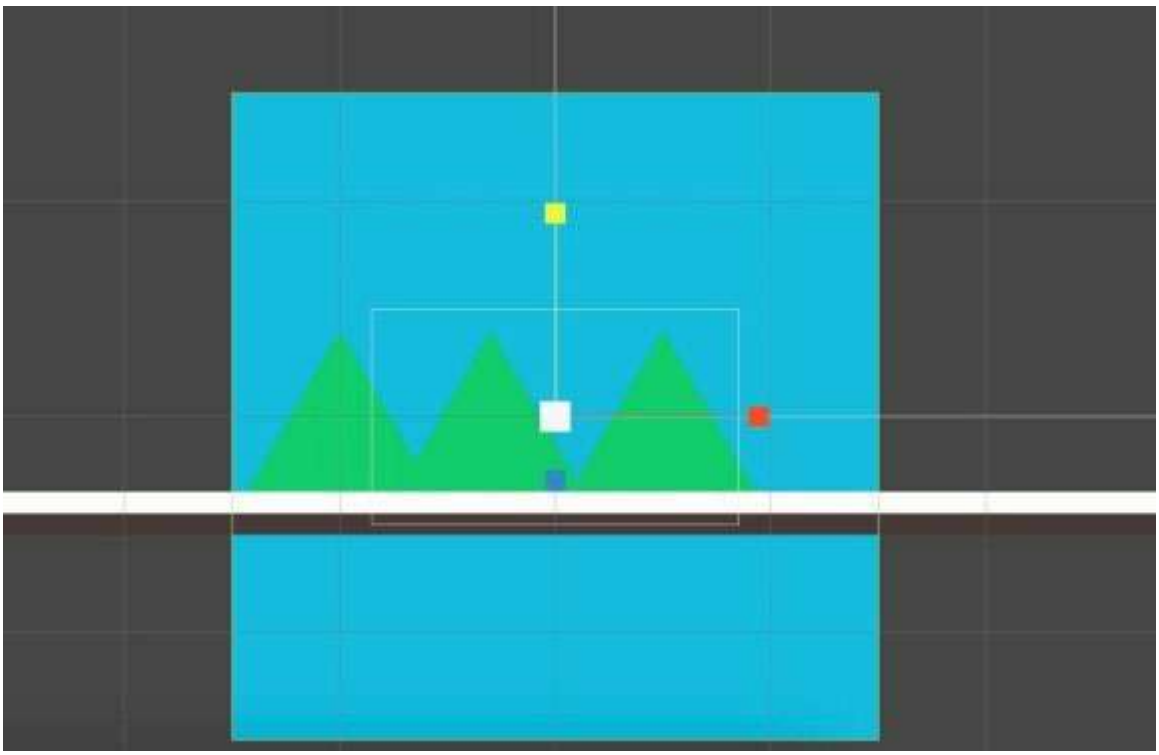


Figure 4-36: The new sky from the Scene view

The sky should look as follows, from the Game view:

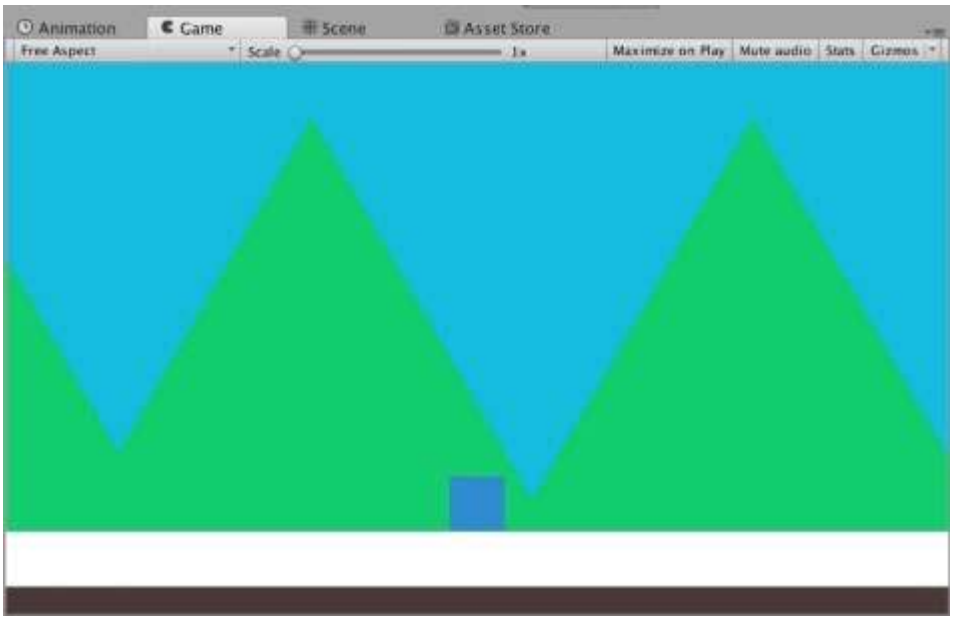


Figure 4-37: The new sky from the Game view

Next, we will add a sun to the scene:

From the Project window, please locate the sprite called If it is not present in the Project window, you can, instead, create a new circular sprite (i.e., select: Create | 2D | Sprites | this will create a new asset called Circle in the Project window.

Drag and drop this asset to the Scene view; this will create a new object called Circle in the Scene view.

Rename the new object

Change its position to (12, 3, 0), its scale to (5, 5, 1) and its color to a light as per the next figure.

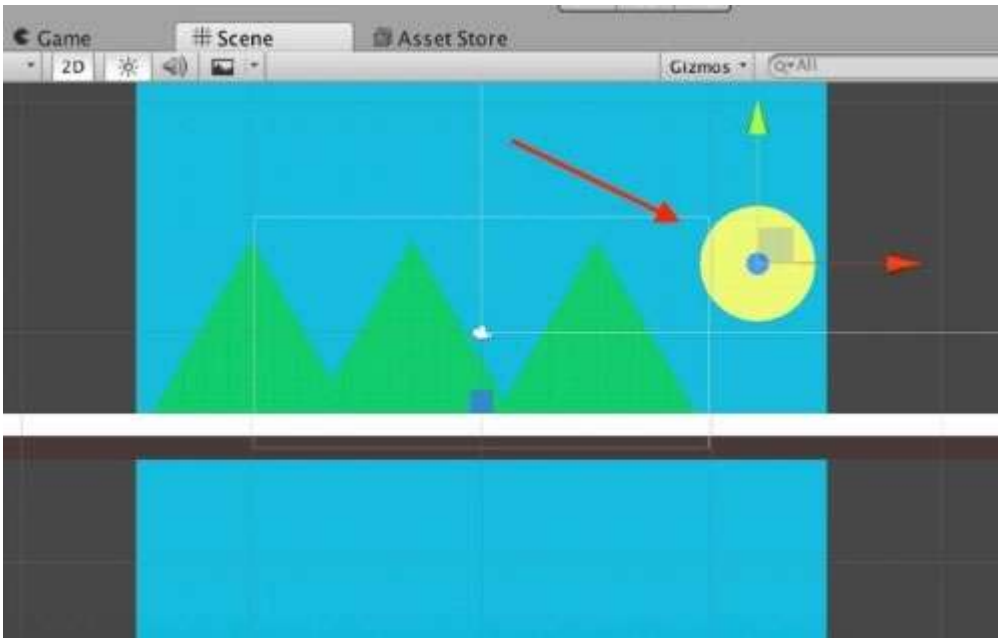


Figure 4-38: Adding the sun

Once this is done, we will add a few clouds generated randomly, to add some life to the scene.

Please create a new polygonal sprite: from the Project window (i.e., select Create | 2D | Sprites |
Rename it

Using the modify the scale of this object to (1, 0.7,
This object will look like the one illustrated in the next figure.

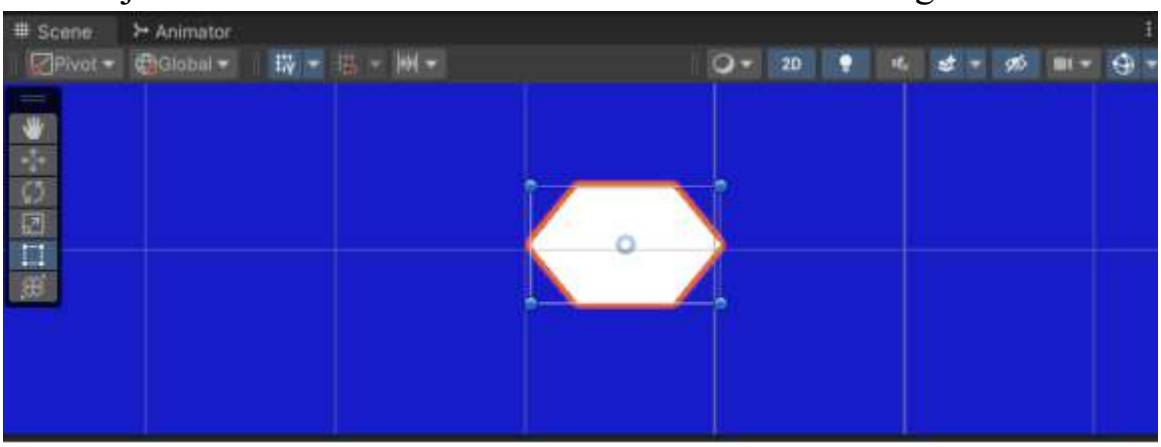


Figure 4-39: Rescaling the cloud

You can then create a new cloud prefab by dragging and dropping this object (i.e., the to the Project view.

Figure 4-40: The new cloud prefab

Once this is done, you can deactivate (or delete) the object called cloud in the Hierarchy window.

We will now modify our scripts so that the game generates clouds at regular intervals:

Please create a new script called Cloud (i.e., from the Project window, select Create | C#

Add the following code to it (new cold in bold):

```
void Update ()
{
transform.Translate (Vector2.left *Time.deltaTime);
if (transform.position.x < -10) Destroy
}
```

Please save your code, check that it is error-free.

You can now add the script Cloud to the prefab

Select the prefab called cloud from the project window
In the Inspector click on the button labelled Open
Drag and drop the script Cloud on the Inspector (so that it becomes a component of the prefab
Exit the prefab editing mode by clicking on the arrow located to the left of the prefab cloud in the

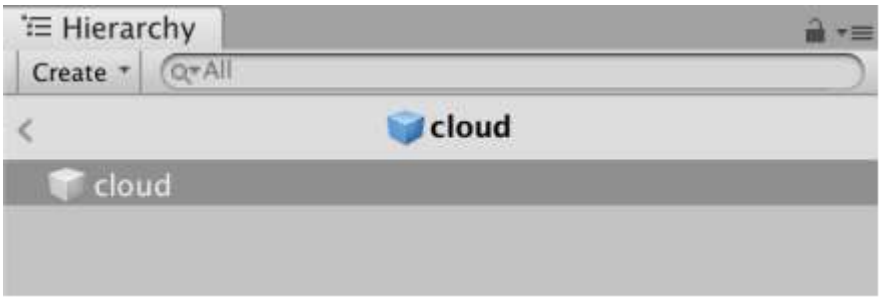


Figure 4-41: Exiting the prefab editing mode

We can then modify the script GenerateObjects so that we can generate these clouds:

Please open the script
Add the following code at the beginning of the class:

```
    float cloudTimer;
    public GameObject cloud;
```

Add the following function just before the end of the class (i.e., before the last closing curly bracket).

```
    void CreateClouds()
    {
    cloudTimer += Time.deltaTime;
    GameObject cloud1;
    Vector3 topRightCorner = GameObject.Find
(“topRightCorner”).transform.position;
    int altitude = Random.Range(0,2);
    if (cloudTimer >= 10)
    {
    cloud1 = (GameObject)(GameObject.Instantiate (cloud,
topRightCorner + -Vector3.up * altitude , Quaternion.identity));
    cloudTimer = 0;
    }
    }
```

In the previous code:

The timer for the cloud is incremented every 10 seconds.
A random variable called altitude is created.
Every 10 seconds, a new cloud is created based on the position of the object called topRightCorner (that we yet have to create) and its relative height is based on the variable called

Since the function has been created, we can now call it from the Update function, so that the clouds are instantiated at regular intervals.

Please add the following code to the Update function:

CreateClouds ();

We can now create the object topRightCorner from where the cloud will be spawned.

Please create a new empty object and rename it topRightCorner (i.e., GameObject | Create

Place it in the top-right corner, as illustrated in the next figure.

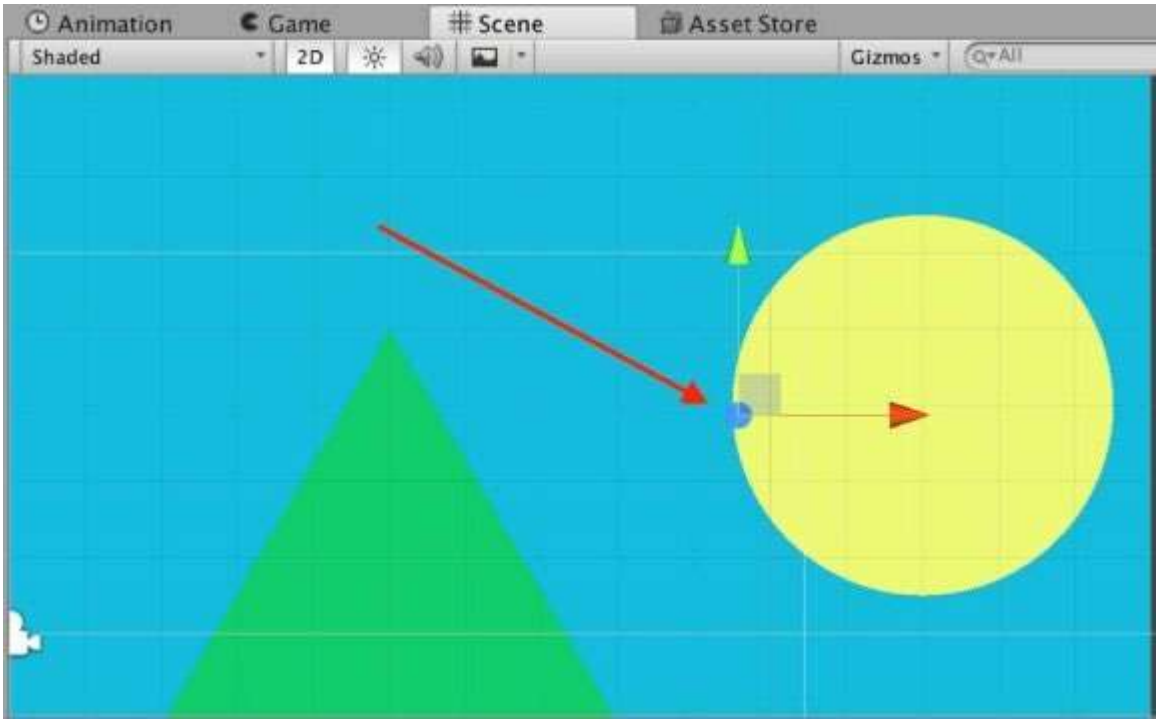


Figure 4-42: Defining the top-right corner of the screen

Last but not least:

Select the object generateObjects in the

In the locate the component called and the variable called

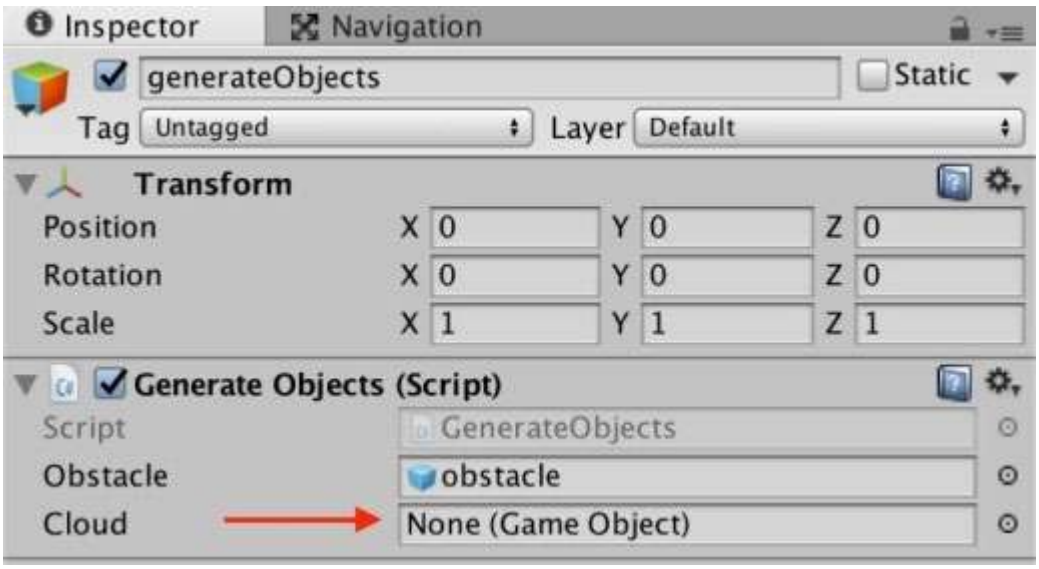


Figure 4-43: Locating the cloud variable in the Inspector

Drag and drop the cloud prefab to the variable cloud for the component

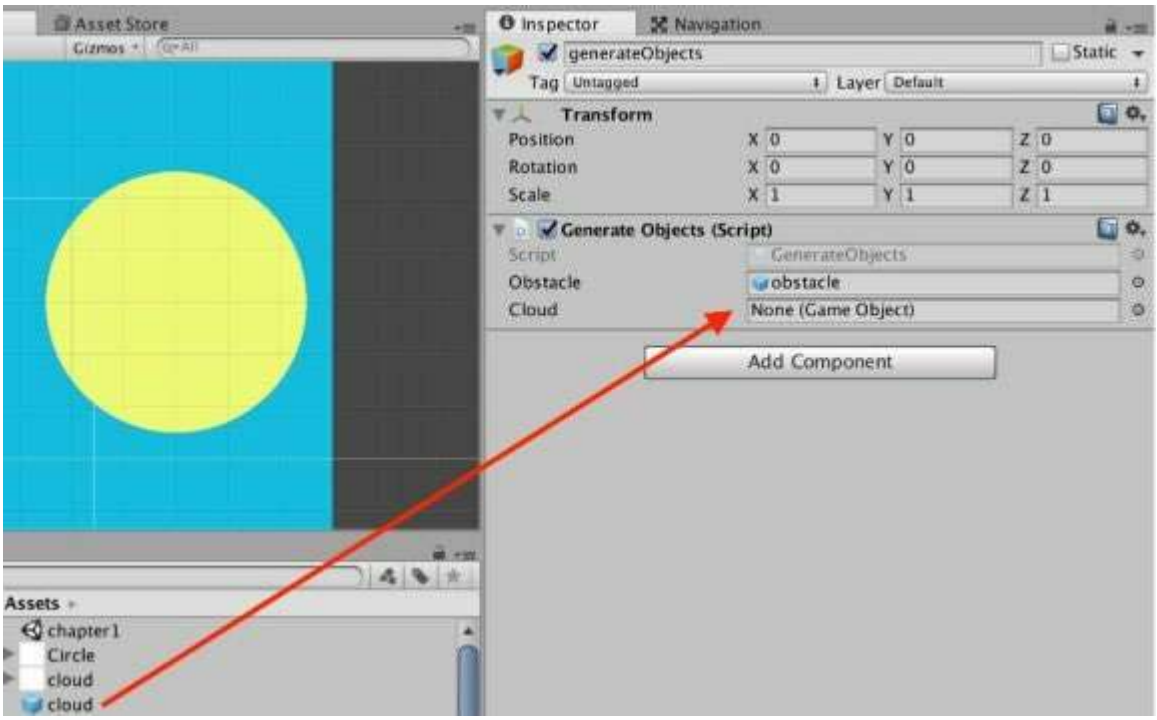


Figure 4-44: Assigning a value to the cloud variable

As you test the scene you should see clouds generated randomly and moving to the left.

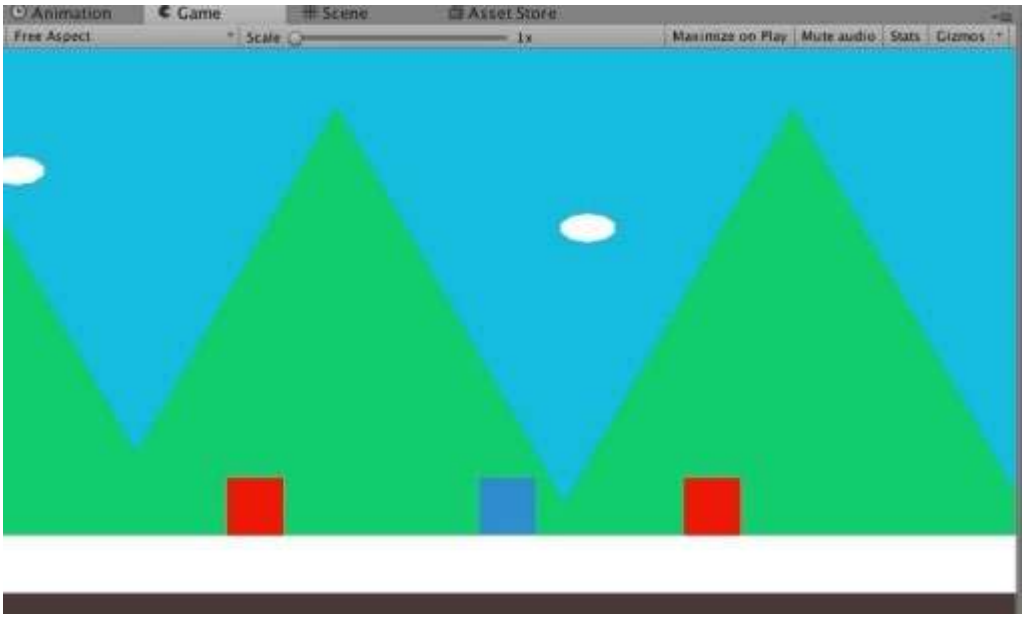


Figure 4-45: Clouds generated randomly.

Once you have checked that the clouds are generated properly, we will start to modify the appearance of the obstacles:

In the Project window, please select the prefab called
Open the prefab using the corresponding button.
Using the scroll-down to the component called Sprite and click on the small circle to the right of the Sprite label, as described in the next figure.

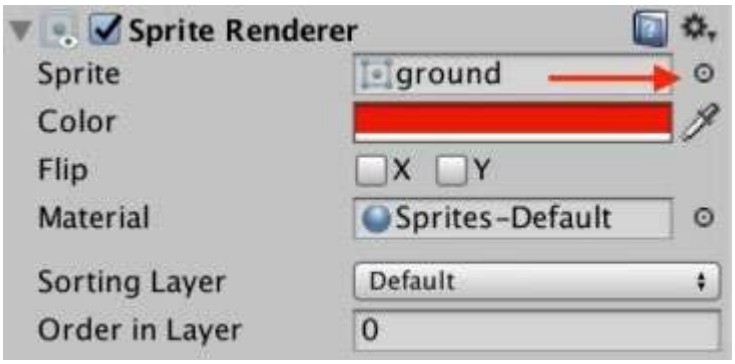


Figure 4-46: Changing the sprite for an object

From the new window, select (i.e., double-click) on the option

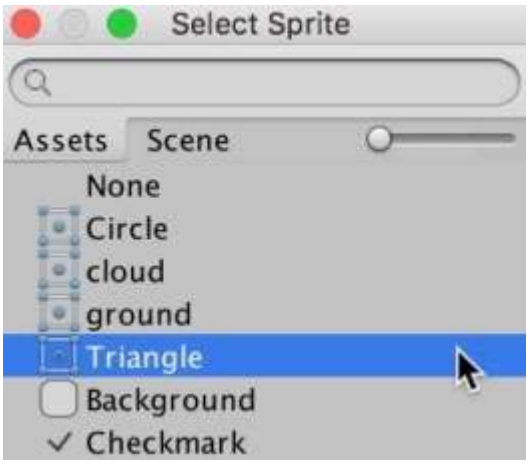


Figure 4-47: Selecting the sprite called Triangle (part1)

This should set the sprite Triangle for this object, as illustrated in the next figure.

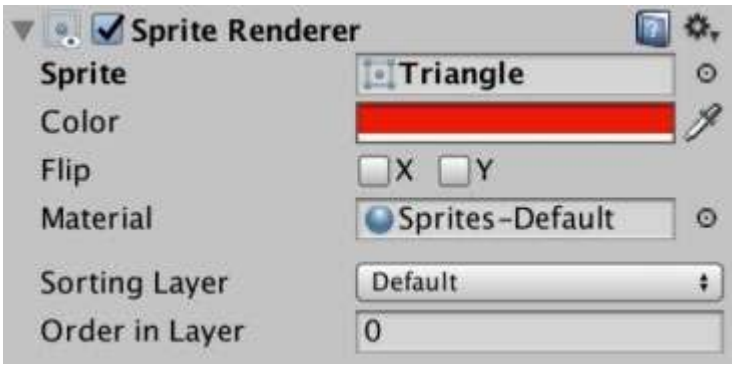


Figure 4-48: Selecting the sprite called Triangle (part 2)

You can now close the editing mode for the prefab by clicking on the arrow to the left of the prefab obstacle located in the

As you play the scene, you will see that the shape of the obstacles is now triangular.

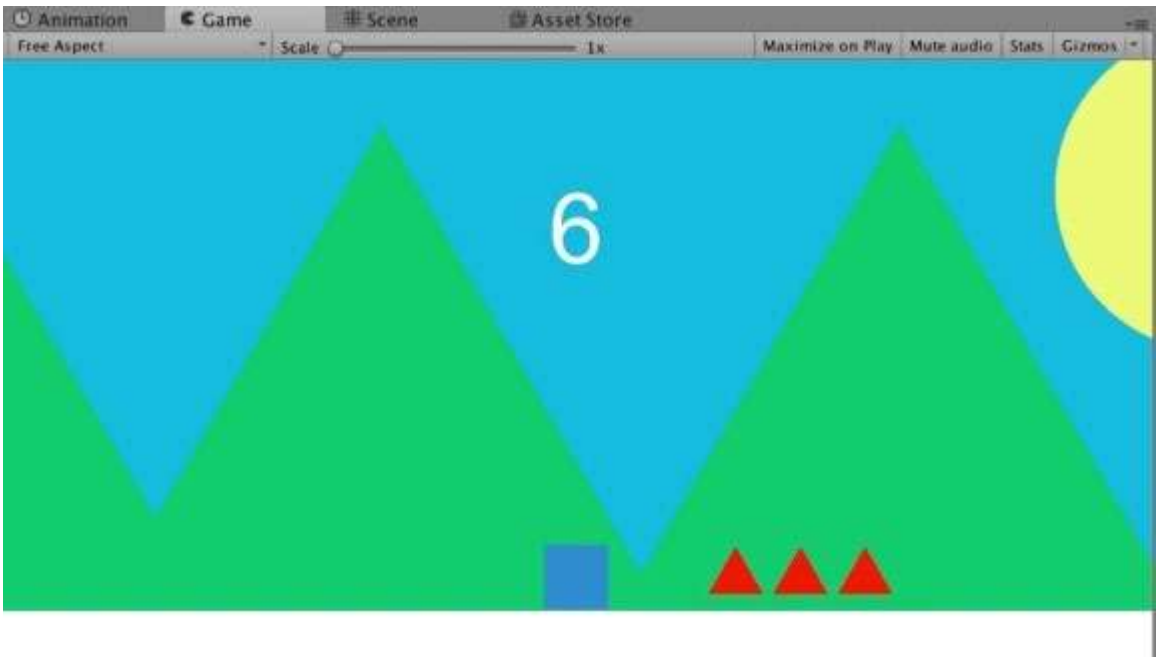


Figure 4-49: Using triangular obstacles

You may notice that these obstacles are now slightly above the ground; so we could modify the function addObstacle in the script GenerateObjects as follows (new code in bold):

Vector3 positionOfPlayer = GameObject.Find (“player”).GetComponent().initialPosition + Vector3.down * .3f;

In the previous code, we use the built-in function Vector3.down to move down the obstacles.

If this does not move down the obstacle, you can also edit the collider of the obstacle (now a triangle) as follows:

Drag and drop the prefab called obstacle to the Scene view.
Select it, and using the Inspector view, scroll down to the component called Box
Click on the button labelled Edit

Figure 4-50: Editing the collider

In the Scene view, drag and drop the bottom edge of the collider upwards.

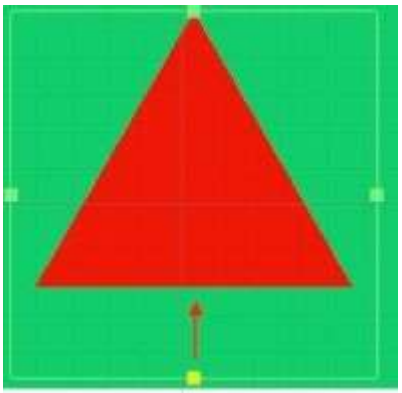


Figure 4-51: The sprite collider before the change

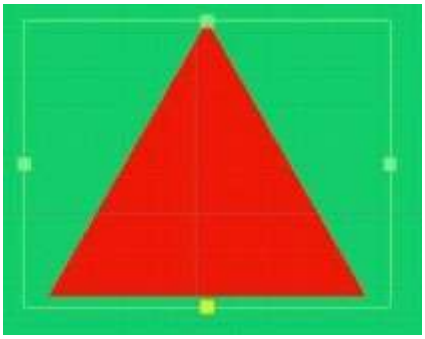


Figure 4-52: The sprite collider after the change.

Click again on the button labelled Edit Collider to end the modification of this collider.

Close the prefab by pressing the left arrow in the Hierarchy window.

Next, we will add some distinctive features to our character, including eyes and a mouth:

Please drag and drop the asset called Circle from the Project window to the Scene view. This asset was used to create the if unsure, you can create a new Circle sprite by selecting Create | 2D | Sprites | Circle from the Project window, and drag and drop it to the Scene view.
This will create an object called Circle in the
Rename this object
In the drag and drop this object on top of the object so that it becomes a child of the object
Change its position to 0.2, and its scale to 0.4, and the attribute Order in Layer to 2 so that it appears above the square for the player.

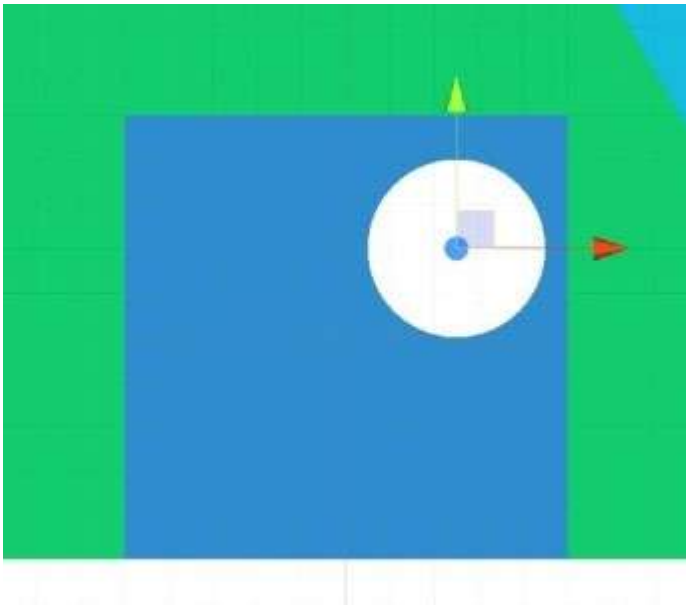


Figure 4-53: Creating the eyes

Duplicate the object called eye and rename the duplicate
Change its scale to (.3, .3, .3) and its color to
Change its attribute Order in Layer to

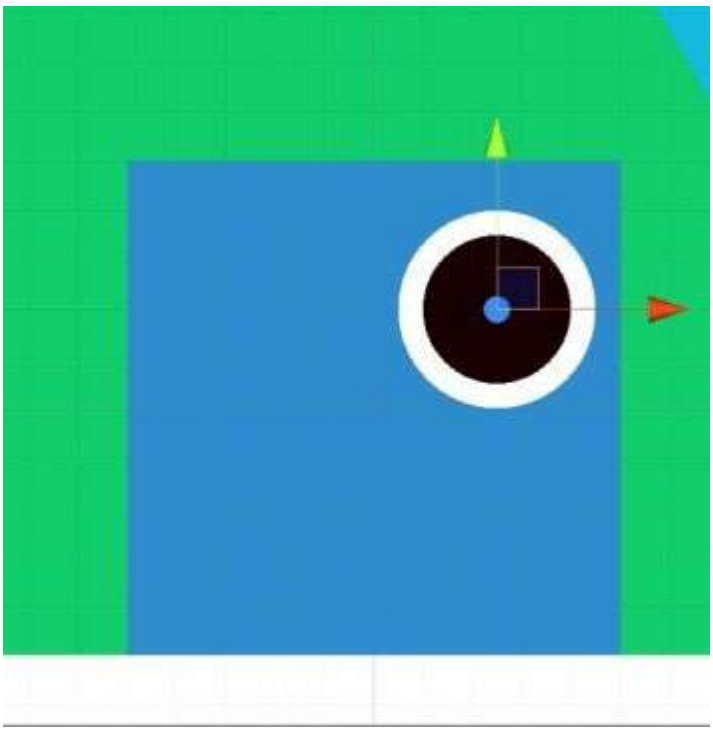


Figure 4-54: Creating the pupils

Next, we will create the mouth:

Duplicate the object pupils and rename the duplicate
Using the scroll-down to the component called Sprite Renderer for this object, and click on the small circle to the right of the label called as described in the next figure.

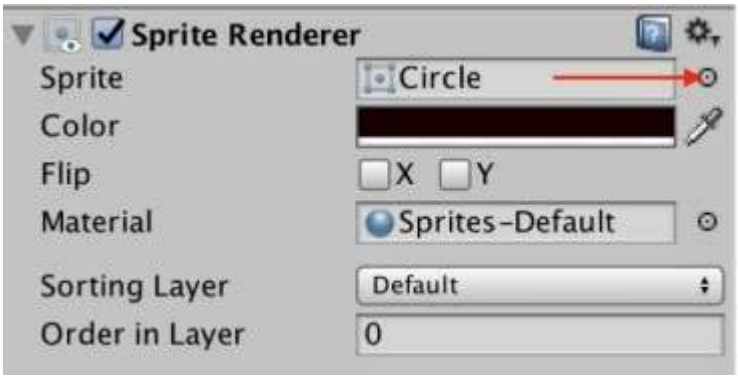


Figure 4-55: Changing the sprite

From the new window, select (i.e., double-click on) the option

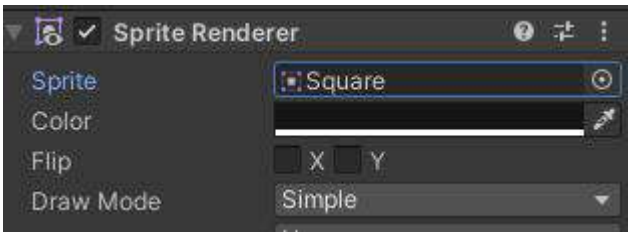


Figure 4-56: Selecting the ground sprite (part 1)

The Sprite Renderer should then look as follows:

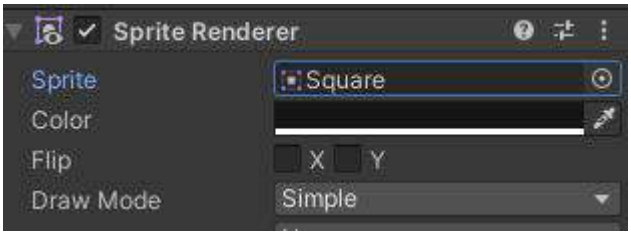


Figure 4-57: Selecting the ground sprite (part 2)

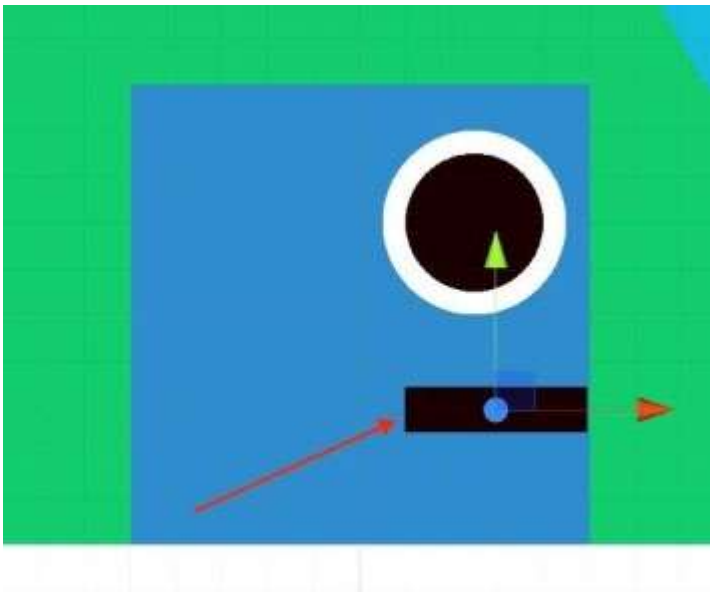


Figure 4-58: Finalizing the mouth

You can adjust the position of the mouth to (.29, -.23, 0) and its scale to (0.41, 0.13,
So that the mouth is visible, set its attribute Order in Layer to 3 (this is so that it is displayed atop the other objects).

If you wish, you can add more distinctive features to the character using the same principle.

You can now test your game and check the look of the player character.

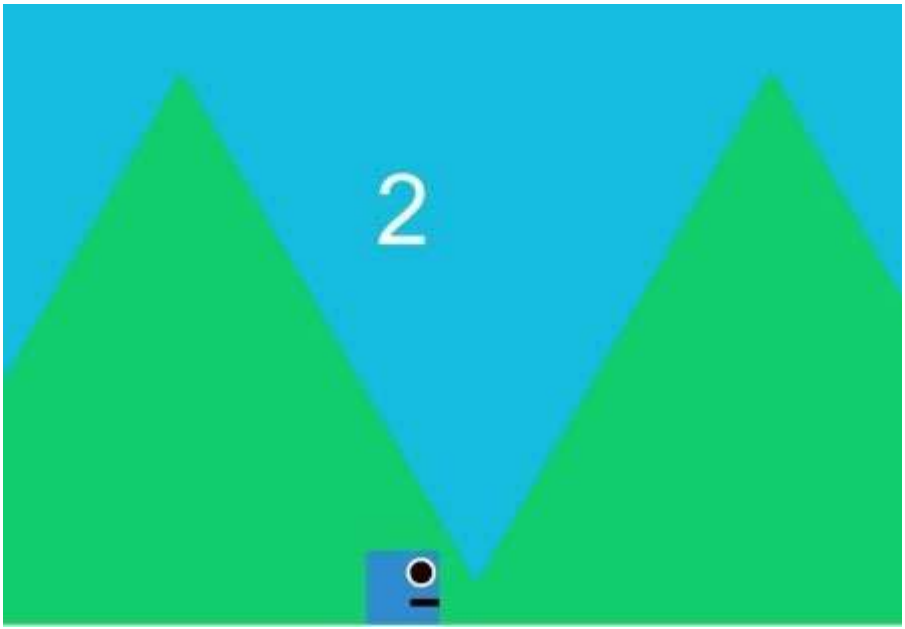


Figure 4-59: The character with added features

Pausing the game

In this section, we will add a mechanism by which the player can pause the game.

A pause button will be displayed in the top left corner of the screen. When this button is pressed, the message will be displayed, along with two buttons to either exit or resume the game.
If the player resumes the game, both the resume and exit buttons, along with the text will be hidden.

So let’s get started!

In Unity, create a new button (i.e., GameObject | UI |
Rename it

Change the label of this button to Select the Text object that is a child of this object, and then select this object and, using the Inspector window, modify its text component.

Figure 4-60: Identifying the text of the button's label in the Hierarchy

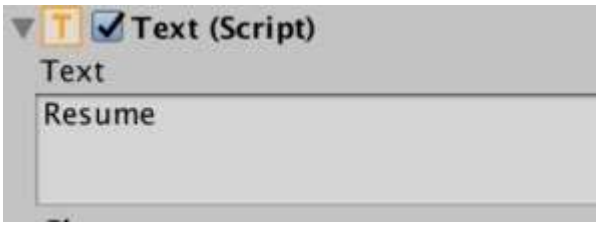


Figure 4-61: Modifying the label of the button

Once you have changed its label, please duplicate this button (i.e., CTRL
+
Call the duplicate
Change the label of the duplicate to
Move these buttons, so that you obtain a layout like the one described in the next figure.

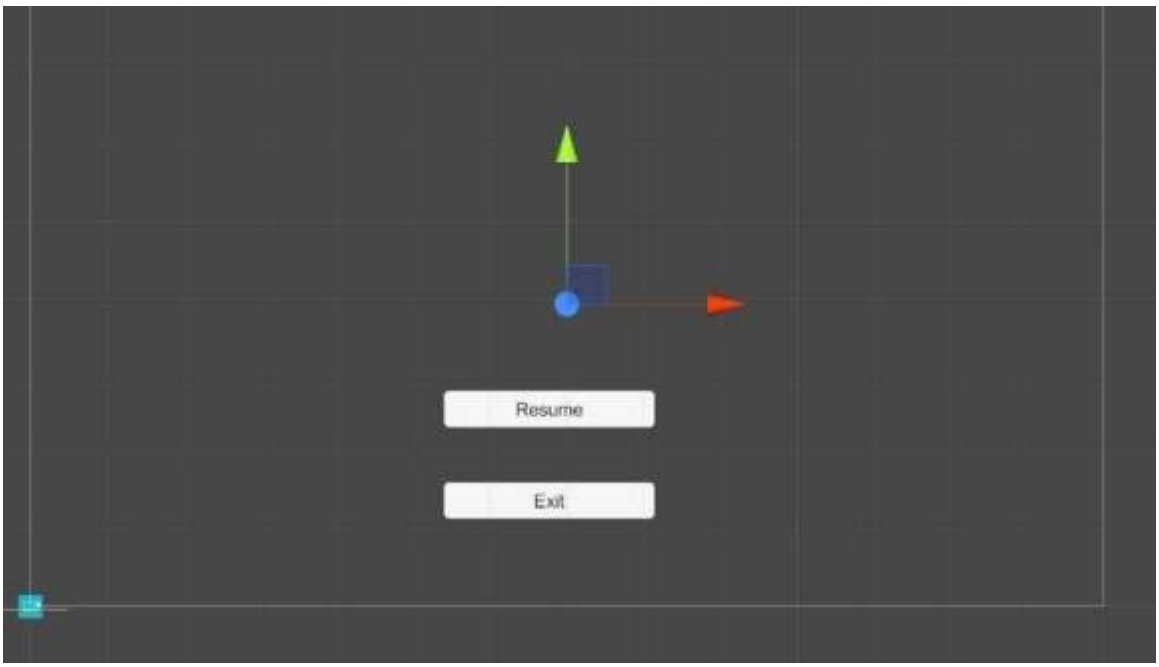


Figure 4-62: A layout for the buttons

Next, we will create a simple text label that will indicate that the game is paused.

Please create a new UI Text object | UI |
Rename it
Change its text to >> Game Paused its color to its width to its height to its font size to and its alignment so that it is centered both vertically and
You may also move this text field above the previous two buttons to obtain a layout similar to the one illustrated in the next figure.

Figure 4-63: Completing the layout for the buttons and the text

Next, we will create the pause button; it will be based on an image that is available in the resource pack.

Please duplicate the object and rename the duplicate
Empty its label.
Move the button to the top left corner, as per the next figure.

Figure 4-64: Adding the pause button

We will now import a sprite for the pause button and apply it to this button:

Locate the file called pause.png in the resource pack.
Import this file into your project (i.e., drag and drop).

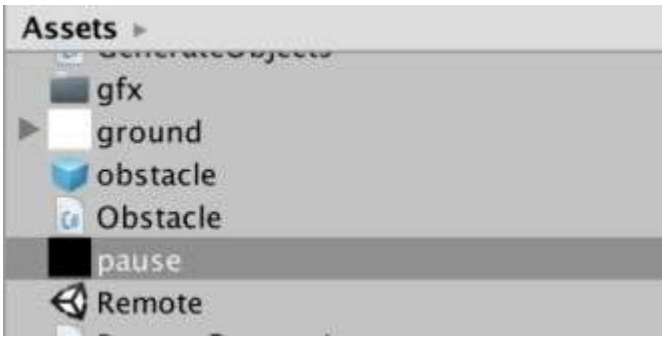


Figure 4-65: Importing the pause button

Select this asset in the Project window.

Using the change its texture type to Sprite (2D and UI) and its Sprite Mode to so that it can be displayed on the user interface.

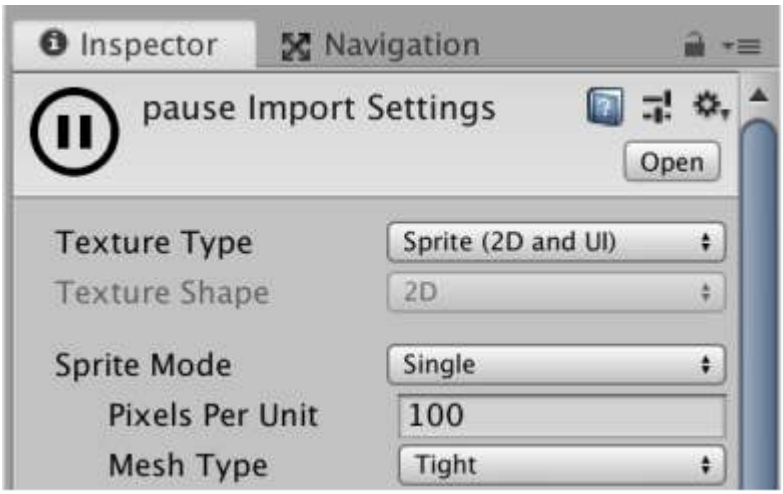


Figure 4-66: Changing the attribute of the button

Once this is done, you can press the button labelled located in the bottom-right corner of the Inspector window.

Figure 4-67: Applying changes to the button

Select the button pauseBt in the
Using the scroll down to its component called

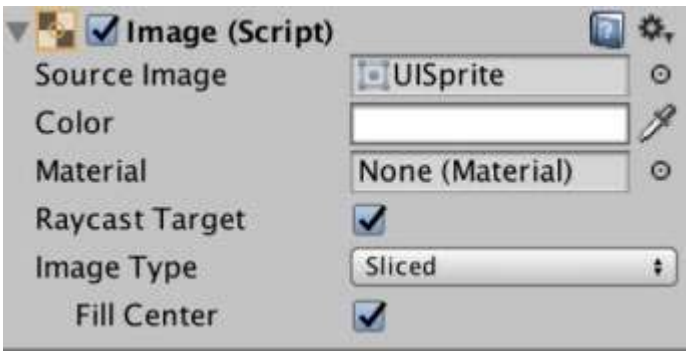


Figure 4-68: Modifying the appearance of a button

Modify the Source Image by clicking on the circle to the right of the label Source

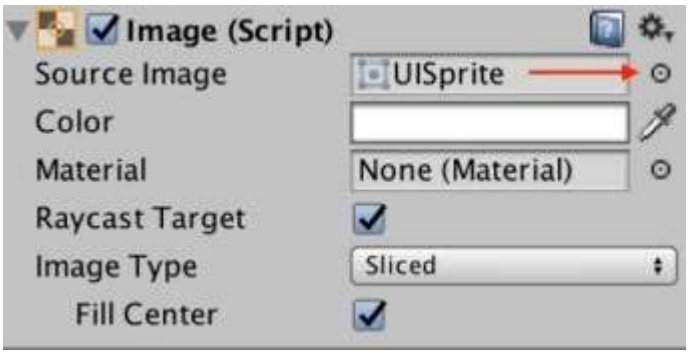


Figure 4-69: Changing the image of a button

In the new window, select (double click) the asset called

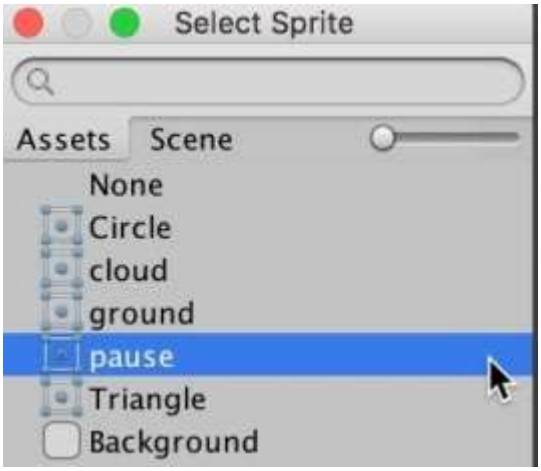


Figure 4-70: Selecting a new image for the button

You can then modify the width and height of the button to (100, 100) and its scale to (0.5, 0.5, 1) and adjust the position of the button so that it fits within the screen, and to obtain a layout similar to the one illustrated in the next figure.

Figure 4-71: The final layout

Now that we have defined the appearance and location of our buttons, we will write the necessary code to handle clicks (or taps) on these buttons.

Please create a new empty object called manageBt | Create
Create a new C# script called ManageBT (i.e., from the Project window, select: Create | C#
Drag and drop this script to the object
Open the script
Add the following code at the beginning of the script.

using UnityEngine.SceneManagement;

The previous code is necessary so that we can use the built-in function

Add the following code just before the end of the class:

public void StartGame()

```
{

SceneManager.LoadScene(“infiniteRunner”);
}
public void Resume()
{
GameObject.Find(“player”).GetComponent().Resume();
}
public void Pause()
{
GameObject.Find(“player”).GetComponent().Pause();
}
public void Exit()
{
Application.Quit ();
}
```
In the previous code:

We define functions that will be called in case the game is started, resumed, stopped, paused or exited.
The function start loads the scene called
The function Resume calls the function Resume that is part of the class The latter is public; this is the reason why it can be accessed from outside its class.
The function Pause calls the function Pause that is part of the class The latter is public; this is the reason why it can be accessed from outside its class.
The function exit just exits/quits the game.

Next, we will need to modify the script called ControlPlayer to include the function Pause and

Please open the script called
Add this code at the beginning of the class:

```
GameObject btPause, btExit, btResume, gamePausedTxt;
public bool paused;
```

In the previous code, we define placeholders that will be linked to the corresponding buttons (i.e., start, resume, exit, etc.).

Modify the Start function as follows (new code in bold):

```
void Start ()
{
paused = false;
initialPosition = transform.position;
gamePausedTxt = GameObject.Find (“gamePaused”);
btResume = GameObject.Find (“resumeBt”);
btExit = GameObject.Find (“exitBt”);
btPause = GameObject.Find (“pauseBt”);
DisplayPauseButtons (false);
}
```

In the previous code:

We state that the game is not paused yet.
We initialize the objects created earlier and link them up to the corresponding buttons.

Now that we have initialized these variables, we will define the actions to be performed when the functions and DisplayPauseButtons are called.

Please add the following code just before the end of the class (i.e.,

```
    public void Pause()
    {
    paused = true;
    DisplayPauseButtons (true);
    Time.timeScale = 0;
    }
    void DisplayPauseButtons(bool state)
    {
```

```
gamePausedTxt .SetActive (state);
    btResume.SetActive (state);
    btExit.SetActive (state);
    btPause.SetActive (!state);
    }
    public void Resume()
    {
    Time.timeScale = 1;
    DisplayPauseButtons (false);
    }
```

We can now allocate actions to buttons:

Please check that the script ManageBT is linked to the object
Please select the button

Using the scroll down to the component called

Click on the + button.

Figure 4-72: Handling events (part 1)

This will show new attributes; you can click on the circle to the right of the field None as illustrated in the next figure.

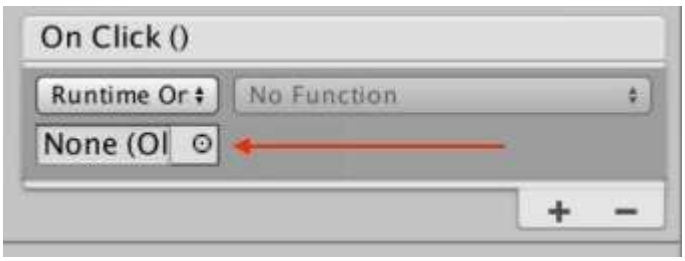


Figure 4-73: Handling events (part 2)

In the Scene window, please search for and select the object

Figure 4-74: Handling events (part 3)

In the Inspector, click on No and select ManageBT | as illustrated in the next figures.

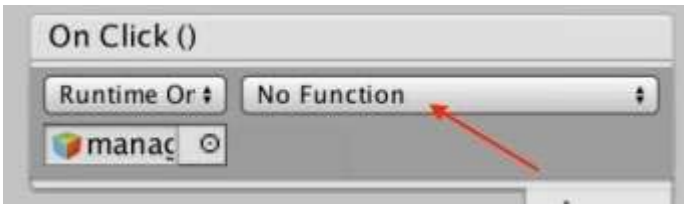


Figure 4-75: Handling events (part 4)

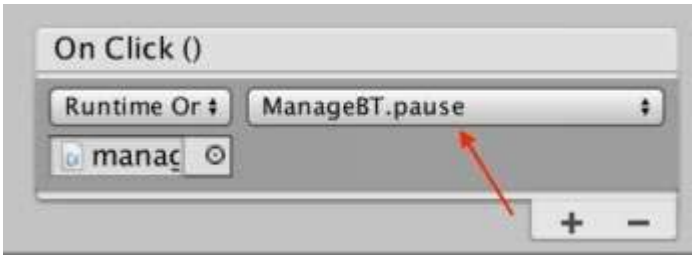


Figure 4-76: Handling events (part 5)

Once this is done, repeat the previous steps so that the button exitBt is linked to the function and so that the button resumeBt is linked to the function

You can then play the scene and check that you can pause the game. If you run into any issue whereby the pause or resume buttons are not responsive, please make sure that they don't overlap with the UI Text.

Level roundup

In this chapter, we have learned to create a simple infinite runner where the player can jump to avoid obstacles. Along the way, we have learned a few interesting skills including: generating objects randomly, detecting the user’s input, or displaying the time onscreen. So, we have covered considerable ground to get you started with your infinite runner!

Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist

Checklist Checklist

Quiz

It is now time to test your knowledge. Please specify whether the following statements are TRUE or FALSE. The answers are available at the end of the book.

To add a force to an object, this object needs to include a Rigidbody or Rigidbody2D component.
It is possible to detect the keys pressed by the player, using the function
To detect collisions between two objects, both these objects need to include a Collider or a Collider2D component.
The function transform.Translate can be used to move an object to a specific location.
The function Random.Range can be used to generate a random number.
If an UI Text object is called scoreUI then the following code can be used to update its text attribute.

```
GameObject.Find (“scoreUI”).GetComponent ().text = ““ + (int)score;
```

A new sprite can be created using the menu Create | Sprites from the Project window.
Whenever a new UI object is created, a Canvas object is also automatically created and set as a parent of the former.
An object can be duplicated using CTRL +
The function Destroy can be used to destroy an object after a few seconds.

Challenge

Now that you have managed to complete this chapter and that you have created your first level, you could improve it by doing the following:

Create and instantiate additional types of obstacles based on a random number (e.g., four boxes stacked horizontally or vertically).
Modify the jumping height of the player by modifying the force applied to this object.

5

Using Classes with C#

In this section, we will go through an introduction to C# programming and look at key aspects that you will need for your games, including:

C# Syntax.
Variable types and scope.
Useful coding structures (e.g., loops or conditional statements).

So, after completing this chapter, you will be able to:

Understand key concepts related to C# programming.
Understand the concepts of variables and methods.
Create a Flappy Bird game based on the concepts covered in this chapter.

The code solutions for this chapter are included in the resource pack that you can download by following the instructions included in the section entitled and Resources for this

Introduction

...

Classes

When coding in C# with Unity, you will be creating scripts that are either classes or use built-in classes. So what is a class?

As we have seen earlier, C# is an programming (OOP) language. Put simply, a C# programme will consist of a collection of objects that interact amongst themselves. Each object has one or more attributes, and it is possible to perform actions on these objects using what are called In addition, objects that share the same properties are said to belong to the same For example, we could take the analogy of a bike. There are bikes of all shapes and colors; however, they share common features. For example, they all have a specific number of wheels (e.g., one, two or three) or a speed; they can have a color, and actions can be performed on these bikes (e.g., accelerate, turn right, turn left, etc.). So in object-oriented programming, the class would be speed or color would be referred as member variables, and accelerate (i.e., an action) would be referred as member methods. So if we were to define a common type, we could define a class called Bike and for this class define several member variables and attributes that would make it possible to define and perform actions on the objects of type

This is, obviously, a simplified explanation of classes and objects, but it should give you a clearer idea of the concept of object-oriented programming, if you are new to it.

Defining a class

So now that we have a clearer idea of what a class is, let's see how we could define a class. So let's look at the following example.

```
public class Bike
{
private float speed;
private int color;
public void accelerate()


{
speed++;
}
public void turnRight()
```

```
{
}
private void calculateDistance()
{
}
}
```

In the code above, we have defined a class, called that includes two member variables and as well as two member methods and Let's look at the script a little closer; you may notice a few things:

The name of the class is preceded by the keywords public in OOP terms, the keyword public is called an access modifier and it defines how (and from where) this class may be accessed and used. In C# there are at least five types of access modifiers, including public (no restricted access), protected (access limited to the containing class or types derived from this class), internal (access is limited to the current assembly), protected internal (we won't be using this access mode in this book), and private (access only from the containing type).
The names of all variables are preceded by their type (i.e., int), and the keyword this means that these integer variables will be accessible throughout our programme or game.

Some of the names of the methods are preceded by the keywords public void (e.g., for the methods accelerate or the void keyword means that the method does not return any data back, while the keyword public means that the method will be accessible throughout our programme.
Some of the names of the methods are preceded by the keywords private void for the method the void keyword means that the method does not return any data back, while the keyword private means that the method will be accessible only from the containing type (i.e.,

Accessing class members and variables

Accessing class methods and variables

Once a class has been defined, it's great to be able to access its member variables and methods. In C# (as for other object-oriented programming languages), this can be done using the dot

The dot notation refers to object-oriented Using dots, you can access properties and functions (or methods) related to a particular object. For example gameObject.transform.position gives you access to the position from the transform of the object linked to this script. It is often useful to read it backward; in this case, the dot can be interpreted as So in our case, gameObject.transform.position can be translated as "the position of the transform of the

Once a class has been defined, objects based on this class can be created. For example, if we were to create a new Bike object, based on the code that we have seen above, the following code could be used.

```
Bike myBike = new Bike();
```

This code will create an object based on the "template" You may notice the syntax:

```
dataType variable = new dataType()
```

By default, this new object will include all the member variables and methods defined earlier. So it will have a color and a speed, and we should also be able to access its accelerate and turnRight methods. So how can this be done? Let's look at the next code snippet that shows how we can access these.

```
Bike b = new Bike();
b.accelerate();
```

In the previous code:

The new bike myBike is created.

The speed is then increased after calling the accelerate method. This method can be called using the dot notation because it is

Note that to call an object's method we use the dot notation.

When defining member variables and methods, it is usually good practice to restrict the access to member variables (e.g., private type) and to define public methods with no or less strict restrictions (e.g., public) that provide access to these variables. These methods are often referred to as getters and setters (because you can get or set values from them).

To illustrate this concept, let's look at the following code:

```
public class Bike
{
private float speed;
private int color;

public void accelerate()
{
speed++;
}
public void turnRight()
{
}
private void claculateDistance()
{
```

———————

```
}
public void setSpeed(float newSpeed)
{
speed = newSpeed;
}
public float getSpeed()
{
```

```
return (speed)
}
}
```

In the previous code, we have declared two new methods: setSpeed and

For the type is void as this method does not return any information, and its access is set to so that it can be accessed with no restrictions.
For the type is float as this method returns the speed, which type is float. Its access is set to so that it can be accessed with no restrictions.

So, we could combine the code created to date in one programme (or new class) as follows in Unity.

```
using UnityEngine;
using System.Collections;
public class TestCode : MonoBehaviour {
public class Bike
{
private float speed;
private int color;
public void accelerate()
{
speed++;
}
public void turnRight()
{
}
private void claculateDistance()
{
```

```
}
    public void setSpeed(float newSpeed)
    {
    speed = newSpeed;
    }
    public float getSpeed()
    {
    return (speed)
    }


    }
    public void Start ()
    {
    Bike myBike = new Bike();
    myBike.setSpeed (23.0f);
    print (myBike.getSpeed());
    }
    }
```

In the previous code, you may notice at least two differences compared to the previous code that we have created:

At the start of the code, the following two lines of code have been added:

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

The keyword using is called a directive; in this particular case it is used to import what is called a put simply, by adding this directive you are effectively importing (or gaining access to) a collection of classes or data types. Each of these namespaces or “libraries” includes useful classes for

your programme. For example, the namespace UnityEngine will include classes for Unity development and System.Collections will include classes and interfaces for different collections of objects. By default, whenever you create a new C# script in Unity, these two namespaces (and associated directives) are included.
We have declared our class Bike within another class called TestCode is, in this case, the containing class.

```
public class TestCode : MonoBehaviour {
```

Whenever you create a new C# script, the name of the script (for example TestCode will be used to define the main class within the script; i.e.,
The syntax: Monobehavior means that the class TestCode is derived from the class This is often referred to as inheritance.

Constructors

As we have seen in the previous section, when a new object is created, it will, by default, include all the member variables and methods. To create this object, we would use the name of the class, followed by as per the next example.

```
Bike myBike = new Bike();
myBike.accelerate();
```

In fact, it is possible to change some of the properties of the new object created at the time it is initialized. For example, instead of setting the speed and the color of the object as we have done in the previous code, it would be great to be able to set these automatically and pass the parameter accordingly when the object is created. Well, this can be done with what is called a A constructor literally helps to construct your new object based on parameters (also referred as arguments) and instructions. So, for example, let’s say that we would like the color of our bike to be specified when it is

created; we could modify the Bike class, as follows, by adding the following method:

```
public Bike (int newColor)
{
color = newColor;


}
```

This is a new constructor (the name of the method is the same as the class), and it takes an integer as a parameter; so after modifying the description of our class (as per the code above), we could then create a new bike object as follows:

```
Bike myBike = new Bike(2);
```

We could even specify a second constructor that would include both the color and the speed as follows:

```
public Bike (int newColor, float newSpeed)
{
color = newColor;
speed = newSpeed;
}
```

You can have different constructors in your class; the constructor used at the initialization stage will be the one that matches the arguments passed.

For example, let's say that we have two constructors for our Bike class.

```
public Bike (int newColor)
{
color = newColor;
}
public Bike (int newColor, float newSpeed)
{
color = newColor;
speed = newSpeed;
}
```

If a new Bike object is created as follows:

Bike newBike = new Bike (2)

...then the first constructor will be called.

If a new Bike object is created as follows:

Bike newBike = new Bike (2, 10.0f)

...then the second constructor will be called.

You may also wonder what happens if the following code is used since no default constructor has been defined.

Bike newBike = new Bike ();

In fact, whenever you create your class, a default constructor is also defined (implicitly) and evoked whenever a new object is created using the new operator with no arguments. This is called a default constructor. In this case, the default values for each of the types of the numerical member variables are used (e.g., 0 for integers or false for Boolean variables).

Note that access to constructors is usually public, except in the particular cases where we would like a class not to be instantiated (e.g., for classes that include static members only). Also note that, as for variables, if no access modifiers are specified, these will be private by default. This is similar for methods.

Destructors

As for constructors, when an object is deleted, the corresponding destructor is called. Its name is the same as the class and preceded by a tilde as illustrated in the next code snippet.

```
~Bike()
{
//add some code here;
}
```

This being said, a destructor can neither take parameters (or arguments) nor return a value.

Static members of a class

When a method or variable is declared as static, only one instance of this member exists for a class. So a static variable will be “shared” between instances of this class. Static variables are usually used to retrieve constants without instantiating a class. The same applies for static method: they can be evoked without having to instantiate a class. This can be very useful if you want to create and avail of tools. For example, in Unity, it is possible to use the method this method usually makes it possible to look for a particular object based on its name. Let’s look at the following example.

```
public void  Start()
{
GameObject t = (GameObject) GameObject.Find(“test”);
}
```

In the previous code, we look for an object called test, and store the result inside the variable t of type However, when we use the syntax we use the static method Find that is available from the class There are many other static functions that you will be able to use in Unity, including Again, these functions can be called without the need to instantiate an object. The following code snippet provides another example based on the class

```
using UnityEngine;
using System.Collections;
public class TestCode : MonoBehaviour {
public class Bike
{
private float speed;
private int color;


private float speed;
private int color;
private static int nbBikes;
```

```
public void countBikes()
{
nbBikes++;
}
public int getNbBikes()
{
return (nbBikes);
}
}
void Start ()
{
Bike bike1 = new Bike();
Bike bike2 = new Bike();
bike1.countBikes();
bike2.countBikes();
print("Nb Bikes:"+bike1.getNbBikes());
}}
```

The following code illustrates the use of static functions.

```
using UnityEngine;
using System.Collections;
public class TestCode : MonoBehaviour
{
public class Bike
{
private float speed;
private int color;
public static void sayHello()
{
print("Hello");
}
}
public void Start()
{
Bike.sayHello();
}
}
```

The previous code would result in the following output:

Hello

In the previous code, we declare a static method called this method is then called in the start method without the need to instantiate (or create) a new This is because, due to its public and static attributes, it can be accessed from anywhere in the programme.

Inheritance

I hope everything is clear so far, as we are going to look at a very interesting and important principle for object-oriented programming: inheritance. The main idea of inheritance is that objects can inherit their properties from other objects (their parent). As they inherit these

properties, they can remain identical or evolve and overwrite some of these inherited properties. This is very interesting because it makes it possible to minimize the code by creating a class with general properties for all objects sharing similar features, and then, if need be, to overwrite and customize some of these properties.

Let's take the example of vehicles; they would generally have the following properties:

Number of wheels.
Speed.
Number of passengers.
Color.
Capacity to accelerate.
Capacity to stop.

So we could create the following class for example:

```
class Vehicles
{
private int nbWheels;
protected float speed;
private int nbPassengers;
private int color;
public void accelerate()
{
speed++;
}
}
```

These features could apply for example to cars, bikes, motorbikes, or trucks. However, all these vehicles also differ; some of them may or may not have an engine or a steering wheel. So we could create a subclass

called based on but with specificities linked to the fact that they are motorized. These added attributes could be:

Engine size.
Petrol type.
Petrol levels.
Ability to fill-up the tank.

The following example illustrates how this class could be created.

```
class MotoredVehicles: Vehicles
{
private float engineSize;
private int petrolType;
private float petrolLevels;
public void fillUpTank()
{
petrolLevels+=10;
}
}
```

When the class is defined, its name is followed This means that it inherits from the class So it will, by default, avail of all the methods and variables already included in the class
We have created a new member method for this class, called

In the previous example, you may notice that the methods and variables that were defined for the class Vehicles do not appear here; this is because they are implicitly added to this new class, since it inherits from the class

Whenever you create a new class in Unity, it will, by default, inherit from the MonoBehaviour class; as a result, it will implicitly include all the

member methods and variables of the class Some of these methods include Start or for example.

When using inheritance, the parent is usually referred to as the base while the child is referred to as the inherited

Now, while the child inherits “Behaviors” from its parents, these can always be modified or, put simply, overwritten. However, in this case, the base method (the method defined in the parent) must be declared as virtual. Also, when overriding this method, the keyword override must be used. This is illustrated in the following code.

```
class Vehicles
{
private int nbWheels;
protected float speed;
private int nbPassengers;
private int color;
public virtual void accelerate()
{
speed++;
}
}
class MotoredVehicles : Vehicles


{
private float engineSize;
private int petrolType;
private float petrolLevels;
private void fillUpTank()
{
petrolLevels += 10;
}
public override void accelerate()
{
```

```
speed += 10;
}
}
```

In the previous example, while the method accelerate is inherited from the class it would normally increase the speed by one. However, by overwriting it, we make sure that in the case of objects instantiated from the class each acceleration increases the speed by 10. We can access the member variable speed from the child class MotoredVehicles because it has been declared as

This point can also be illustrated using some classes in Unity. Let's look at the next example.

```
using UnityEngine;
using System.Collections;
public class TestCode : MonoBehaviour {
public class Bike
{
private float speed;
private int color;


public static void sayHello()
{
print ("Hello");
}
}
public void Start ()
{
Bike.sayHello();
}
}
```

In this example, we have created a class this class inherits from by default this class includes, amongst other things, a definition for the methods Start and however, by default, these two methods are blank; this

is the reason why we overwrite these methods for the class TestCode (inherited from so that the Start method actually displays some information.

There are obviously more concepts linked to inheritance; however, the information provided in this section should get you started easily. For more information on inheritance in C#, you can look at the official

---

Methods

Methods or functions can be compared to a friend or colleague to whom you gently ask to perform a task, based on specific instructions, and to return the information to you then. For example, you could ask your friend the following: you please tell me when I will be celebrating my 20th birthday given that I was born in So you give your friend (who is good at Math :-)) the information (date of birth) and s/he will calculate the year of your 20th birthday and give this information back to you. So in other words, your friend will be given an input (i.e., the date of birth) and return an output (i.e., the year of your 20th birthday). Methods work exactly this way: they are given information (and sometimes not), perform an action, and then (sometimes, if needed) return information back.

In programming terms, a method (or function) is a block of instructions that performs a set of actions. It is executed when invoked (or put more simply from the script, or when an event occurs (e.g., the player has clicked on a button or the player collides with an object; we will see more about events in the next section). As for member variables, member functions or methods are declared and they can also be called.

Methods are very useful because once the code for a method has been created, it can be called several times without the need to re-write the same code over and over again. Also, because methods can take parameters, a method can process these parameters and produce or return

information accordingly; in other words, they can perform different actions and produce different information based on the input. As a result, methods can do one or all of the following:

- Perform an action.
- Return a result.
- Take parameters and process them.

A method has a syntax and can be declared as follows (in at least two ways).

AccessType typeOfdataReturned nameOfTheFunction ()

{
Perform actions here...
}

In the previous code the method does not take any input; neither does it return an output. It just performs actions.

OR

AccessType typeOfDataReturned nameOfTheFunction ()
{
Perform actions here...
}

Let’s look at the following method for instance.

```
public int calculateSum(int a, int b)
{
return (a+b);
}
```

In the previous code:

The method is of type there are no access restrictions.
The method will return an integer.

The name of the method is
The method takes two arguments (i.e., integer parameters).
The method returns the sum of the two parameters passed (the parameters are referred to as a and b within this method).

A method can be called using the () operator, as follows:

```
nameOfTheFunction1();
nameOfTheFunction2(value);
```

```
int test = nameOfTheFunction3(value);
```

In the previous code, a method is called with no parameter (line 1), or with a parameter (line 2). In the third example (line 3), a variable called test will be set with the value returned by the method

You may, and we will get to this later, have different methods in a class with the exact same name but that take different types of parameters. This is often referred as polymorphism, as the method literally takes different forms and can process information differently based on the information (e.g., type of data) provided.

Accessing methods and access modifiers

As we have seen previously, in C# there are different types of access modifiers. These modifiers specify from where a method can be called and can be public (no restricted access), protected (access is limited to the containing class or types derived from this class), internal (access is limited to the current assembly), protected internal (we won't use this access type in this book), and private (i.e., access is limited to the containing type).

Common methods

In Unity, there are many methods available by default, and they are called built-in methods. Some of these functions are called when an event occurs. For example:

called at the start of the scene.

called every time the screen is refreshed.
called whenever the First-Person Controller (a built-in controller used to make it possible to navigate through your scene using a first-person view) collides with an object.

As we will see later, because most of your classes will inherit by default from the class they will, by default, include several methods, including Start and Update that you will be able to override (i.e., modify for your own use).

Scope of variables

Whenever you create a variable in C#, you will need to be aware of the scope and access type of the variable so that you use it where its scope makes it possible for you to do so.

The scope of a variable refers to where you can use this variable in a script. In C#, we usually make the difference between global variables and local

You can compare the term local and global variables to a language that is either local or global. In the first case, the local language will only be used (i.e., spoken) by the locals, whereas the global language will be used (i.e., spoken) and understood by anyone whether they are locals or part of the global community.

When you create a class definition along with member variables, these variables will be seen by any method within your class.

Global variables are variables that can be used anywhere in your script, hence the name global. These variables need to be declared at the start of the script (using the usual declaration syntax) and outside of any method; they can then be used anywhere in the script as illustrated in the next code snippet.

```
class MyBike
{
private int color;
```

```
private float speed;
public void accelerate()


{
speed++;
}
}
```

In the previous code we declare the variable speed as a member variable and access it from the method accelerate.

Local variables are declared within a method and are to be used only within this method, hence the term local, because they can only be used locally, as illustrated in the next code snippet.

```
public void  Start()
{
int myVar;
myVar = 0;
}
public void Update()
{
int myVar2;
myVar2 = 2;
}
```

In the previous code, myVar is a local variable to the method and can only be used within this function; myVar2 is a local variable to the method and can only be used within this method.

Events

Throughout this book and in C#, you will read about and use events. So what are they?

Well, put simply, events can be compared to something that happens at a particular time, and when this event occurs, something (e.g., an action) needs to be done. If we make an analogy with daily activities: when the

alarm goes off (event) we can either get-up (action) or decide to go back to sleep. When you receive an email (event), you can decide to read it (action), and then reply to the sender (another action).

In computer terms, it is quite similar, although the events that we will be dealing with will be slightly different. So, we could be waiting for the user to press a key (event) and then move the character accordingly (action), or wait until the user clicks on a button on screen (event) to load a new scene (action).

In Unity, whenever an event occurs, a function (or method) is usually called (the function, in this case, is often referred as a handler, because it "handles" the event). You have then the opportunity to modify this function and add instructions (i.e. statements) that should be followed, should this event occur.

To take the analogy of daily activities: we could write instructions to a friend on a piece of paper, so that, in case someone calls in our absence, the friend knows exactly what to do. So an event handler is basically a set of instructions (usually stored within a function) to be followed in case a particular event occurs.

Sometimes information is passed to this method about the particular event, and sometimes not. For example, when the screen is refreshed the method Update is called. When the game starts (i.e., when a particular script is enabled), the method Start is called. When there is a collision between the player and an object, the method OnControllerColliderHit is called. In this particular case (i.e., collision), an object is usually passed to the method that handles the event so that we get to know more about the other object involved in the collision.

As you can see, there can be a wide range of events in our game, and we will get to that later on. In this book, we will essentially be dealing with the following events:

when a script is enabled (e.g., start of the scene).

when the screen is refreshed (e.g., every frame).
when a collision occurs between the player and another object.
when the game starts (i.e., once).

Polymorphism (general concepts)

The word polymorphism takes its meaning from poly (several) and morph (shape); so it literally means several forms. In object-oriented programming, it refers to the ability to process objects differently (or more specifically) depending on their data type or class. Let's take the example of adding. If we want to add two numbers, we will make an algebraic addition (e.g., 1 + 2). However, adding two string variables may mean concatenating them (adding them one after the other). For example, adding the text "Hello" and the text "World" would result in the text As you can see, the way an operation is performed on different data types may vary and produce different results. So again, with polymorphism we will be able to customize methods (or operations) so that data is processed based on its type of class. So, let's look at the following code which illustrates how this can be done in C#.

```
public class AddObjects
{


public int add (int a, int b)
{
return (a + b);
}
public string add (string a, string b)
{
return (a + b);
}
}
```

In the previous code, it is possible to add two different types of data: integers and strings. Depending on whether two integers or two strings are

passed as parameters, we will be calling either the first add method or the second add method.

Dynamic polymorphism

In C#, dynamic polymorphism can be achieved using both abstract classes and virtual functions.

In C#, it is possible to create a class that will provide a partial implementation of an interface. Broadly, an interface defines what a class should include (i.e., member methods, member variables or events), but it does not declare how these should be implemented. So, an abstract class will include abstract methods or variables; which means that this class will define the name and type of the variables, the name of the methods, as well as the type of data returned by this method. It is called abstract because you cannot implement this type of class (it can never be "materialized"); however, it can be used as a template (or "dream" class) for derived classes. Let's look at the following example.

```
abstract class Vehicule


{
public abstract void decelerate();
}
class Bike : Vehicule
{
private float speed;
private int color;
public Bike(float newSpeed)
{
speed = newSpeed;
}
public override void decelerate()
{
speed—;
}
```

}

In the previous code:

We declare an abstract class
We declare an abstract method called
We then create a new class called inherited from the abstract class Vehicle, but with its own constructor.
We then override the abstract method decelerate to use our own implementation.

Using an abstract class just means that we list methods that would be useful for the children; however, the children will have to define how the method should be implemented.

The second way to implement dynamic polymorphism is by using virtual methods or variables. In the case of virtual we declare a method that will be used by default by objects of this class or inherited classes; however, in this case, even if the method is ready to be used (i.e., because we have defined how it should be implemented), it can be changed (or overridden) by the child (i.e., the inherited class) to fit a specific purpose. In this case (i.e., inherited method), we need to specify that we override this method using the keyword

The key difference between an abstract and a virtual method is that, while an abstract method should be overridden, a virtual methods may be overridden if the base method (i.e., the method declared in the base class) does not suit a particular purpose.

Let's look at an example:

```
class Vehicle
{
protected float speed;
public virtual void accelerate()
```

```
{
speed += 10;
}
public Vehicle(float newSpeed)
{
speed = newSpeed;
}
}
class Bike : Vehicle
{
public override void accelerate()
{
speed++;


}
public Bike(float newSpeed) : base(newSpeed){}
}
```

In the previous code:

We declare a class

It includes both a protected variable speed and a virtual method called

This method is virtual, which means that inherited classes will be able to modify (override) it, if need be.

The class Bike has its own constructor; however, this constructor inherits from the parent's constructor; this means that by calling the constructor of the class Bike, we effectively call the parent's (of the base's) constructor, using the syntax:

```
public Bike(float newSpeed) : base(newSpeed){}
```

We then create a new class Bike that inherits form the class In this class, we override the method accelerate using the keyword override so that the speed is just incremented by one.

NameSpaces

When you create a new script in Unity, it usually includes the following lines at the top of the script automatically.

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

This code is effectively specifying the location of some classes that you may use in your code, and provides information about what namespace (which is comparable to a folder or directory) a specific class belongs to. This is to avoid any clash or confusion and to ensure that even if a class is declared in two different namespaces, that it is clear as to which class (and namespace) you want to use.

So what is a name space?

Namespaces are containers within which you can declare classes; let's look at the following example:

```
namespace NameSpace1
{
public class MyClass
{
public static int add(int a, int b)
{
return (a + b);
}
}
}
namespace NameSpace2
{
```

```
public class MyClass
{
public static int add(int a, int b, int c)
{
return (a + b + c);
}
}
}
```

In the previous code:

We declare two namespaces called NameSpace1 and
Within these namespaces, that act as containers, we declare a class called
So we have two classes with the exact same name (i.e., however, we can manage to tell them apart based on their namespace (or their container).

Now that the two different namespaces have been defined, we need to find a way to specify which namespace will be used when classes with the same name are used or instantiated.

This can be done in at least two ways. The first way is to implicitly mention the namespace and the nested class within. For example, we could write the following code to refer to the first class, as illustrated in the next code snippet.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class MyNAmeSpacesExample : MonoBehaviour
{
void Start ()
```

```
{
int sum1 = NameSpace1.MyClass.add(1,2);
int sum2 = NameSpace2.MyClass.add(1,2,3);
print ("Sum1:" + sum1 + "; Sum2:" + sum2);
}
}
```

In the previous code:

We access the method that is both public and static, from the class MyClass that is within the namespace

We also access the method add (that is both public and static) from the class MyClass that is within the namespace

The add method that is used in the previous example is so it can be accessed from anywhere in our programme; it is also static so it can be accessed without the need to instantiate an object of the class

Another way to do this is to specify, from the very start of the script, that we will be using the namespaces NameSpace1 or and this can be done with the keyword called using as follows:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using NameSpace1;
public class MyNameSpacesExample : MonoBehaviour
{
void Start ()
{
int sum1 = MyClass.add(1,2);
print ("Sum1:" + sum1);
}
```

}

In the previous code:

We declare (or we make a reference to) the namespace
This means that, if in doubt about where to find a particular class, it may be found in this namespace.
We then call the method however, because we have defined a reference to the namespace the system will automatically look into this namespace to find the class MyClass and the method

So how can namespaces be used in Unity?

Using namespaces in Unity can make your code more concise and it can also save you a lot of time. For example, let's say that you want to write some text onscreen through UI objects (such as UI Text objects) from your script. In this case, you may use code that is similar to the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class NameSpaceForUI : MonoBehaviour
{
// Use this for initialization
void Start () {
GameObject.Find ("TextUI").GetComponent ().text = "";
GameObject.Find ("scoreUI").GetComponent ().text = "Score:";
}
}
```

In the previous code, we declare that we will be using the UI namespace with the code:

using UnityEngine.UI;

If we had not used the namespace the following line

GameObject.Find ("TextUI").GetComponent ().text = "";

may have needed to be written as follows instead:

GameObject.Find ().text = "";

So by specifying the namespace UnityEngine.UI we can shorten our code and use GetComponent or GetComponent as both the Text and Slider classes are part of the namespace

More often than not, you may not need to use additional namespaces in Unity unless you create your own library. However, as illustrated previously, it can be useful to shorten your code when classes from the same namespace are employed several times.

When creating your scripts, you may wonder when you need the following statements:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

Whether you will need to use these lines depends on what you need to perform in your code:

The namespace most of your scripts will be linked to and will therefore need to extend the class since the class MonoBehaviour belongs to the UnityEngine namespace, you will, in most cases, need this line. In addition, several useful classes are provided within the UnityEngine namespace.

The namespace this namespace includes many basic classes including IEnumerators which are often used for coroutines.

The namespace this namespace includes some useful classes provided by C# including Lists or Dictionaries.

So, if you need to keep one of these, you may keep although, it is good practice to keep all three lines, just in case they may be needed later in your class.

Level Roundup

In this chapter, we have learned some concepts related to C# and Object-Oriented Programming. We also learnt how to define and to use classes, along with member variables and methods. Along the way, we also looked at other programming concepts such as variables, loops, and conditional statements. So, we have covered considerable ground to get you started with C#!

Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist
Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist
Checklist Checklist Checklist Checklist Checklist Checklist Checklist
Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Quiz

It is now time to test your knowledge. Please specify whether the following statements are TRUE or FALSE. The answers are available at the end of the book.

The value of a variable always remains constant.
A method always returns information.
A method may not return information.
If a method is void, it will return an integer value.
An array can store several variables at a time.
A class usually includes a constructor.
A for loop can be used to go through all the elements of an array.
A public method is accessible from anywhere.
A private variable is accessible only from members of the class.
A protected variable is accessible only from members of the class.

6 Creating a Simple 2D Shooter

In this section, we will start by creating a simple level, including:

A spaceship symbolized by a triangle that you will be able to move in four directions.
The ability for the spaceship to fire missiles.
The ability for the player to destroy targets with the missiles.
A camera that displays the scene.
Meteorites (or moving targets) generated randomly.

So, after completing this chapter, you will be able to:

Detect keystrokes.
Generate random events.
Instantiate objects.
Add velocity to objects (i.e., to the moving targets).
Modify sprites' properties such as their color.
Move objects from a script.

Adding the spaceship

So, in this section, we will start to create the spaceship that will be used by the player; it will consist of a simple sprite (for the time-being) that we will be able to move in four directions using the arrow keys on the keyboard: left, right, up and down.

Please create a new scene

If you have not already done so, please import the packages 2D Sprite and 2D Spite

Once this is done, you can click on the button called Create project (at the bottom of the window) and Unity should
Once this is done, you can check that the 2D mode is activated, based on the 2D logo located in the top right-corner of the Scene view.

Figure 6-1: Activating the 2D mode

We will now create a new sprite for our spaceship; it will be made of a simple triangle.

From the Project view, please select Create | 2D | Sprites |
This will create a new asset called Triangle in the Project window.

Figure 6-2: Creating a new triangle asset

Once this is done, you can drag and drop this sprite (i.e., the white object with the label from the Project window to the Scene view; this will create a new object called Triangle in the Hierarchy view.

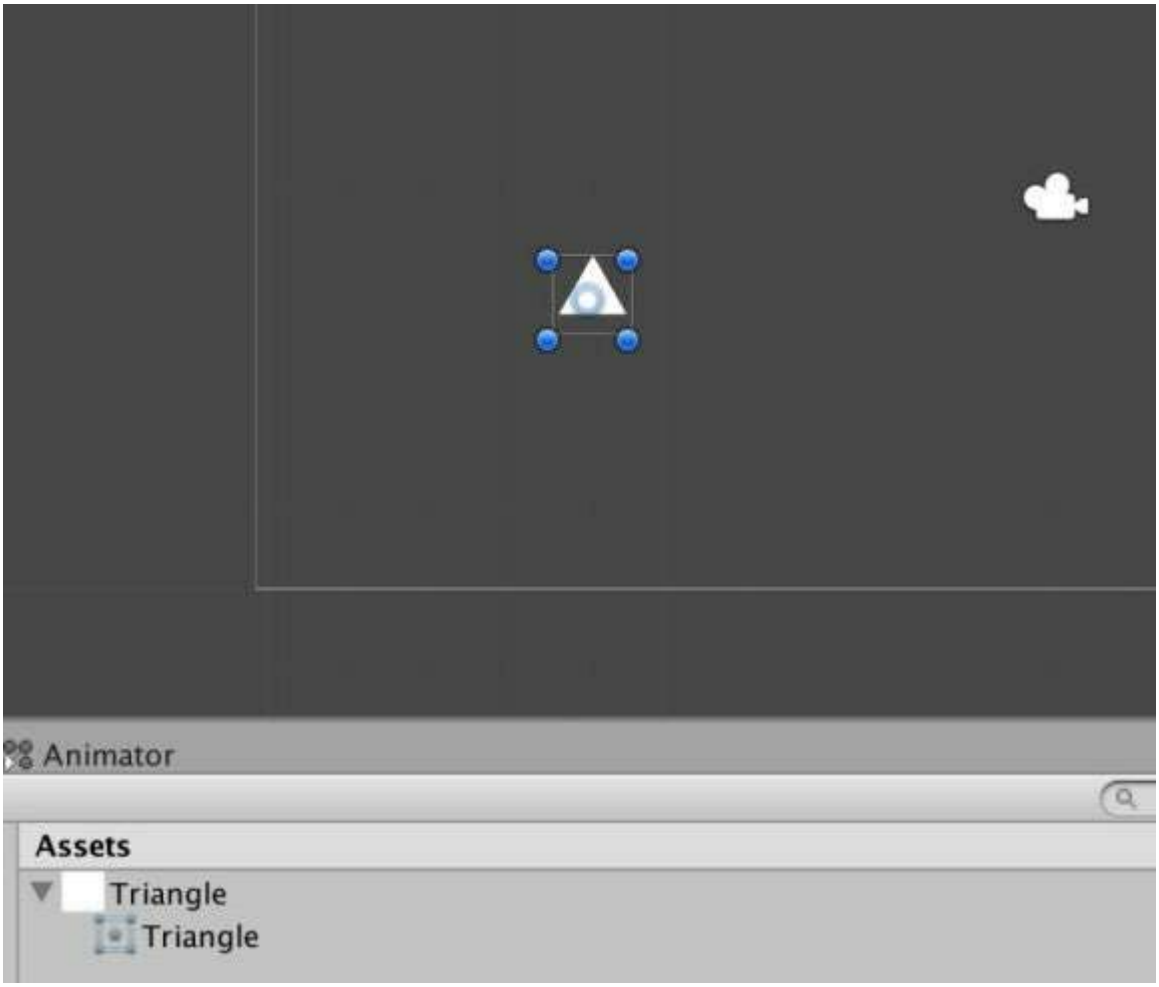


Figure 6-3: Adding the player character

In the previous window, you may notice the white lines at the bottom and to the left of the screen; these are the boundaries that define what will be visible onscreen; so by dropping your object within these lines, you ensure that the player will be seen (or captured) by the camera.

Please rename this object player for now, using the Hierarchy window: to rename this object, you can right-click on it in the Hierarchy window, and then select the option Rename from the contextual menu.
Change its position to (0, 0,

So at this stage, we have a new player character (i.e., the spaceship) and we will need to move it according to the keys pressed on the keyboard; so let’s do just this:

Please create a new C# script (i.e., select Create | C# Script from the Project window) and rename this script
Once this is done, please open this script and add the following code to it (new code in bold):

```
void Update ()
{
if (Input.GetKey (KeyCode.LeftArrow))
{
gameObject.transform.Translate (Vector3.left * 0.1f);
}
if (Input.GetKey (KeyCode.RightArrow))
{
gameObject.transform.Translate (Vector3.right * 0.1f);
}
if (Input.GetKey (KeyCode.UpArrow))
{
gameObject.transform.Translate (Vector3.up * 0.1f);
}
if (Input.GetKey (KeyCode.DownArrow))
{
gameObject.transform.Translate (Vector3.down * 0.1f);
}
}
```

In the previous code:

We use the function Update to check for keyboard inputs.

If the left arrow is pressed, we move the object linked to this script (i.e., the spaceship) to the left (i.e., 0.1 meter to the left).
If the right arrow is pressed, we move the object linked to this script (i.e., the spaceship) to the right (i.e., 0.1 meter to the right).

Note that we use the function GetKey that checks whether a key has been pressed; however, if you wanted to check whether a key has been released then you could use the function GetKeyDown instead.

You can now save the script, check for any error in the Console window, and link the script (i.e., drag and drop it) to the object called player that is in the Hierarchy view. Once this is done, you can play the scene and check that you can move the player left or right. After pressing the arrow keys on your keyboard, you should see that the spaceship moves in four directions.

Note that to play and stop the scene, you can press the shortcut CTRL + or use the black triangle located at the top of the window.

Shooting missiles

In this section, we will get the player to shoot missiles whenever s/he presses the space bar; this will involve the following steps:

Creating an object for the missile.
Saving this object as a prefab (i.e., a template).
Detecting when the space bar has been pressed (and then released) by the player.

Instantiating the missile prefab and adding velocity to it so that it moves up when fired.

First, let’s create a new object for the missile:

You can now stop the scene (e.g., CTRL + P).
Using the Project window, please create a new circular sprite | 2D | Sprites |
Once this is done, please drag and drop this asset from the Project window to the Scene (or window, this will create a new object: rename that object Using the rescale this object to (0.1, 0.1, The position of this object does not matter for now.

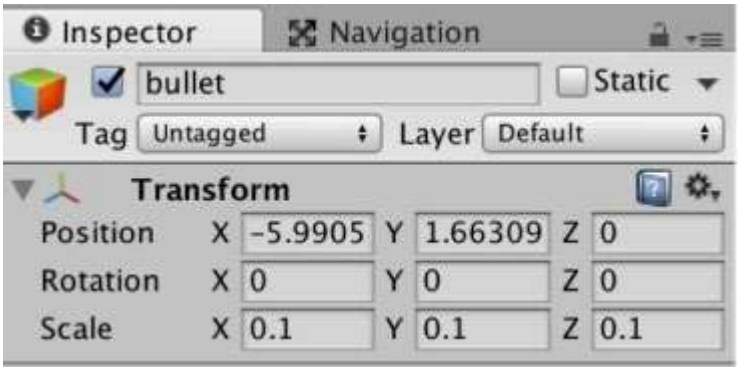


Figure 6-4: Scaling-down the bullet

Add a Rigidbody2D component to this object (i.e., select Components | Physics2D | Rigidbody2D from the top menu) and set the Gravity Scale attribute of this component (i.e., to 0, as illustrated in the next figure.

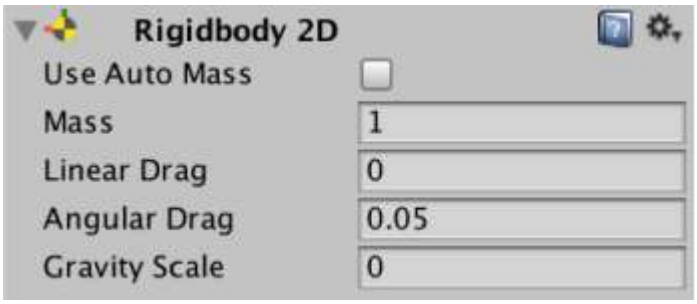


Figure 6-5: Setting the gravity scale

By adding a Rigidbody2D component to this object, we ensure that we can apply forces to it, or modify its velocity; this being said, because we have a top-down view, we do not want this object to be influenced by

gravity (otherwise it would fall down), and this is why we set the Gravity Scale attribute to 0 for this object.

We can now convert this bullet to a prefab by dragging and dropping this object (i.e., to the Project view.

Figure 6-6: Creating a prefab for the bullet

You can now delete the object called bullet from the

Last but not least, we need to add some code that will be used to instantiate and propel this bullet if the player presses the space bar.

Please open the script called
Add the following code at the beginning of the script (new code in bold).

```
public class MovePlayer : MonoBehaviour
{
public GameObject bullet;
```

Please add the following code to the Update function:

```
if (Input.GetKeyDown (KeyCode.Space))
{
GameObject b = (GameObject)(Instantiate (bullet, transform.position + transform.up*1.5f, Quaternion.identity));
b.GetComponent ().AddForce (transform.up * 1000);
}
```

In the previous code:

We create a new
This GameObject will be based on the template called
If the player hits the space bar, the new bullet is instantiated just above the spaceship.
We then add an upward force to the bullet so that it starts to move.

You can now save your script, and check that it is error-free in the Console window.

If you click on the object called player that is present in the and if you look at the you should see that a new field called bullet has appeared for the component

Please drag and drop the prefab called bullet to this field (as illustrated on the next figure).

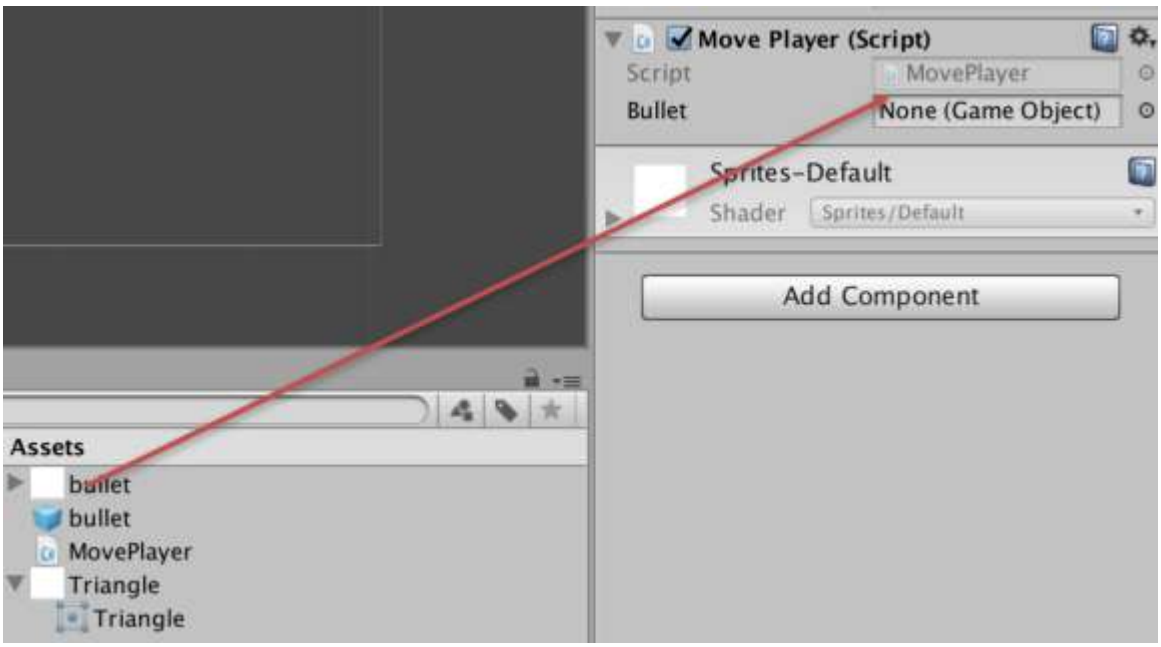


Figure 6-7:Adding the bullet prefab

Once this done, you can play the scene, and check that after pressing the space bar, you are able to fire a bullet.

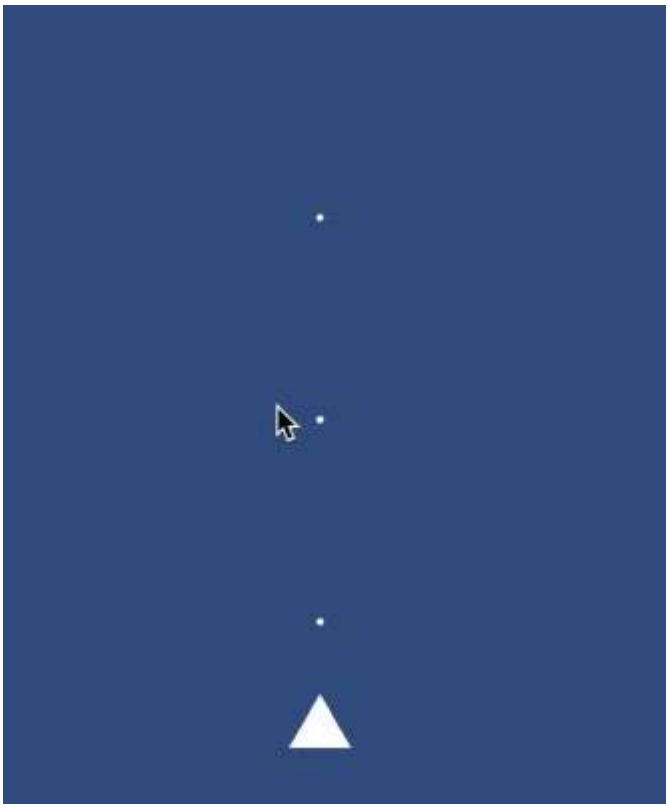


Figure 6-8: Shooting projectiles

Destroying the target

Now that we can shoot missiles (or bullets), we just need to be able to destroy the objects colliding with the missiles; so we will create new objects that will be used as targets for the time being.

Please create a new Square sprite (from the Project window, select: Create 2D | Sprites |

Drag and drop this sprite to the Scene view.

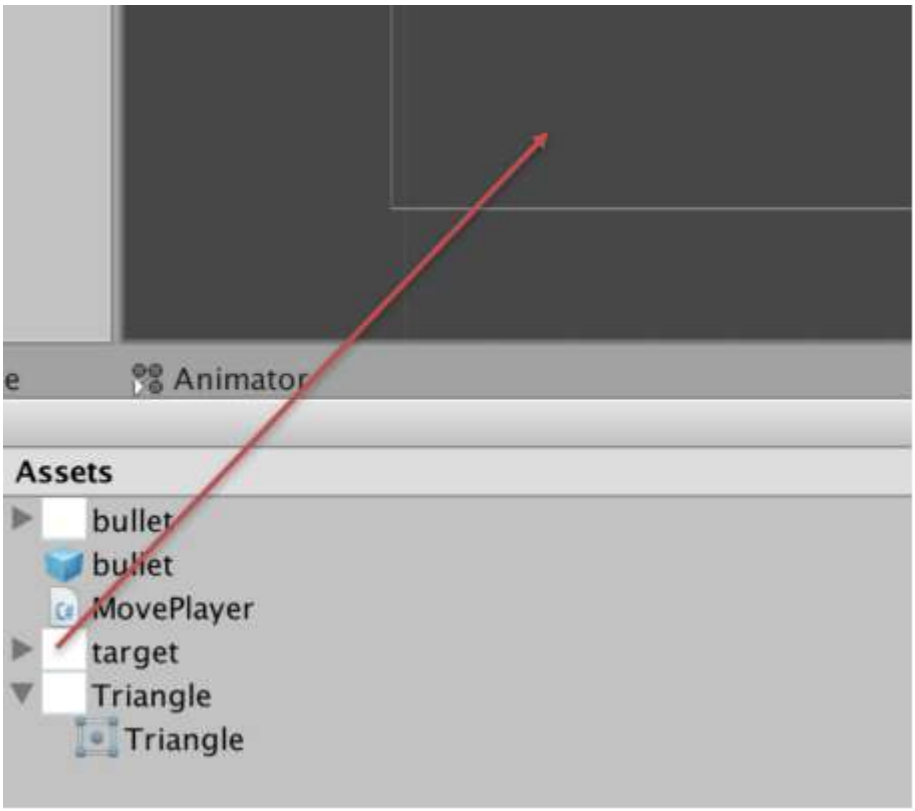


Figure 6-9: Adding a target to the scene

This will create a new object; rename this new object
Please select this object.

Add a BoxCollider2D to this object (i.e., select Components | Physics2D | BoxCollider2D from the top menu). This is so that collisions can be detected.

We will now create a new tag for this object. A will help to identify each object in the scene, and to see the object that the bullets (or the player) are colliding with.

Please select the object called target in the

In the Inspector window, click on the drop-down menu called Untagged (to the right of the attribute called tag), as described on the next figure.

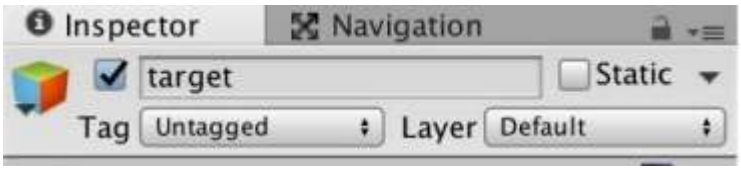


Figure 6-10: Creating a tag (part1)

From the drop-down menu, please select the option Add Tag...

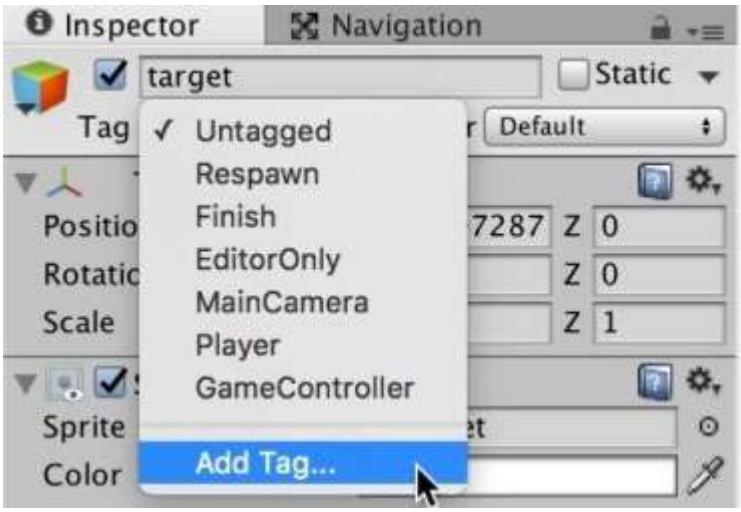


Figure 6-11: Creating a tag (part 2)

In the new window, click on the + button that is located below the label is

Figure 6-12: Creating a tag (part 3)

Please specify a name for your tag (i.e., using the field to the right of the label Tag

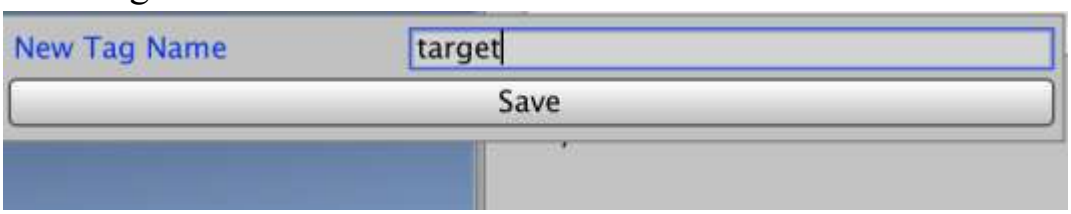


Figure 6-13: Adding a tag (part 2)

Press the Enter/Return key on your keyboard to save your new tag.
Select the object target in the Hierarchy again, and, using the select the tag that you have just created.

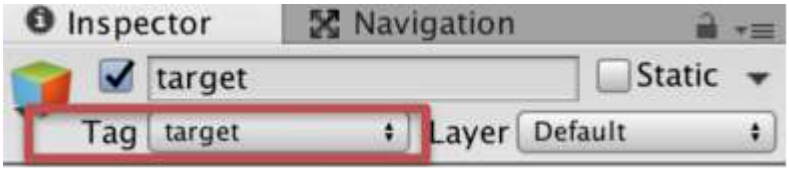


Figure 6-14: Adding a tag (part 3)

Last but not least, we will create a prefab from this target by dragging and dropping the object target to the Project window.

Next, we will create a new script that will be linked to the bullet (or missile), so that, upon collision with a target, this target should be destroyed (based on its tag).

Please create a new script called from the Project window, select Create | C#
Open this script.
Add the following code to it (just after the function

```
void OnCollisionEnter2D(Collision2D coll)
{
if (coll.gameObject.tag == "target")
{
Destroy (coll.gameObject);
Destroy (gameObject);
}
```

}

In the previous code:

We detect the objects colliding with the bullet.
When this occurs, we check if this object is a target; if this is the case, this target is then destroyed.
The bullet is also destroyed in this case.

Once this is done, we can save our script and link it to the bullet prefab.

Please save the script called Bullet and check that it is error-free.
Once this is done, please drag and drop it on the prefab called in the Project window.

You can then click once on the prefab called and check, using the Inspector window, that it includes the script

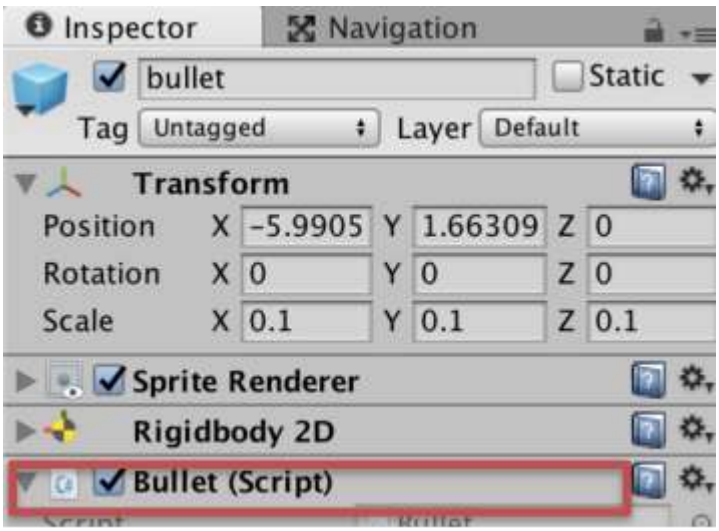


Figure 6-15: Checking the components of the Bullet prefab

Last but not least, we will need to add a collider to our Bullet prefab, so that it actually collides with other objects:

Please select the prefab called
From the top menu, select Components | Physics2D |

You can now test your game:

Add a target to your scene (by dragging and dropping a target prefab) if no targets are in the scene yet.
Move the target object just above the as illustrated in the next figure.

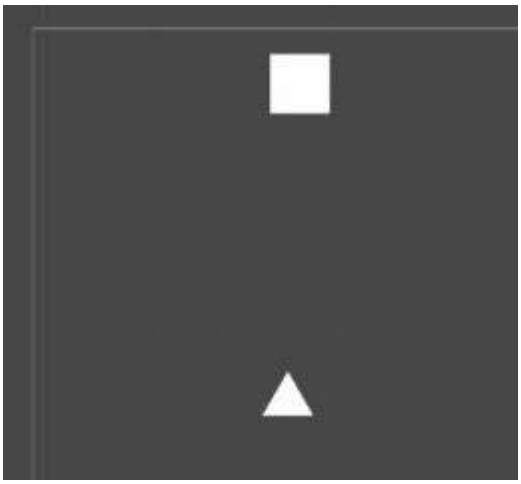


Figure 6-16: Checking the bullet prefab

Please play the scene, fire a missile (i.e., press the space bar), and check that, upon collision between the bullet and the target, both objects are destroyed.

Note that since you will be firing several bullets, we could choose to destroy a bullet after 10 seconds (by this time it should have hit a target), by modifying the script Bullet as follows (new code in bold):

```
void Start ()
{
Destroy (gameObject, 10);
```

}

You can test your scene and see that after 10 seconds the bullet is destroyed.

Before we go ahead, it may be a good idea to save our scene:

Please select File | Save Scene As from the top menu, and save your scene as
You can also save your project | Save

Next, we will just create a slightly different type of target; that is: a moving target that will move downwards and that the player will have to avoid or to destroy; so let's implement this feature:

Using the Project window, please duplicate the prefab called that we have just created (i.e., select the target prefab, and the press CTRL +
Rename the duplicate moving_target (i.e., right-click +
Select the prefab moving_target in the Hierarchy and add a RigidBody2D component to it (i.e., select Component | Physics2D |
Using the Inspector window, set its attribute called Gravity Scale (for the component to 0, as illustrated on the next figure. This is so that the object does not fall indefinitely (since it is a top-down view).

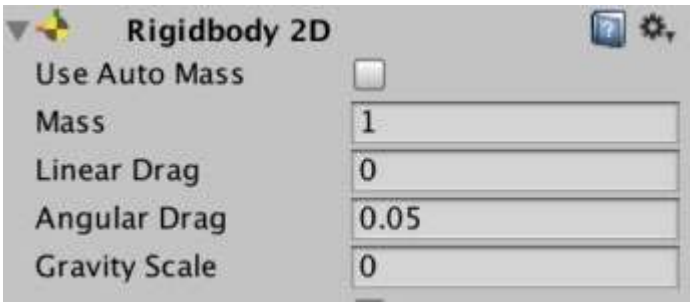


Figure 6-17: Adjusting the gravity scale

Next, we will create a script that will be linked to this object and that will set its initial velocity downwards.

Please create a new C# script called
Modify the Start function as follows (new code in bold).

```
void Start ()
{
GetComponent ().velocity = Vector2.down * 10;
}
```

In the previous code, we access the Rigidbody2D component of the object linked to this script (this will be the moving target), and then set the velocity downwards.

You can now save your script, check that it is error-free, and drag and drop it to the prefab called

Figure 6-18: Linking the script to the target

So that we can test the scene, please drag and drop the prefab moving_target to the Scene view and play the scene, you should see that this particular target moves downwards.

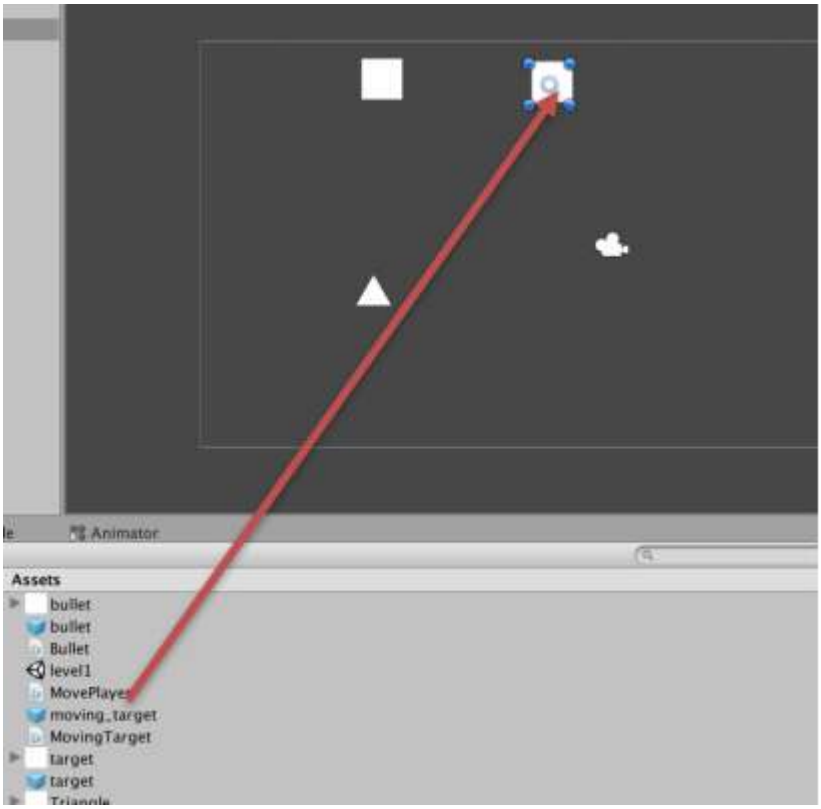


Figure 6-19: Adding a moving target to the scene

Spawning moving targets randomly

Last but not least, we will create a mechanism through which the moving targets are created randomly, a bit like meteorites, so that flying "meteorites" appear randomly onscreen and move downwards. For this, we will be doing the following:

We will create an empty object that will spawn these moving targets. These will be instantiated at regular intervals and at random positions.

We will also ensure that the moving targets are spawned in the current view (i.e., relatively close to the player so that they can be captured and displayed by the camera).

So let's get to it:

Please create a new empty object called targetSpawner in the Hierarchy window (i.e., select GameObject | Create
Create a new C# script called
Open the script.
Add the following code at the beginning of the class (new code in bold):

```
public class SpawnMovingTargets : MonoBehaviour {
float timer = 0;
public GameObject newObject;
```

Add the following code to the Update function (new code in bold):

```
void Update ()
{
timer += Time.deltaTime;
float range = Random.Range (-10, 10);
Vector3 newPosition = new Vector3
(GameObject.Find("player").transform.position.x + range,
transform.position.y, 0);
if (timer >= 1)
{
GameObject t = (GameObject)(Instantiate (newObject, newPosition,
Quaternion.identity));
timer = 0;
}


}
```

In the previous code:

We increase the value of our timer every seconds.

We then define a variable called it will be a random number between -10 and this variable will be used to define a random position that is to the left (i.e., -10 to or to the right (i.e., 0 to of the player; this is so that the target instantiated is close enough to the player and within the field of the view of the camera.
We then create a new vector called newPosition that uses the variable range defined earlier for the x coordinate; the y-coordinate of the object linked to this script (this will be the empty object is then used for the y-coordinate of the object that is being instantiated.
A new object is then instantiated every second: every time the value of the variable timer is greater than 1, timer is reset to 0 and a new prefab (i.e., moving target) is instantiated

There are of course many other ways to create this feature, but this version is relatively simple, to start with.

Next, we just need to set-up the targetSpawner object:

Please check that the script that you have just created is error-free.

Drag and drop this script (i.e., to the object called targetSpawner in the Alternatively, you can add the script to the object targetSpawner by selecting this object in the and by then dragging and dropping the script (i.e., to the Inspector window, as illustrated on the next figure.

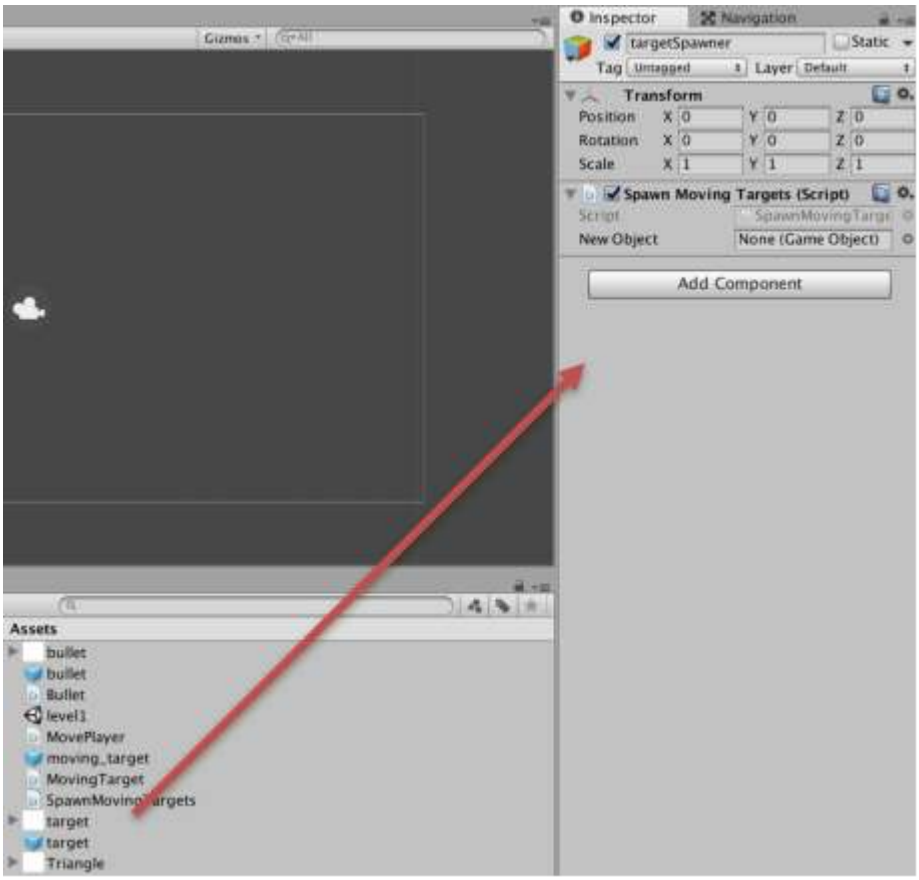


Figure 6-20: Adding a script to the object targetSpawner

Please select the object targetSpawner in the Hierarchy window.
Drag the prefab called moving_target from the Project window to the field called newObject in the as described in the next figure.

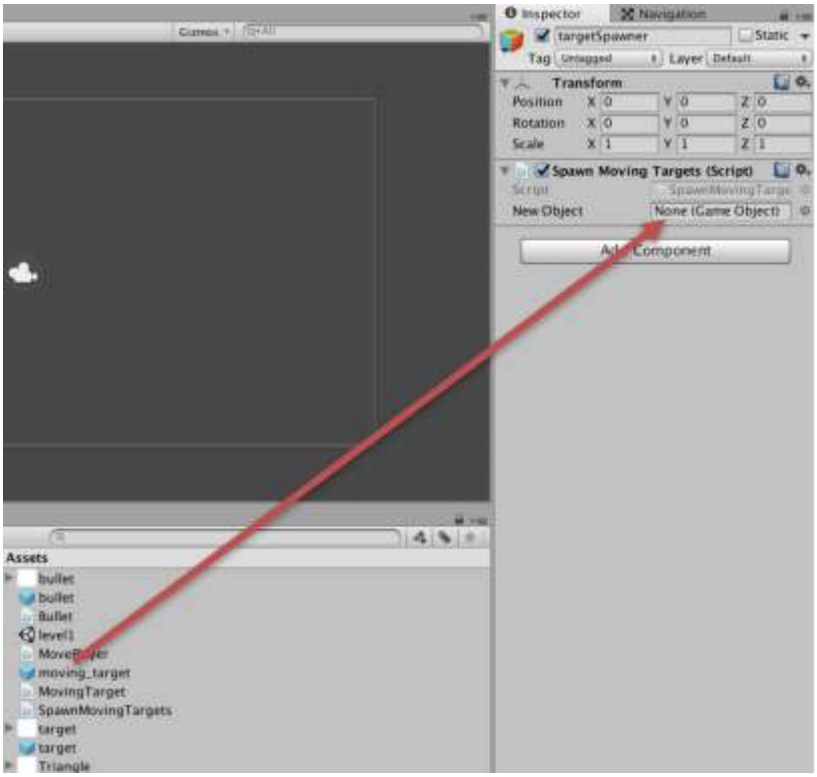


Figure 6-21: Setting the prefab to be spawn (part 1)

The component SawnMovingTarget should then look as follows.

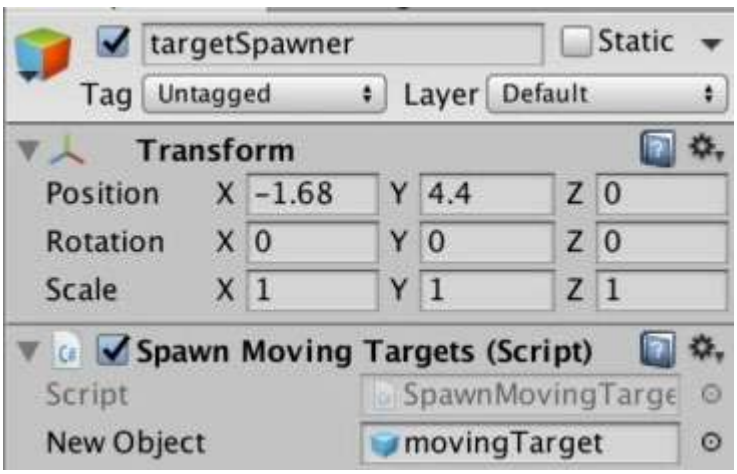


Figure 6-22: Setting the prefab to be spawn (part 2)

Last, using the Scene view, we just need to move the object called targetSpawner at the upper boundary of the screen; this is so that the moving targets are instantiated at the very top of the screen, just above the player.

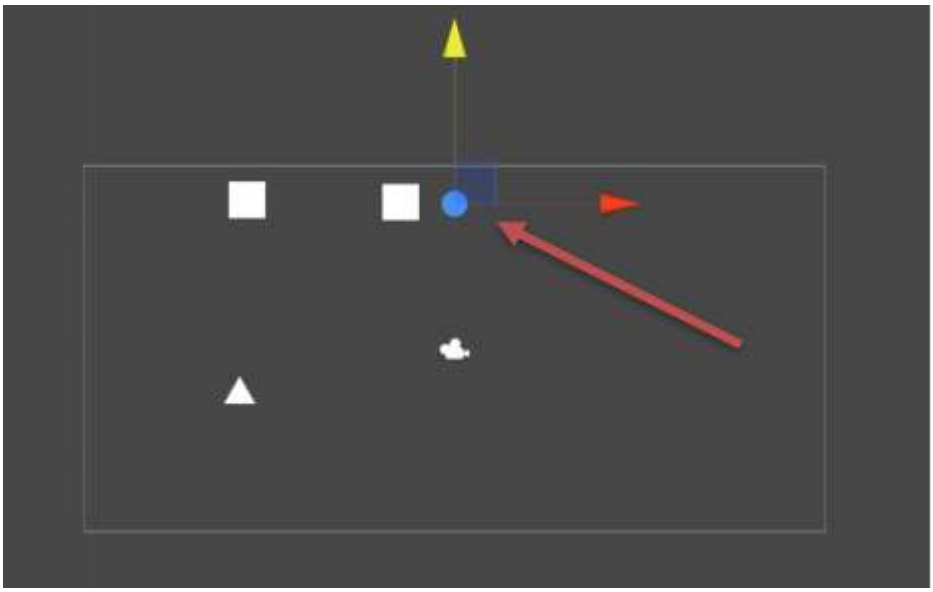


Figure 6-23: Moving the targetSpawner object

Once this is done, you can delete or deactivate the objects called moving_target and target that are already in the scene (i.e., the two squares that you could see in the previous figure), and test the scene. To deactivate these objects, you can select them and, using the Inspector window, uncheck the box to the left of their name.

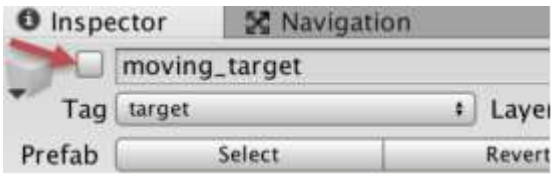


Figure 6-24: Deactivating the moving target

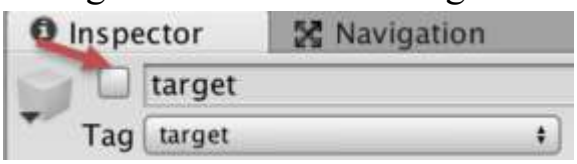


Figure 6-25: Deactivating the target

As you play the scene, you should see that a new moving target is instantiated every second at random, as described on the next figure.

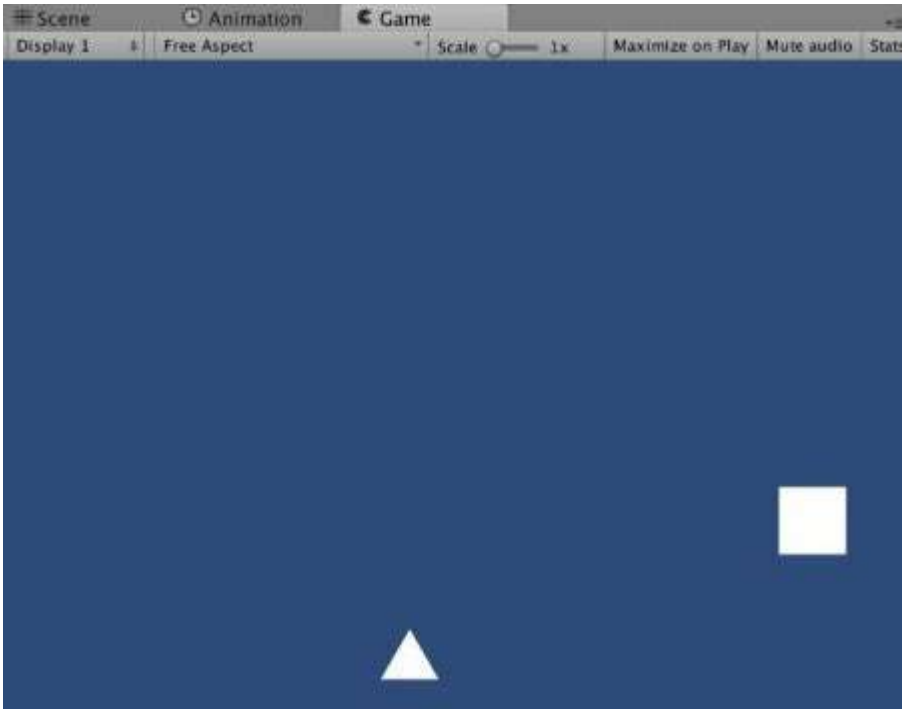


Figure 6-26: Spawning moving targets

Managing Damage

Now that we have created a moving target that the player can shoot, we will create a script that manages the damage taken by the target so that it is destroyed only after being hit several times by the player's bullets.

Please create a new script called ManageTargetHealth (i.e., select Create | C# Script from the Project window)
Add the following code at the beginning of the class (new code in bold).

```
public class ManageTargetHealth : MonoBehaviour {
public int health, type;
public static int TARGET_BOULDER = 0;
```

In the previous code, we create three variables: and
health will be used to determine the health (or strength) of each target so that we know how much damage it can sustain before being destroyed.

type is used to set different types of targets; each of these will have different levels of health (or strength).
TARGET_BOULDER will be used as a type for our moving targets (i.e., boulders). Note that this variable is both static and this means that it can be accessed from outside its class; also, because it is this variable can be accessed without the need to instantiate a new object of type

We will come back to this principle later, but in a nutshell, static variables and functions can be used by other classes with no instantiation required; you can consider these static variables and functions as utility classes and variables that can be used without the need to be part of a particular a bit like a friend granting your access to his or her car without the need for you to be the owner. For example, the function Debug.Log can be used from anywhere in your game, although, you don't need to instantiate an object of type Debug for this purpose; the same holds true for the function again, you can use this function to find a particular object; however, you don't need to instantiate an object of class GameOBject to be able to use this function

Now, we just need to specify the health (or strength) of the target, based on its type, in the Start function.

Please add the following code to the Start function (new code in bold).

```
void Start ()
{
if (type == TARGET_BOULDER) health = 20;
}
```

Add a new function called at the end of the class (i.e., before the last closing curly bracket) as follows:

```
public void GotHit(int damage)
{
health-= damage;
if (health <= 0) DestroyTarget ();
}
```

In the previous code:

We declare a function called its return type is void because it does not return any value; it takes a parameter of type int that will be referred to as damage within this function.
We then set the value of the variable health by subtracting the value of the variable damage from the previous value of the variable this is equivalent to the following code:

```
health = health – damage;
```

If the health is 0 or less, we then call the function called

We now just need to create the function called

Please add a new function called destroyTarget at the end of the class (i.e., before the last closing curly bracket) as follows:

```
public void DestroyTarget()
{
Destroy (gameObject);
}
```

In the previous code:

We create a new function called destroyTarget of type void (since it does not return any value).
This function destroys the object linked to this script (i.e., the target).

Once this is done, we can save and use this script:

Please save your code and check that it is error-free.
Using the Project view, drag and drop this script (i.e., ManageTargetHealth) on both the target and the moving_target prefabs.

Next, we just need to modify the script SpawnMovingTarget so that we specify the type of the target that is to be created.

Please modify the spawning script (i.e., as follows (new code in bold).

```
if (timer >= 1)
{
GameObject t = (GameObject)(Instantiate (newObject, newPosition, Quaternion.identity));

t.GetComponent ().type = ManageTargetHealth.TARGET_BOULDER;
timer = 0;
}
```

In the previous code: we specify that the value of the variable called for the script called that is a component of the object t is

Note that we have accessed the static variable TARGET_BOULDER from the class ManageTargetHealth without instantiating an object of type this is because the variable TARGET_BOULDER is static.

Last but not least, we can add the following code to the script

```
if (coll.gameObject.tag == "target")
{
//Destroy (coll.gameObject);
coll.gameObject.GetComponent().GotHit(10);
Destroy (gameObject);
}
```

You can now play the scene and test that the moving targets disappear after being hit twice.

For testing purposes, you can also drag and drop the script ManageTargetHealth on the prefab called reactivate the object target in the Scene view, and then fire bullets at this target. It should disappear after two bullets have been fired.

Level Roundup

In this chapter, we have learned how to create a simple level with a spaceship, for the player, that can fire missiles and destroy static or moving targets. We also managed to create moving targets spawn at regular intervals but at random locations. Finally, we also learned to create Rigidbody2D and BoxCollider2D components and detect collision between the player's bullets and the targets. So, we have covered considerable ground to get you started with the first level of your 2D shooter.

Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

---

Quiz

Now, let's check your knowledge! Please answer the following questions (the answers are included in the resource pack) or specify if these statements are either correct or incorrect.

The method Random.GenerateRandomNumber is used to generate random numbers.
Sprites can be created using the menu Create |
The function Input.GetKeyDown is called to detect when a key has been pressed and subsequently released.
The function Input.GetKey is called whenever a key is being pressed.
Static variables cannot be accessed outside their class.
The following code will add force to the Rigidbody2D component of the object linked to the script.

```
GetComponent ().AddForce (transform.up * 1000);
```

The following code will destroy 10 instances of the current object:

```
Destroy (gameObject, 10);
```

When a collision between two objects (each with a 2DCollider) occurs, the function OnCollisionEnter2D is called.
A function of type void does not return any value.
Only square sprites can be created in Unity.

Challenge 1

Now that you have managed to complete this chapter and that you have created your first level, you could improve the level by doing the following:

Modify the color of each target.

Modify the speed (or frequency) at which the moving targets are created.

Challenge 2

Now that you have managed to complete this chapter and that you have created your first level, you could improve it by doing the following:

In the script create different types of targets; for example:

```
public static int TARGET_BOULDER_2 = 1;
```

In the same script, modify the Start function so that the health of this particular target is set accordingly (i.e., different strength for different

targets). For example:

If (type ==) health =;

Modify the script so that the boulder created is created at random; for example, you could generate a number between 1 and 2; based on this number, you will generate a boulder of type 0, or a boulder of type 1.

7

Using Lists and Dictionnaries in C#

In this section, we will go through an introduction to C# programming and look at key aspects that you will need for your games, including:

C# Syntax.
Variable types and scope.
Useful coding structures (e.g., loops or conditional statements).

So, after completing this chapter, you will be able to:

Understand key concepts related to C# programming.
Understand the concepts of variables and methods.
Create a Flappy Bird game based on the concepts covered in this chapter.

The code solutions for this chapter are included in the resource pack that you can download by following the instructions included in the section entitled and Resources for this

Lists

As we have seen in the previous sections, it is sometimes useful to employ arrays. However, when you are dealing with a large amount of data, or data that is meant to grow overtime, lists may be more useful, as they include built-in tools to sort and organize your data.

In addition, lists are generally more efficient as your data grows. So, you don’t always need to use lists; however, they may be more efficient to organize your data, especially for large and evolving data sets.

So let’s look into

You can declare a list as follows:

List myList;

The declaration follows the syntax:

List nameOfVariable;

So you could create a list of integers, strings, or Cubes if you wished.

Once the list has been created, C# offers several built-in methods that make it possible to manipulate a list, including:

adds an item at the end of the list.
inserts an item at a specific index.
removes an item from the list.
removes an item at a specific index
returns the number of items in a list.
sorts the elements of a list.

Each element in the list has a default index that is relative to when it was first added to the list; the earlier the item is added to the list and the lower the index. So the first item added to the list will have, by default, the index 0, the second item will have the index 1, and so on. Let’s look at an example:

```
List listOfNames = new List ();
listOfNames.Add ("Mary");
listOfNames.Add ("Paul");
print ("Size of List" + listOfNames.Count);
//this will display “Size of list 2”
listOfNames.Remove("Paul");

print ("Size of List after removing" + listOfNames.Count);
//this will display “Size of list 1”
```

In the previous code:

We create a new list of string variables.
We then add two elements to the list: the strings Mary and
We display the size of the list before and after an item has been removed from the list.

Dictionaries

Lists are very useful, and dictionaries, which are special type of lists, take this concept a step further. With dictionaries, you can define a dataset with different records, and each record is accessible through a key instead of an index; for example, let's consider a class of students, each with a first name, a last name, and a student number. To represent and manage this data, we could create code similar to the following:

```
public class Student{
public string firstName;
public string lastName;
public Student(string fName, string lName)
{
firstName = fName;
lastName = lName;
}
}
```

We could then create code that uses this class as follows:

```
DictionaryStudent> students = new DictionaryStudent>();
students.Add("ST123",new Student("Mary", "Black"));
students.Add("ST124",new Student("John", "Hennessy"));
print ("Name of student ST124 is " + students ["ST124"].firstName);
```

In the previous code:

We declare a dictionary of
When declaring the dictionary: the first parameter, which is a is used as an index or a this index will be the student
The second parameter will be an object of type
So effectively we create a link between the key and the Student object.
We then add students to our dictionary.
When using the Add method, the first parameter is the key (or the student id in our case: ST123 or ST124 here), and the second parameter is the student object. This student object is created by calling the constructor of the class Student and by passing relevant parameters to the constructor, such as the student's first name and last name.
Finally, we print the first name of a specific student based on its student

As for lists, Dictionaries have several built-in functions that make it easier to manipulate them, including:

to add a new item to the dictionary.
to check if a record with a specific key exists in the dictionary.
to remove an item from the dictionary.

Events

Put simply, events can be compared to something that happens at a particular time. When this event occurs, something (an action for example) needs to be performed. To simplify things, we could draw an analogy between events and daily activities: when your alarm goes off in the morning (which is an event) you can either get-up (this is an action) or decide to go back to sleep. Similarly, when you receive an email (this is another event), you can decide to read it (this is another action), and then reply to the sender (another action).

In computer terms, the concept of events is relatively similar, although the events that we will be dealing with in C# will be slightly different to the ones we just mentioned. For example, we could be waiting for the user to press a key (an event) and then move the character accordingly (an action), or wait until the user clicks on a button on screen (an event) to load a new scene (which is an action).

Usually in Unity, whenever an event occurs, a function is called. The function, in this case, is often referred as a handler, because it "handles" the event that just occurred. You have then the opportunity to modify this function and to add instructions (which include statements) that should be executed when this event occurs.

To draw an analogy with daily activities: we could write instructions to a friend on a piece of paper, so that, in case someone calls in our absence, this friend knows exactly what to do. So an event handler is basically a set of instructions, usually stored within a function, that need to be followed in case a particular event occurs.

Sometimes information is passed to this method about the particular event that just occurred, and sometimes not. For example, in Unity, when the screen is refreshed the method Update is called. When a particular script is enabled, the method Start is called. When there is a collision between the player and an object, the method OnControllerColliderHit is called. For this particular event (which is a collision), an object is usually

passed to the method that handles the event so that we get to know more about the other object involved in the collision.

As you can see, many events can occur in our game, and in your games, you will more than likely be dealing with the following events:

when a script is enabled (e.g., start of the scene).
when the screen is refreshed (e.g., every frame).
when a collision occurs between the player and another object.
when the game starts (i.e., once throughout the lifetime of your game).

Level Roundup

In this chapter, we have learned some concepts related to C# and Object-Oriented Programming. We also learnt how to define and to use classes, along with member variables and methods. Along the way, we also looked at other programming concepts such as variables, loops, and conditional statements. So, we have covered considerable ground to get you started with C#!

Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist
Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist
Checklist Checklist Checklist Checklist Checklist Checklist Checklist
Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Quiz

Now, let’s check your knowledge! Please answer the following questions or specify if these statements are either correct or incorrect (the solutions are at the end of the book).

The value of a variable always remains constant.
A method always returns information.
A method may not return information.
If a method is void, it will return an integer value.
An array can store several variables at a time.
A class usually includes a constructor.
A for loop can be used to go through all the elements of an array.
A public method is accessible from anywhere.
A private variable is accessible only from members of the class.
A protected variable is accessible only from members of the class.

8 Creating a Word Guessing Game

In this section we will create a 2D word guessing game with the following features:

A word will be picked at random from an existing list.
The letters of the word will be hidden.
The players will try to guess each letter by pressing a letter on their keyboard.
Once a letter has been discovered, it will then be displayed onscreen.
The player has a limited number of attempts to guess the word.

So, after completing this chapter, you will be able to:

Read words from a text file.
Pick a random word.
Process and assess the letters pressed by the player.
Display the letters that were correctly guessed by the player.
Track and display the score.
Check when the player has used too many guesses.

Figure 8-1: The final game

Creating the interface for the game

So, in this section, we will start to create the core of the word guessing game; it will consist of text fields initially blank, and located in the middle of the screen.

Please create a new scene and save it as select File | Save Scene

Once this is done, you can check that the 2D mode is activated, based on the 2D logo located in the top right-corner of the Scene view, as illustrated in the next figure.

Figure 8-2: Activating the 2D mode

First, we will remove the background image for our If you look at your Game view, it may look like the following figure.

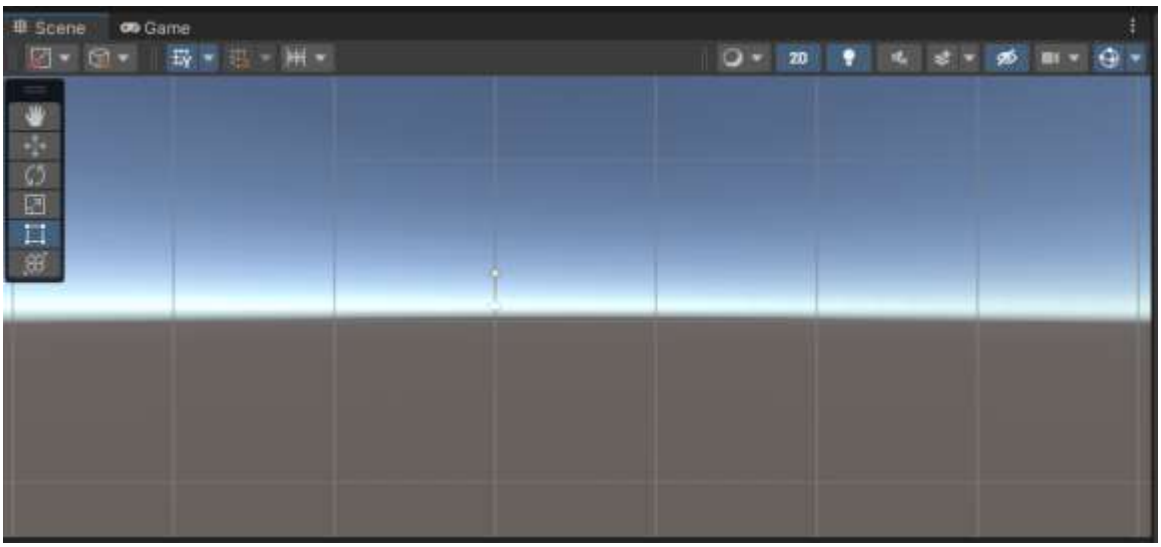


Figure 8-3: The initial background

If it is the case, then please do the following:

From the top menu, select: Window | Rendering |
Select the tab labelled
Then delete the Default Skybox that is set for the attribute called SkyBox (i.e., click on the attribute to the right of the label Skybox and press DELETE on your keyboard).
You can also change the attribute Source for the section Environment Lighting to

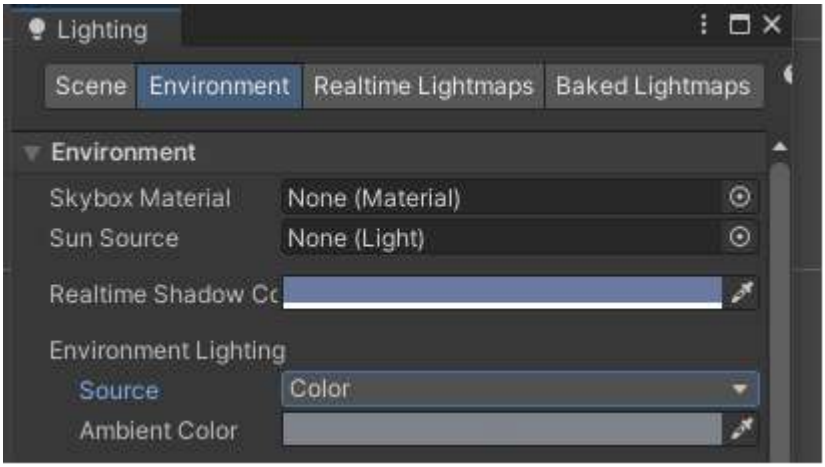


Figure 8-4: Lighting properties

Once this is done, your Game view should look like the following.

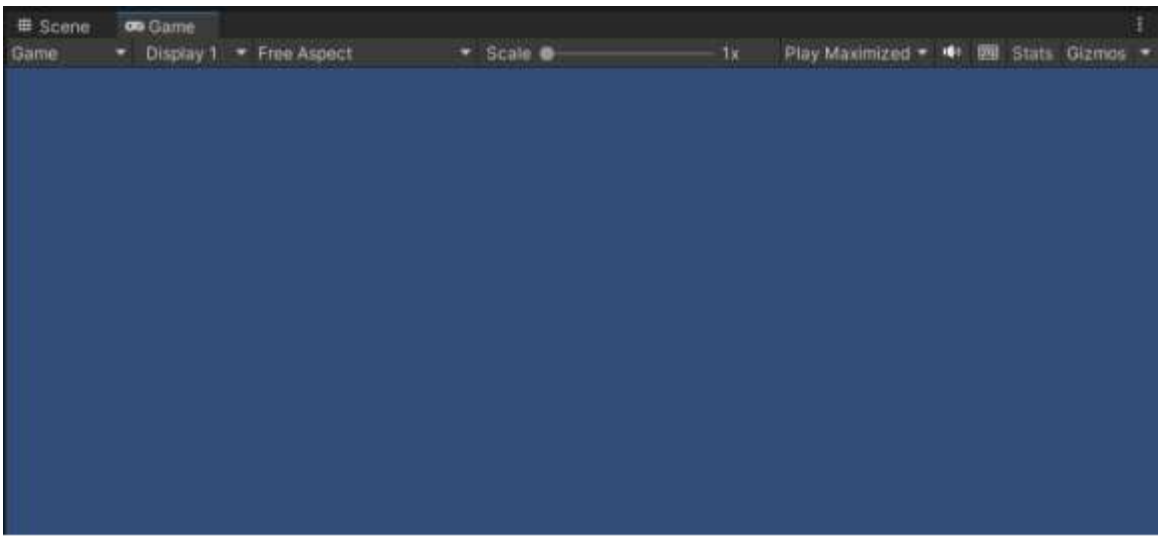


Figure 8-5: The Game view after deleting the SkyBox

We will now create a text field that will be used for the letters to be guessed.

From the top menu, please select GameObject | UI | Text - This will create a UI Text object called Text along with a Canvas object.
Please rename this text object

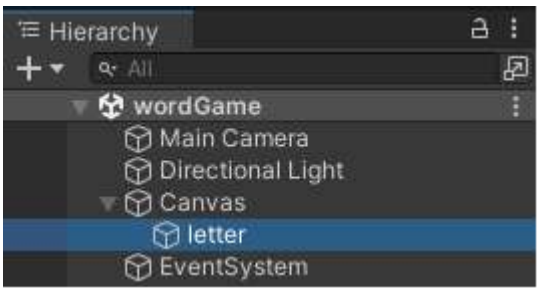


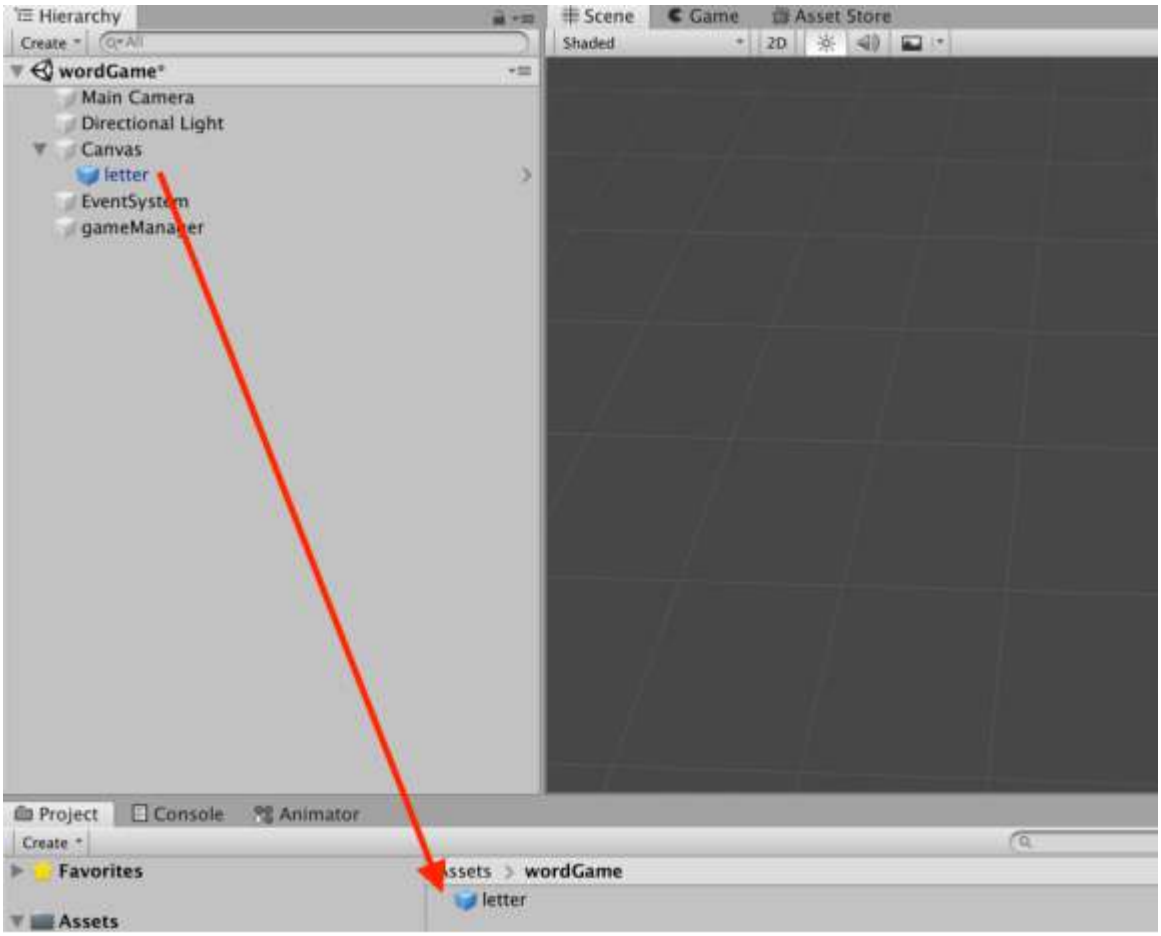
Figure 8-6: Creating a new letter

Select this object (i.e., letter) in the and, using the Inspector window, please set its attributes as follows:

For the component Rect Position = Width = 100 and Height =
For the component TextMeshPro – Text Font-size = Color = please empty the
For the component TextMeshPro – Text vertical alignment = horizontal alignment =

Once this is done, we will create a prefab from this object, so that we can instantiate it later on (i.e., create objects based on this prefab).

Please drag and drop the object letter from the Hierarchy window to the Project window.



This will create a new prefab called

Next, we will create a gameManager object; this object will be in charge of setting the layout for the game and processing the user's entries; in other words, it will be responsible for running and managing the game.

Please create a new empty object | Create
Rename this new object

Next, we will create a script that will be attached to the gameManager object; this script will be in charge of running the game (e.g., displaying letters, processing user inputs, etc.).

From the Project window, select Create | C#

Rename the new script
You can then open this script.

In this script, we will display several letters in the middle of the screen.

Please add this code at the beginning of the class.

public GameObject letter;

In the previous code we create a new public variable called it will be accessible from the Inspector since it is public, and it will be set (or initialized) with the letter prefab that we have created earlier. This variable will be used to generate new letters based on that template (i.e., the prefab).

Please check that the code is error-free in the Console window.

Drag and drop the script GameManager from the Project window on the object called gameManager located in the

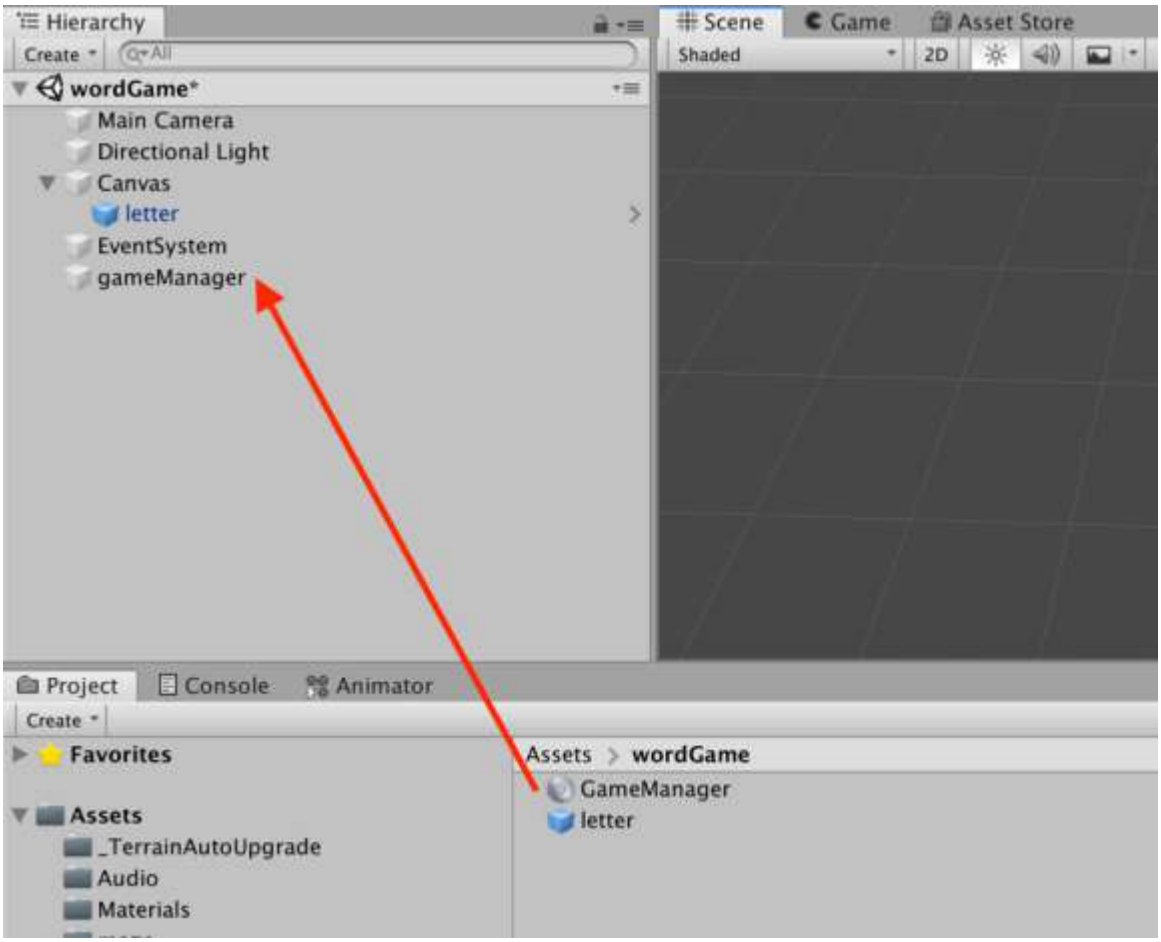


Figure 8-7: Adding a script to the game manager

You can now delete the object letter in the Hierarchy view.
Once this is done, you can select the object called in the
Using the Inspector window, you will see that this object now includes a new component called GameManager with an empty field called

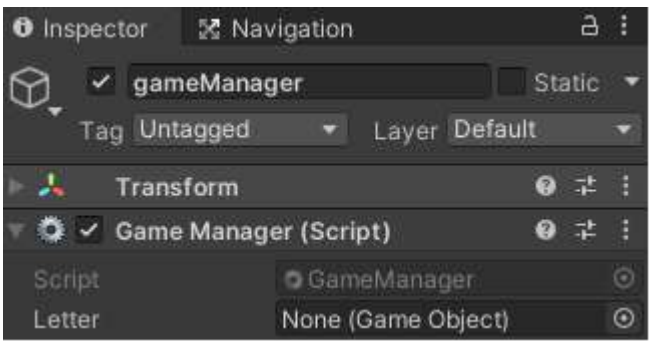


Figure 8-8: A new component added to the game manager

Please drag and drop the prefab called letter from the Project window to this empty field in the Inspector window, as described in the next figure.

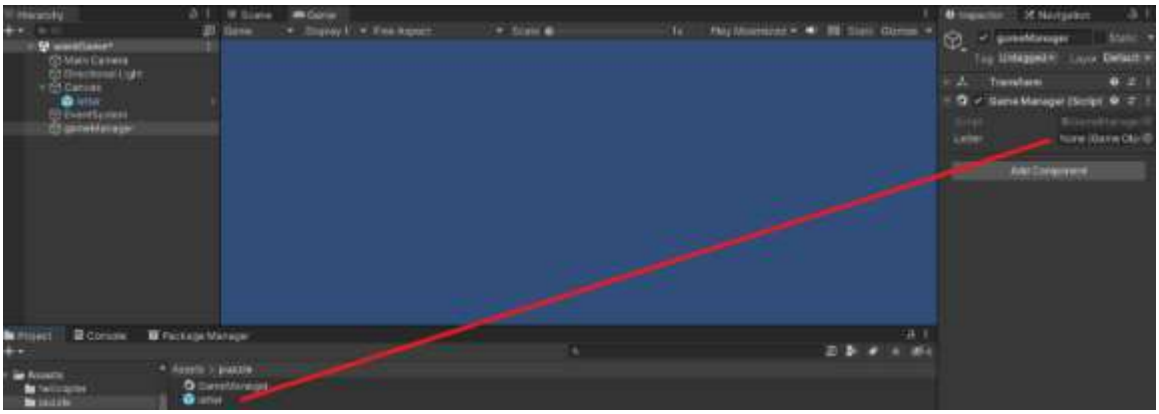


Figure 8-9: Initializing the letter variable with a prefab

Once this is done, we can start to write a function that will create the new letters.

Please open the script

Add the following code at the end of the class (i.e., just after the function

```
void InitLetters()
{

int nbLetters = 5;
for (int i = 0; i < nbLetters; i++) {
Vector3 newPosition;
newPosition = new Vector3 (transform.position.x + (i * 100),
transform.position.y, transform.position.z);
GameObject l = (GameObject)Instantiate (letter, newPosition,
Quaternion.identity);
l.name = ”letter” + (i + 1);
l.transform.SetParent(GameObject.Find (“Canvas”).transform);
}
}
```

In the previous code:

We define that we will display five letters using the variable this number is arbitrary, for the time being, so that we can ensure that we can display letters onscreen; this number of letters will, of course, vary later on, based on the length of the word to be guessed.
We then create a loop that will loop five times (once for each letter).
In each iteration, we define the position of the new letter using the variable

This position of the letter is calculated by combining the position of the object gameManager that is linked to this script (i.e., plus the size of the letter (i.e., this size/width was set-up earlier-on with the Inspector using the width attribute) multiplied by the variable so the position of the first letter on the x-axis will be transform.position + the second one will be at the x position transform.position + and so on.
We instantiate a letter and also set its name.
Finally, we set the parent of this new object to be the object called this is because, as a UI Text object, this object needs to be associated to a canvas in order to be displayed onscreen; this is usually done by default as you create a new UI Text object with the editor in Unity; however, this needs to be done manually here, as this object is created and added to the Scene from a script.

Finally, please add the following code to the Start function.

InitLetters();

As you play the you should see that new letters have been created in the

Figure 8-10: The newly-created letters

If you double-click on one of them (e.g., in the you should see where they are located and their layout in the Scene view.

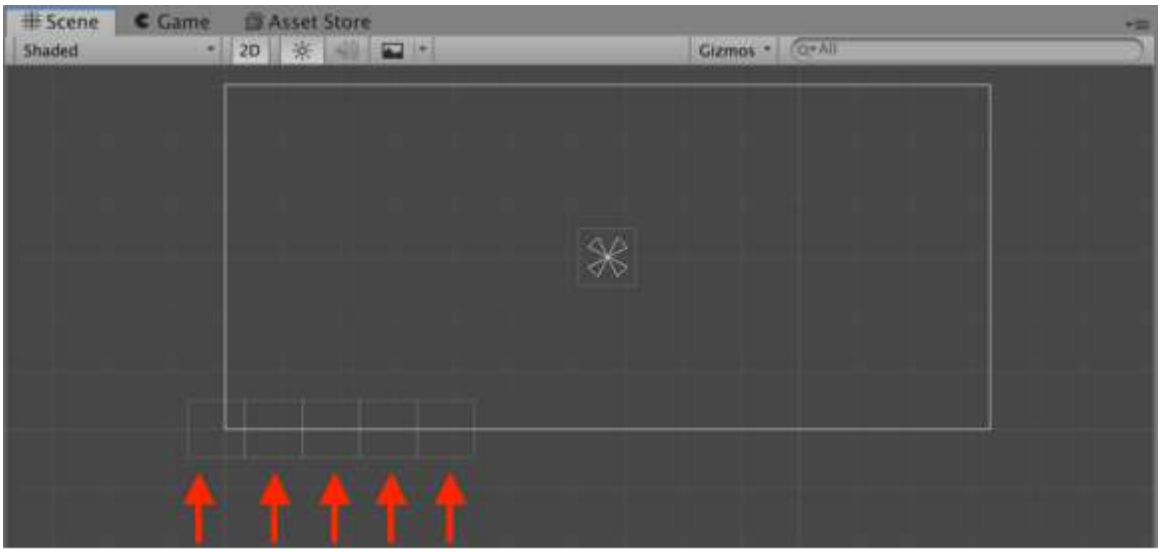


Figure 8-11: The layout of the letters

Now, because the position of the letters is based on the position of the game manager, you may notice, as for the previous figure, that the letters are not centered properly. So we need to ensure that these letters are properly aligned vertically and horizontally. For this purpose, we will do the following:

Create an empty text object located in the middle of the screen.
Base the position of each letter on this object.
Ensure that all letters are now properly aligned.

So let’s proceed:

Please create a new UI Text object | UI | and rename it
Select this object in the
Using the in the component called change its position to PosX=0 and you can leave the other attributes as they are.

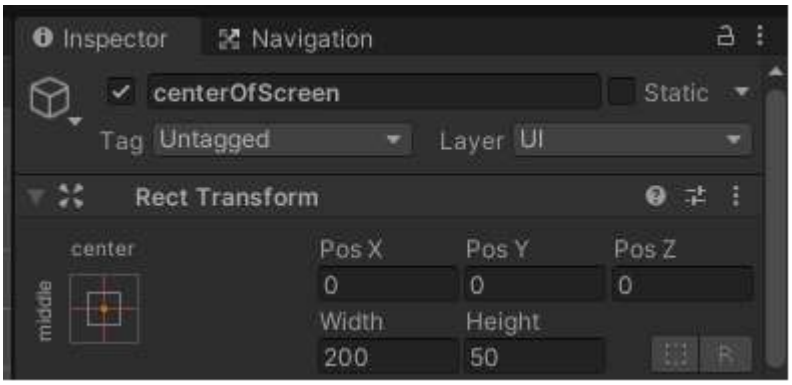
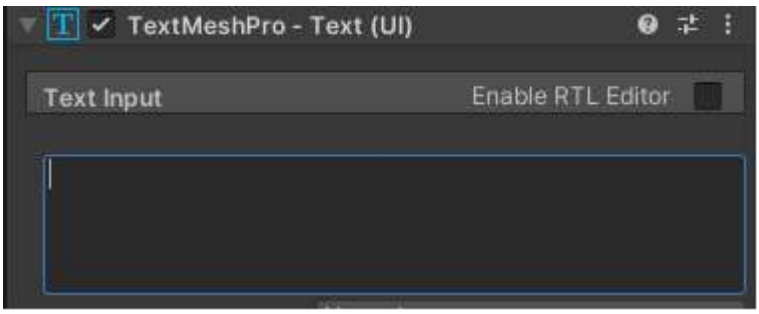


Figure 8-12: Changing the position attributes

Because this object is a UI object, setting its position to (0,0) will guarantee that it will be displayed in the center of the screen; this is because the coordinates of the UI object (for the component are based on the view/camera. So PosX=0 and in this case, corresponds to the center of the screen; using an empty object would have been different as the coordinates would be world coordinates and not related to the screen/view.

Using the in the component called Text, delete the default text, so that this UI Text is effectively an empty field.



Once this is done, we can modify the code in the script GameManager to center our letters.

Please open the script
Add the following code to the start of the class (new code in bold).

```
public GameObject letter;
public GameObject cen;
void Start () {
cen = GameObject.Find (“centerOfScreen”);
InitLetters ();
}
```

In the previous code, we declare a new variable called cen that will be used to refer to the object
In the function initLetters that we have created earlier-on, please modify this line

```
newPosition = new Vector3 (transform.position.x + (i * 100), transform.position.y, transform.position.z);
```

... with this code...

```
newPosition = new Vector3 (cen.transform.position.x + (i * 100), cen.transform.position.y, cen.transform.position.z);
```

In the previous code, we now base the position of our letters on the center of the screen.

Please save your script, and check that it is error-free.
Play the and you should see, in the Scene view, that the letters are now aligned vertically; however, they are slightly offset horizontally, as illustrated in the next figure.

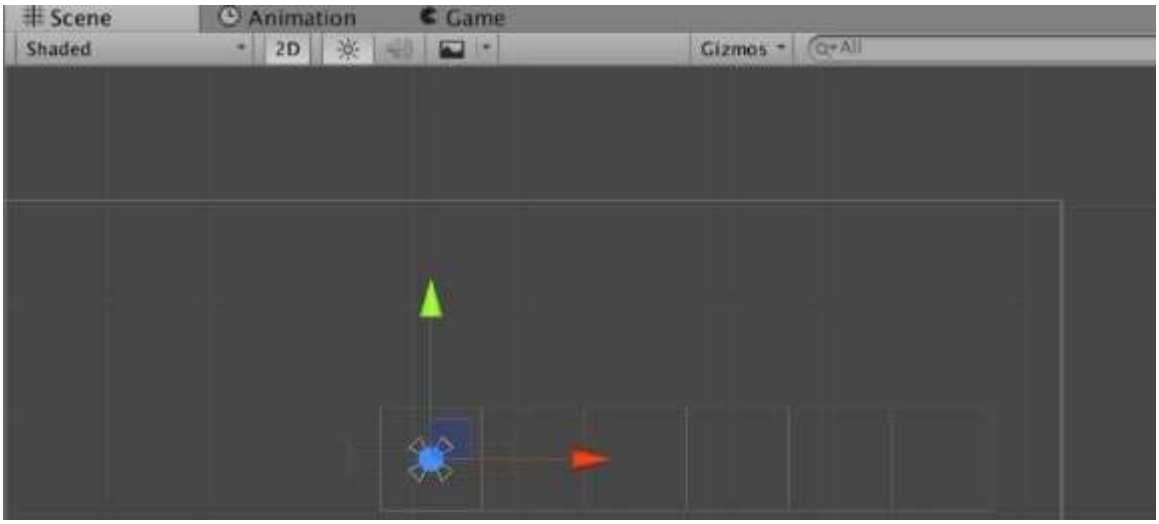


Figure 8-13: Aligning the letters

So there is a last change that we can include in our script, so that each letter is centered around the center of the screen; this will consist in offsetting the position of each letter based on the center of the screen as follows, so that the middle of the word matches with the center of the screen.

Please open the script GameManager and, in the function called replace this line:

newPosition = new Vector3 (cen.transform.position.x + (i * 100), cen.transform.position.y, cen.transform.position.z);

with this line...

newPosition = new Vector3 (cen.transform.position.x + ((i-nbLetters/2.0f) *100), cen.transform.position.y, cen.transform.position.z);

In the previous code, we offset the position of each letter based on the center of the screen; so the x-coordinate of the first letter will be the x-coordinate of the second letter will be and so on.

You can save your script, play the and look at the Scene view; you should see that the letters are now properly aligned, as illustrated on the

next figure.

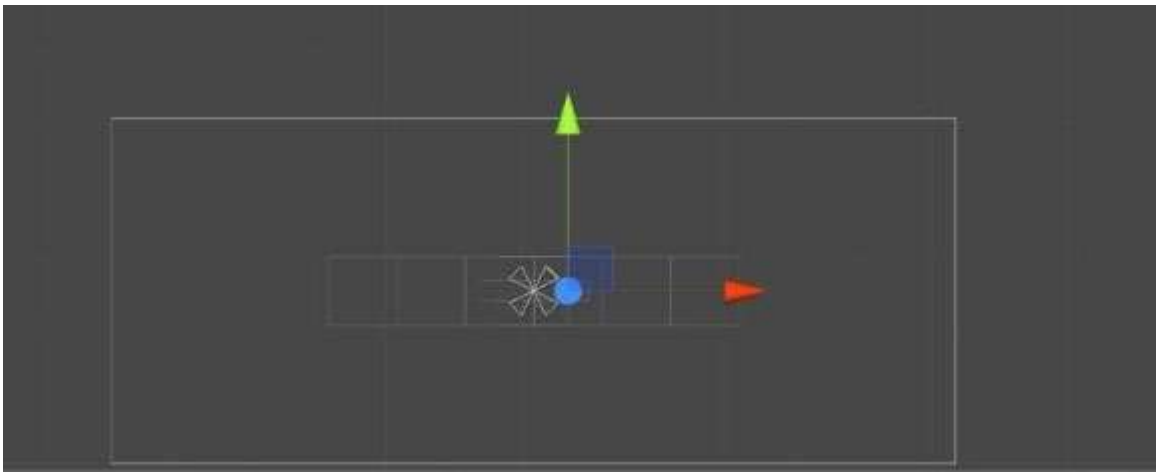


Figure 8-14: The letters are now properly aligned

If you would like to see what the letters would look like, you can, while the game is playing, select each newly-created letter in the Hierarchy and modify its Text attribute in the Inspector (using the component as illustrated in the next figure.

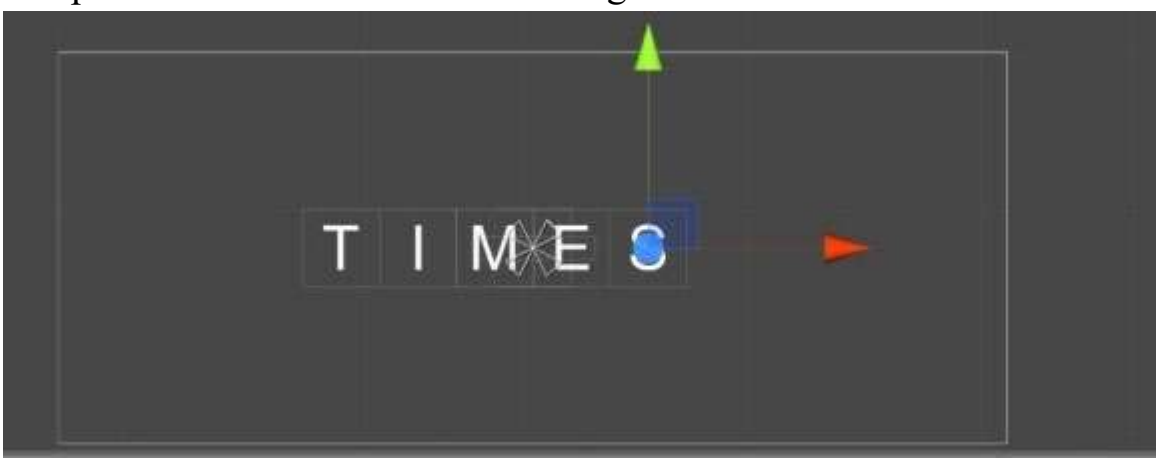


Figure 8-15: Changing the text of each letter at run-time

Detecting and processing the user input

Perfect. So, at this stage, we have a basic interface for our game, and we can display letters onscreen. So, in this section, we will implement the main features of the game, that is:

Create a new word to be guessed.

Count the number of letters in this word.

Display corresponding empty text fields.
Wait for the user to press a key (i.e., a letter) on the keyboard.

Detect the key pressed by the user.
Display the corresponding letters in the word to be guessed onscreen.

So let's start.

Please open the script called
Add the following code at the beginning of the class.

using TMPro;

In the previous code, we include the namespace TMP so that we can use TextMeshPro classes in our code.

Add the following code at the start of the class (new code in bold).

```
public GameObject letter;
public GameObject cen;
private string wordToGuess = ““;
private int lengthOfWordToGuess;
char [] lettersToGuess;
bool [] lettersGuessed;
```

In the previous code:

We declare four new variables.
wordToGuess will be used to store the word to be guessed.
lengthOfWordToBeGuessed will store the number of letters in this word.

lettersToGuess is an array of char (i.e., characters) including every single letter from the word to be discovered by the player.

letterGuessed is an array of Boolean variables used to determine which of the letters in the word to guess were actually guessed correctly by the player.

Next, we will create a function that will be used to initialize the game.

Please add the following function to the class.

```
void InitGame()
{
wordToGuess = “Elephant”;
lengthOfWordToGuess = wordToGuess.Length;
wordToGuess = wordToGuess.ToUpper ();
lettersToGuess = new char[lengthOfWordToGuess];
lettersGuessed = new bool [lengthOfWordToGuess];
lettersToGuess = wordToGuess.ToCharArray ();
}
```

In the previous code:

We declare a function called
In this function, we initialize the variable this will be the word Elephant for the time being.
We then capitalize all the letters in this word; as we will see later in this chapter, this will make it easier to match the letter typed by the user (which usually is upper-case) and the letters in the word to guess.
We then initialize the array called lettersToGuess and

Note that for Boolean variables, the default value, if they have not been initialized, is As a result, all variables in the array called lettersGuessed will initially be set to false (by default).

Finally, we initialize the array called lettersToGuess so that each character within corresponds to the letters in the word to guess; for this, we convert the word to guess to an array of characters, which is then saved into the array called

Once this function has been created, we will need to process the user's input; for this purpose, we will create a function that will do the following:

Detect the letter that was pressed by the player on the keyboard.
Check if this letter is part of the word to guess.
In this case, check if this letter has not already been guessed by the player.
In this case, display the corresponding letter onscreen.

Let's write the corresponding code.

Please add the following function to the script

```
void CheckKeyboard()
{
if (Input.GetKeyDown(KeyCode.A))
{
for (int i=0; i < lengthOfWordToGuess; i++)
{
```

```
if (!lettersGuessed [i])
{
if (lettersToGuess [i] == 'A')
{
}
}
}
}
}
```

And then add this code (new code in bold)

```
if (lettersToGuess [i] == 'A')
{
lettersGuessed [i] = true;
GameObject.Find(“letter”+(i+1)).GetComponent().text = “A”;
}
```

In the previous

We declare the function called
We then create a loop that goes through all the letters of the word to be guessed; this is done from the first letter (i.e., at the index 0) to the last one.
We check if this letter has already been guessed.
If it is not the case, we check whether this letter is

If this is the case, we then indicate that this letter (i.e., the letter was found.

We then display the corresponding letter onscreen.

Last but not least, we just need to be able to call these two functions to initialize the game and to also process the user's inputs.

Please add the following code to the Start function (new code in bold).

```
void Start () {
cen = GameObject.Find ("centerOfScreen");
InitGame ();
InitLetters ();
}
```

Please make sure that the function initGame is called before InitLetters (as illustrated in the previous code) in the Start function; this is because, as we will see later, the function InitLetters will use some of the information that has been set in the function initGame (i.e., the number of letters). So in order for our game to work correctly, the function initGame should be called before the function

Please add the following code to the Update function.

```
void Update () {
CheckKeyboard ();
}
```

The last change we need to add now is linked to the number of letters to be displayed; as it is, the number is set to 5 by default; however, we

need to change this in the function called so that the number of UI Text objects that corresponds to the letters in the word to be guessed reflects the length of the word that we have just created.

Please modify the function InitLetters as follows (new code in bold).

```
void InitLetters()
{
int nbLetters = lengthOfWordToGuess;
```

So at this stage, we have all the necessary functions to start our game; so you can save the script, check that it is error free and play the As you play the if you press the A key on the keyboard, the letter A should also be displayed onscreen, as it is part of the word

Figure 8-16: Detecting the key pressed

So, this is working properly, and we could easily add more code to detect the other keys; this would involve using the code included in the function and copying/pasting it 25 times to be able to detect the other 25 keys/letters, using the syntax once for each key. So the code could look as follows:

```
if (Input.GetKeyDown(KeyCode.A))

{
```

```
...
}
if (Input.GetKeyDown(KeyCode.B))
{
...
}
```

```
if
{
...
}
```

Now, this would be working perfectly, however, this would also involve a lot of repetitions (copying 24 times the same code); so to make the code more efficient, we will use a slightly different way of detecting the key pressed by the player. This method will involve the following:

Check if a key was pressed.
Check if this key is a letter.
Proceed as previously to check whether this letter is part of the word to be guessed.

Please create a new function called as follows:

```
void checkKeyboard2()
{
if (Input.anyKeyDown && !Input.GetMouseButtonDown(0))
{

char letterPressed = Input.inputString.ToCharArray () [0];
int letterPressedAsInt = System.Convert.ToInt32 (letterPressed);
```

```
    if (letterPressedAsInt >= 97 && letterPressed <= 122)
    {
    for (int i=0; i < lengthOfWordToGuess; i++)
    {
    if (!lettersGuessed [i])
    {
    letterPressed = System.Char.ToUpper (letterPressed);
    if (lettersToGuess [i] == letterPressed)
    {
    lettersGuessed [i] = true;
    GameObject.Find("letter"+(i+1)).GetComponent().text =
letterPressed.ToString();
    }
    }
    }
    }
    }
    }
```

In the previous code:

We detect whether a key has been pressed using the keyword We also check that the mouse was not pressed using the keyword

If this is the case, we save the key (i.e., the letter) that was pressed into the variable called For this, given that any key pressed on the keyboard is stored as a string, we need to convert this string value to a character; the character recorded in the variable letterPressed will effectively be the first character of the string that corresponds to the key pressed by the player. Once this is done, we convert the letter pressed (i.e., character) to an integer value.

We then check, using the integer value associated with the key pressed (or what is called the ASCII code), that the key is a letter; this corresponds to

an integer value between 97 and
Once this final check is complete, we do the exact same as we have done earlier in the function checkBoard that we have created previously (i.e., this is the same code).

The last thing we need to do is to call the function chekBoard2 instead of the function checkBoard by amending the Update function as follows (new code in bold):

```
void Update () {
//CheckKeyboard ();
CheckKeyboard2 ();
}
```

That's it!

Once this is done, please save your code, check that it is error-free and test the As you press the keys P and you should see that they now appear onscreen.

Figure 8-17: Detecting all the keys pressed

Choosing random words

At this stage, the game works properly and the letters that the player has guessed are displayed onscreen; this being said, it would be great to

add more challenge by selecting the word to guess at random from a list of pre-defined words. So in the next section, we will learn to do just that; we will start by choosing a word from an array, and then from a text file that you will be able to update yourself without any additional coding.

So let's get started.

Please open the script
Add the following code at the beginning of the class.

```
private string [] wordsToGuess = {"car", "elephant","autocar" };
```

In the previous code, we declare an array of string variables and we add three words to it.
Add the following code to the function initGame (new code in bold).

```
//wordToGuess = "Elephant";

int randomNumber = Random.Range (0, wordsToGuess.Length);
wordToGuess = wordsToGuess [randomNumber];
```

In the previous code:

We comment the previous code.
We create a random number that will range from 0 to the length of the So in our case, because we have three elements in this array, this random number will range from 0 to 2 (i.e., from 0 to
We then set the variable called wordToGuess to one of the words included in the array called this word will be picked at random based on the variable called

You can now save you script, and check that it is error-free.

There is a last thing that we could do; because the word is chosen at random, the player will not have any idea of the length of this word; so to give an indication of the number of letters to be guessed, we could display questions marks for all the letters to be guessed, onscreen as follows:

Please select the prefab called letter from the Project window.
Using the and the component called Text change its text to as described in the next figure.

Figure 8-18: Changing the default character for letters

Lastly, we can also deactivate the object called letter that is in the

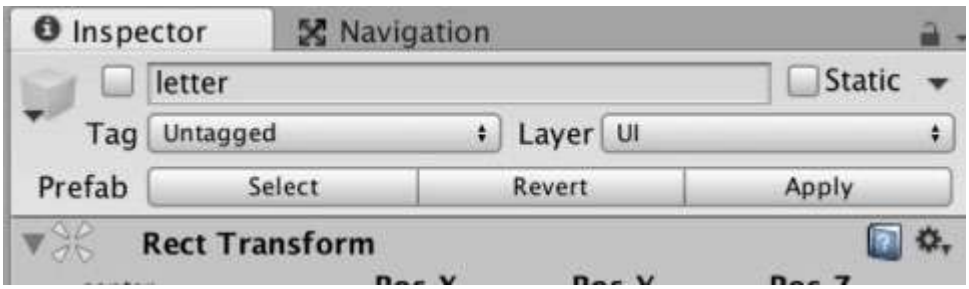


Figure 8-19: Deactivating the letter object

You can now play the and you should see questions marks where letters need to be guessed.

Figure 8-20: Displaying question marks

Tracking the score and the number of attempts

So at this stage, we have a game were we generate random words that need to be guessed by the player. So we will start to finalize our game by adding the following features:

A starting screen.
A game-over screen.
Display the number of guesses.
Set and display the maximum number of attempts.

Detect if all the letters in the word to be guessed were found.
Restart the level with a new word whenever the previous word has been guessed.
Load the game-over screen if the maximum number of attempts has been reached.

First, we will create a new splash-screen for our game; it will consist of a new Scene with a button to proceed to the game. In this splash-screen, we will also initialize the score to 0.

Create a new scene: from the Project view, select Create | and rename it

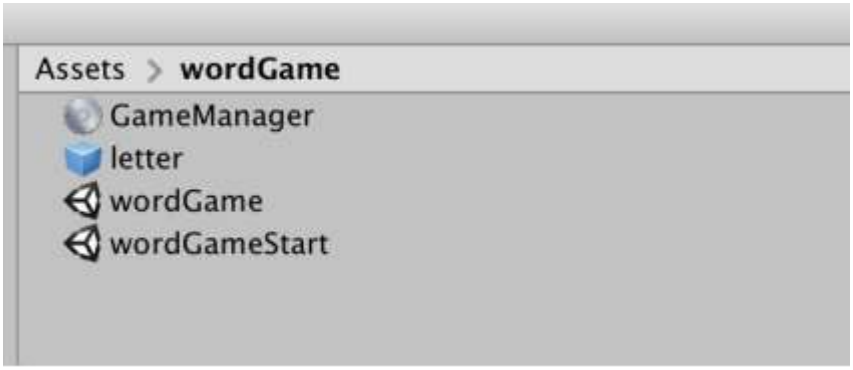


Figure 8-21: Creating a new Scene

In the Project window, double-click on this new scene.
If the skybox appears in the background, you can, as we have done in the previous sections, remove it by changing the Lighting options (i.e., Window | Rendering | Lighting

Once this is done, we can create a button that will be used to start the game:

Please, select GameObject | UI | this will create a new button along with its corresponding

Figure 8-22: Creating a new button

Select this button in the
Using the Inspector window, and the component called change its position to = 0; PosY =

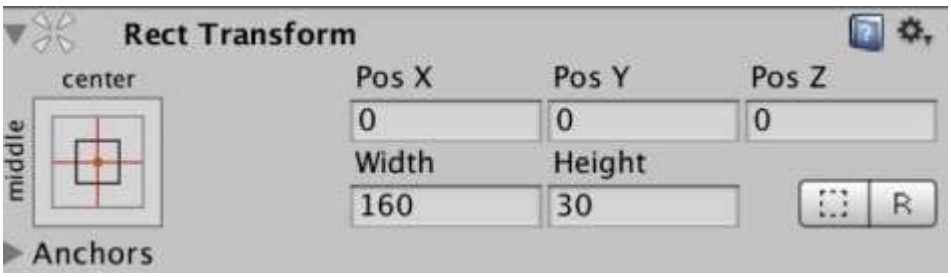


Figure 8-23: Changing the position of the button

Using the select the object called Text that is a child of the object called Button, as illustrated in the next figure.

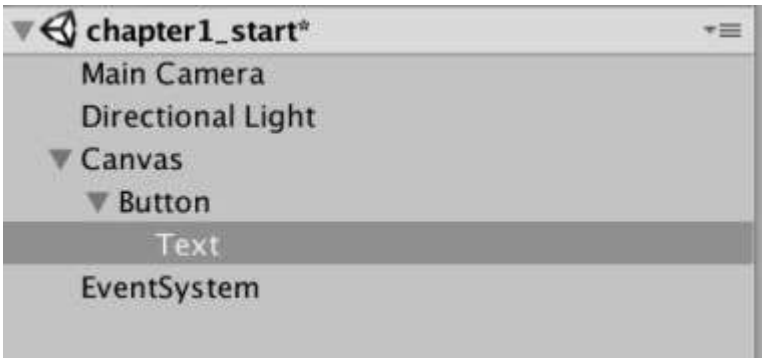


Figure 8-24: Setting the text for the button

Using the in the component called change the text attribute to >> Start as described in the next figure.

Figure 8-25: Changing the text for the button

Figure 8-26: The button with the new text

Next, we need to create a script (and the associated empty object) that will be used to trigger an action when the button is pressed.

Please create a new empty object | Create and rename it

Then, from the Project window, select: Create | C#
Rename the new script ManageButtons and open it.

We can now add the code that will initialize the score; the new code will also be used to launch the game when the button is pressed.

Please add the following code at the start of the script (new code in bold):

```
using UnityEngine;
using System.Collections;
using UnityEngine.SceneManagement;
public class ManageButtons : MonoBehaviour {
```

In the previous code we import the library called as we will be using the class SceneManager in the next code to load a new The class SceneManager is part of the library called

Please modify the Start function as follows:

```
void Start ()
{
PlayerPrefs.SetInt (“score”, 0);
}
```

In the previous code, we declare and initialize a variable called this variable is saved in the User which means that it will be accessible throughout the game (i.e., from any scene).

Please create the following function just before the end of the class:

```
public void startWordGame()
{
(“wordGame”);
}
```

In the previous code, we define the function when this function is called, the scene called wordGame is loaded.

Next, we will link the button to the function

Please save the script and check that it is error-free.
Once this is done, drag and drop the script ManageButtons from the Project window to the empty object manageButton in the as illustrated in the next figure.

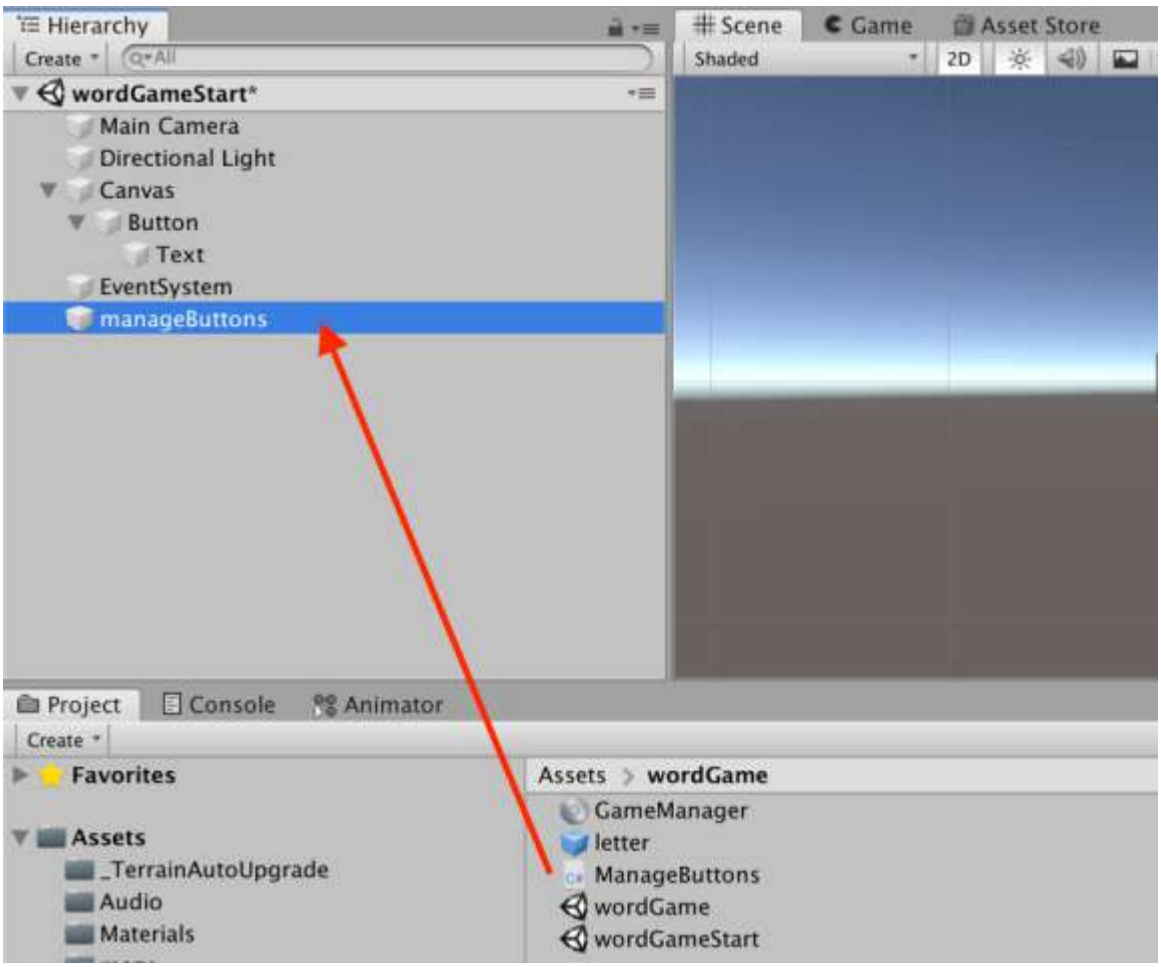


Figure 8-27: Adding a script to the empty object

You can then select the object manageButtons in the Hierarchy and look at the Inspector window; you should see that the component ManageButtons has been added.

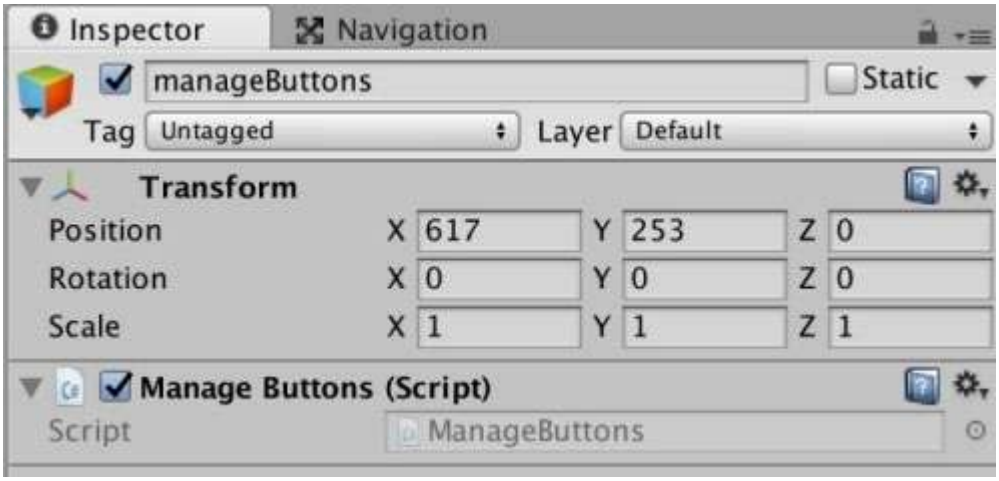


Figure 8-28: Checking the new component

Next, we will set-up the button so that the function startWordGame is called whenever this button is pressed.

Please select the object called Button in the
Using the in the section called click on the + sign that is below the label List is

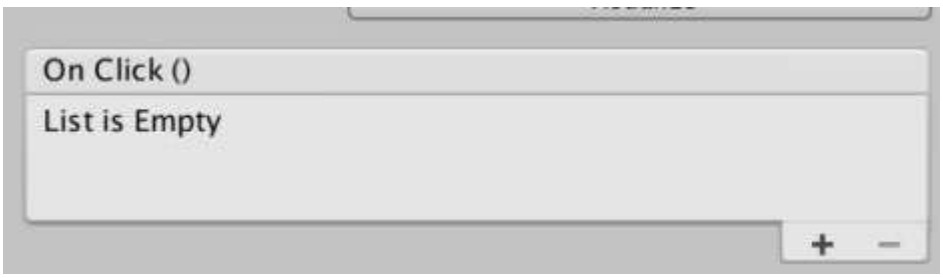


Figure 8-29: Handling clicks (part 1)

Once this is done, this section should now include a new field called None as illustrated on the next figure.

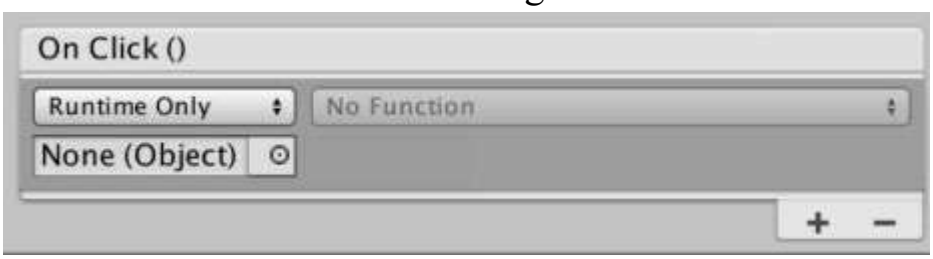


Figure 8-30: Handling clicks (part 2)

You can then drag and drop the object called manageButtons from the Hierarchy to the section called None (Object) for this object.

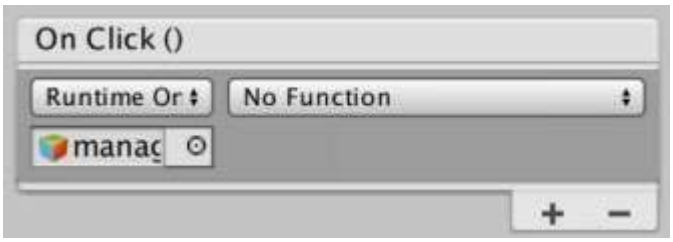


Figure 8-31: Handling clicks (part 3)

Following this, you can then click on the drop-down menu called and select: ManageButtons | By doing so, we specify that if the button is pressed, the function startWordGame from the script ManageButtons should be called.

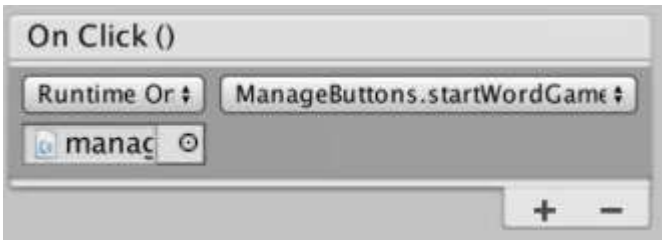


Figure 8-32: Handling clicks (part 4)

Last but not least, we just need to make sure that all scenes to be played in the game are part of the build settings; this is a way to declare the scenes that should be loaded to Unity.

Please open the Build Settings by selecting File | Build
Drag and drop the scenes wordGame and wordGameStart from the Project view to the Build Settings window.

You can unselect any of the other scenes used previously in this book.

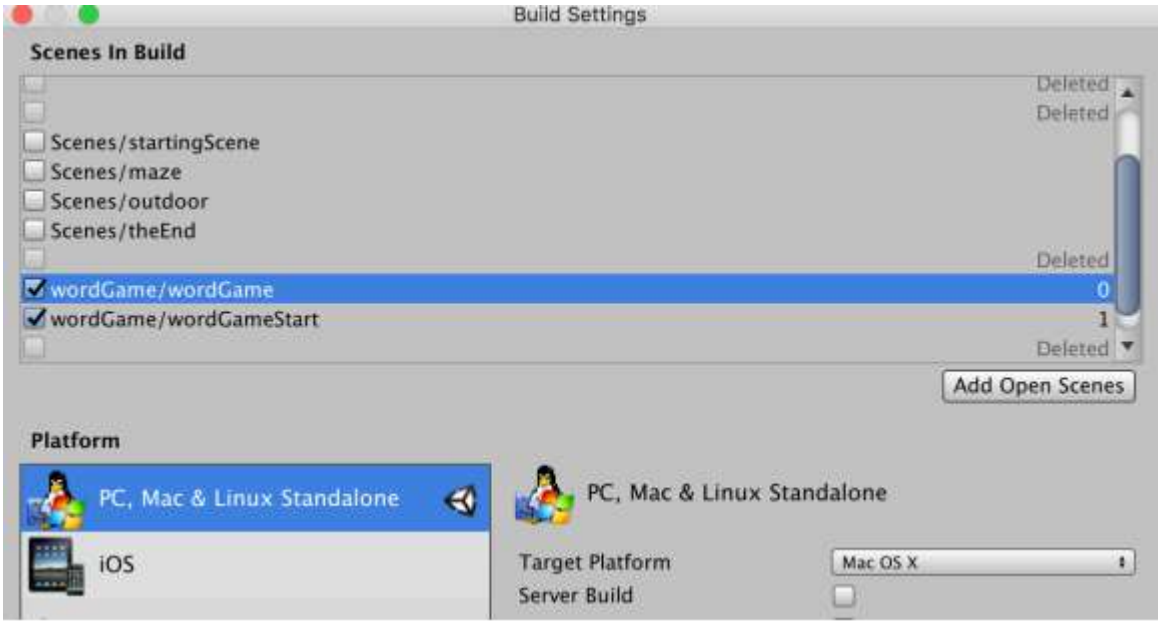


Figure 8-33: Updating the Build Settings

You can now, play the scene (i.e., and check that the game starts after you press the START button.

Next, we will be focusing on recording the number of attempts made by the player. So we will create some code that will track how many guesses the player has made, in the view to evaluate if the player has reached the maximum number of attempts (to be defined later in this section).

Please save the current scene +S or APPLE +
Open the scene called
Open the script called
Add the following code at the beginning of the class.

```
private int nbAttempts,maxNbAttempts;
```

Add the following code to the Start function:

```
nbAttempts = 0;
maxNbAttempts = 10;
```

Note that the number of attempts has been set to 10 arbitrarily; however, to give a fair chance to the player when long words are to be guessed, the variable maxNbbAttemps should be set to be at least the length of the word to be guessed. For this purpose, you could, if you wished, modify the method called to include the following line (new code highlighted in bold):

```
lengthOfWordToGuess = wordToGuess.Length;
wordToGuess = wordToGuess.ToUpper();
maxNbAttempts = wordToGuess.Length*2;
```

In the previous code, we set the number of attempts to be twice the length of the word to be guessed; if you were want to modify the difficulty level, you can change this factor from 2, to 1 or 3.

Please add the following code to the function checkKeyboard2 (new code in bold):

```
if (letterPressedAsInt >= 97 && letterPressed <= 122)
{
nbAttempts++;
UpdateNbAttempts();
```

In the previous code, we increase the value of the variable and we then refresh the user interface by calling the function that we will define in the next paragraph.

Please create the new function UpdateNbAttempts as follows:

```
void UpdateNbAttempts()

{
```

GameObject.Find (“nbAttempts”).GetComponent ().text = nbAttempts + “/” + maxNbAttempts;

}

In the previous code, we update the text of the Text UI object nbAttempts (that we will create in the next section) so that it displays the number of attempts made by the player.

Please add the following line to the Start function.

UpdateNbAttempts();

Please save your script, and check that it is error-free.

Finally, we just need to create a UI Text object called nbAttempts where the number of attempts will be displayed.

Please create a new UI Text component | UI | and rename it
Change its color to change its font-size to empty its text, and, using the Move tool, place this object in the top-left corner of the white rectangle that defines the viewable area for the player, as described on the next figure.

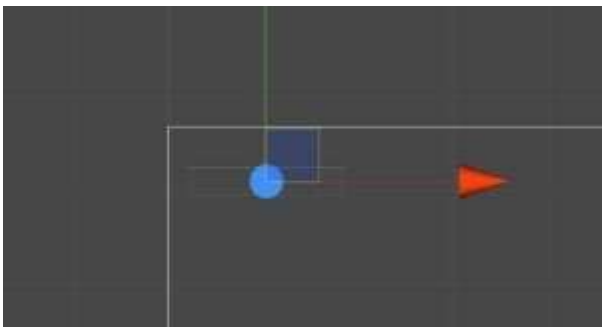


Figure 8-34: Modifying the user interface

Once this is done, you can test the scene and check that the user interface is updated every time you make a guess (i.e., when you press a letter on the keyboard).

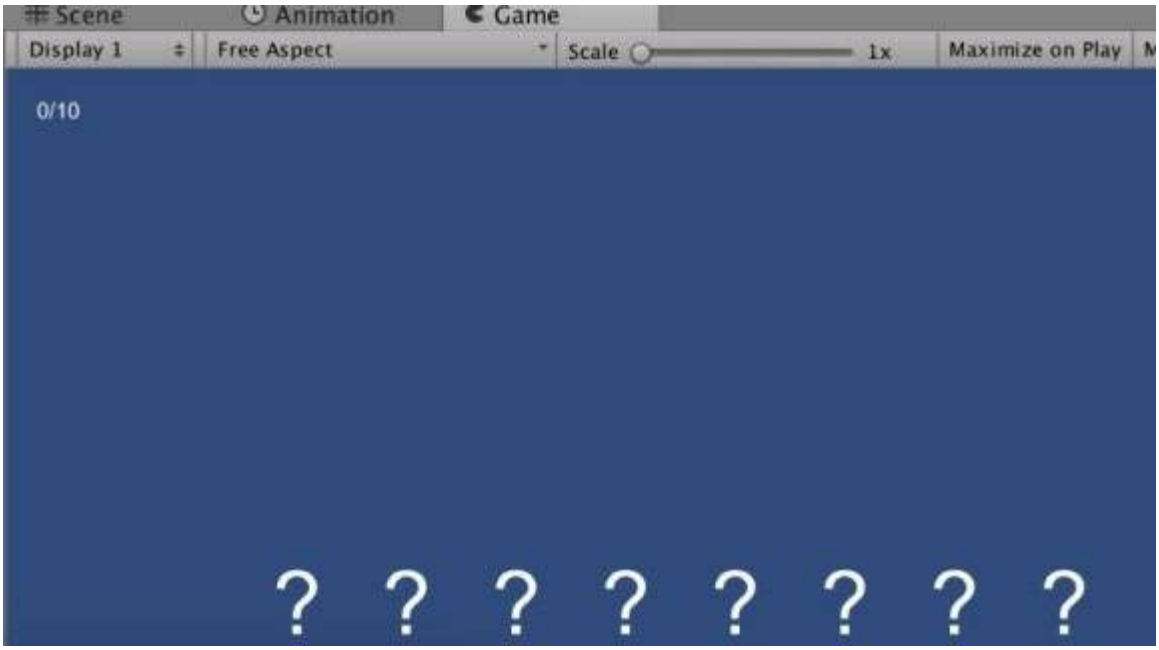


Figure 8-35: Displaying the number of attempts

Next, we will update the score. So the process will be similar to what we have done previously, as we will:

Create a UI Text object that will display the
Increase the score whenever a letter has been found.
Update the text of the corresponding UI Text object.

So let’s gets started:

Open the scene called
Using the Hierarchy window, please duplicate the object called nbAttempts and rename the duplicate
Move this object (i.e., to the top-right corner of the screen.

Figure 8-36: Displaying the score

Once this is done, we can update our script:

Please open the script called

Add this code at the beginning of the class.

```
int score = 0;
```

Add the following function just before the end of the class (before the last closing curly bracket) so that we can update the score onscreen.

```
void UpdateScore()
{

GameObject.Find (“scoreUI”).GetComponent ().text = “Score:” +
score;
}
```

Add the following code to the Start function.

```
UpdateScore();
```

Modify the function checkKeyboard2 as follows (new code in bold):

```
if (lettersToGuess [i] == letterPressed)
{
lettersGuessed [i] = true;
GameObject.Find("letter"+(i+1)).GetComponent().text =
letterPressed.ToString();
score = PlayerPrefs.GetInt ("score");
score++;
PlayerPrefs.SetInt ("score", score);
UpdateScore ();
}
```

In the previous code, we just increase the score by one every time the player has guessed a letter correctly; the score accessed from the Player it is increased by one then saved in the Player and the interface is then updated accordingly through the function

Please save your code, and check that it is error-free. As you play the scene the score should be displayed in the top-right corner, as illustrated in the next figure.

Figure 8-37: Displaying the score

Last but not least, we will modify the script called ManageButtons so that the score is only initialized in the splash-screen.

Please open the script
Modify the Start function as follows (new code in bold):

```
    void Start ()
    {
    if (SceneManager.GetActiveScene().name == “wordGameStart”)
PlayerPrefs.SetInt (“score”, 0);
    }
```

In the previous code, the score is initialized only if we are in the scene called wordGameStart (i.e., the splash-screen).

Now, the last thing we need to do is to check whether the player has guessed the word using less than the maximum number of attempts allowed; if this is the case, a win scene will be displayed; otherwise, if the player has failed to guess the word within the maximum number of attempts allowed, a scene called lose will be displayed instead. So first, we will create these two scenes (i.e., win and lose).

Please duplicate the scene called in the Project view, select the Scene called and press CTRL +
This will create a duplicate.
Please rename the duplicate
Open this Scene (i.e.,
Select the object called Text in the

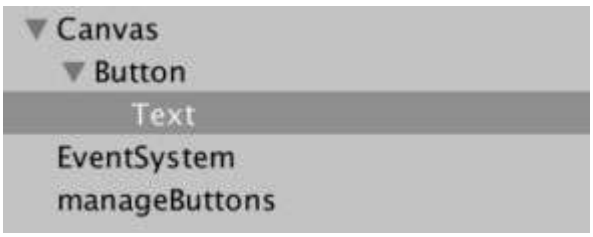


Figure 8-38: Changing the label of the button

Using the for the component called change the attribute text to >>
RESTART

Figure 8-39:Changing the label of the button

If you wish, you can also add a UI Text object with the text as follows.

Create a new UI Text object and rename it

Using the change this object's width to 600 and its height to
Align its text so that it is centered vertically and horizontally, using the section called Paragraph in the component called
Change the font-size to
Change the font-color to
Center the text both horizontally and vertically.

Please make sure that there is no overlap between the UI Text object and the button, otherwise clicks on the button may not be detected.

Change the text to
The scene should look like the next figure.

Figure 8-40: Displaying the game-over screen

You can now save the scene.

Now that the scene called wordGameEnd has been created, we can modify the code used in the game scene so that the wordGameEnd scene is loaded when the number of guesses is more than the maximum allowed.

Please save the current scene (i.e., press CTRL +
Open the scene
Open the script called
Add the following code at the beginning of the class.

```
using UnityEngine.SceneManagement;
```

Add the following code to the function checkKeyboard2 (new code in bold).

```
if (letterPressedAsInt >= 97 && letterPressed <= 122)
{
nbAttempts++;
UpdateNbAttempts ();
if (nbAttempts > maxNbAttempts)
```

{

SceneManager.LoadScene (“wordGameEnd”);

}

In the previous code, we check whether we have reached the maximum number of attempts; in this case, the scene called wordGameEnd is loaded.

Please save your script and check that it is error-free.

We can now add the scene wordGameEnd to the Build Settings (i.e., select File | Build as we have done for the other scenes.

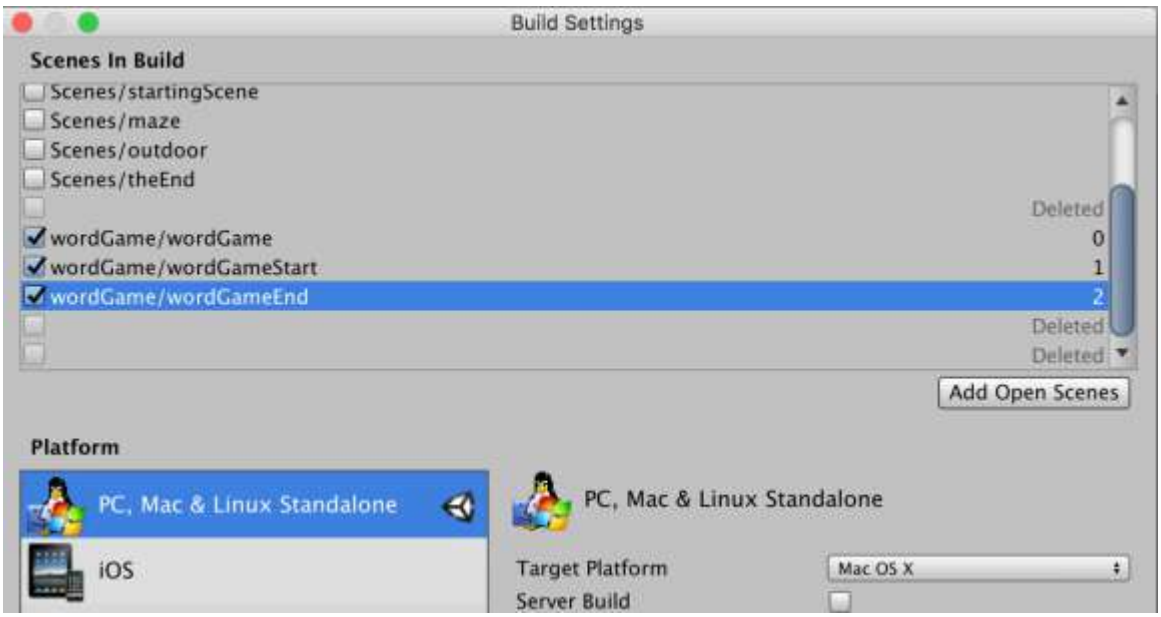


Figure 8-41: Adding a Scene to the Build Settings

Once this is done, you can play the scene; as you reach the maximum number of attempts, the scene wordGameEnd should be displayed.

Next, we will create code that will assess whether the player has managed to guess all the letters in the word. For this purpose, we will do the following:

Create a function that will be called whenever the player has correctly guessed a letter.
This function will check if all the letters were guessed accurately.
In this case, it will save the word to be guessed, and then display it in the win scene.

So let's create this function:

Please open the script
Add the following script at the end of the class.

```
void CheckIfWordWasFound()
{

bool condition = true;
for (int i = 0; i < lengthOfWordToGuess; i++)
{
condition = condition && lettersGuessed [i];
}
if (condition)
{
PlayerPrefs.SetString ("lastWordGuessed", wordToGuess);
SceneManager.LoadScene ("wordGameWin");
}
}
```

In the previous code:

We define a function called
We then declare a Boolean variable called condition that will be used to determine if all the letters were found.
This variable called condition is initially set to true.

We then go through each variable of the array called letterGuessed to check the letters that were guessed correctly by the player. If one of the word's letters was not found (i.e., even just one), the variable called condition will be set to false.
The following code effectively performs a logical AND between all the variables of the array as all of them need to be true (i.e., found) for the variable condition to be so this is the same as saying letter1 was guessed, and letter2 was guessed and letter3 was guessed, ..., and the last letter was guessed then the condition is true"

```
condition = condition && lettersGuessed [i];
```

If the variable condition is we then save the word that was guessed in the player preferences, so that it can be accessed and displayed in the next scene.
We then load the win scene (that we yet have to create).

We can now add a call to this function from the checkKeyboard2 function, as follows (new code in bold):

```
score++;
PlayerPrefs.SetInt ("score", score);
updateScore ();
CheckIfWordWasFound ();
```

So, we just need to create that new scene:

Please save your code and check that it is error free.
Using the Project window, duplicate the scene called wordGameEnd + and rename the duplicate
Open the new scene called

Using the duplicate the object called message (the one used to display the message and rename the duplicate
Using the Move tool, move the object wordGuessed below the button that is already in the scene, as in the next figure.

Figure 8-42: Moving the new UI Text

We can change the text of the UI Text called so that it now displays the text
We can also empty the text of the UI Text object wordGuessed so that the interface looks like the next figure.

Figure 8-43: Modifying the GUI

We will now create a script that will be attached to the object wordGuessed and that will display the actual word that was just guessed by the player.

Please create a new C# script called
Drag and drop this script on the object called
Open this script.
Add this code at the beginning of the script.

```
using TMPro;
```

Modify the Start function as follows:

```
void Start ()
{
GetComponent().text = PlayerPrefs.GetString (“lastWordGuessed”);
}
```

Please save your script and your scene.
Add your scene to the Build

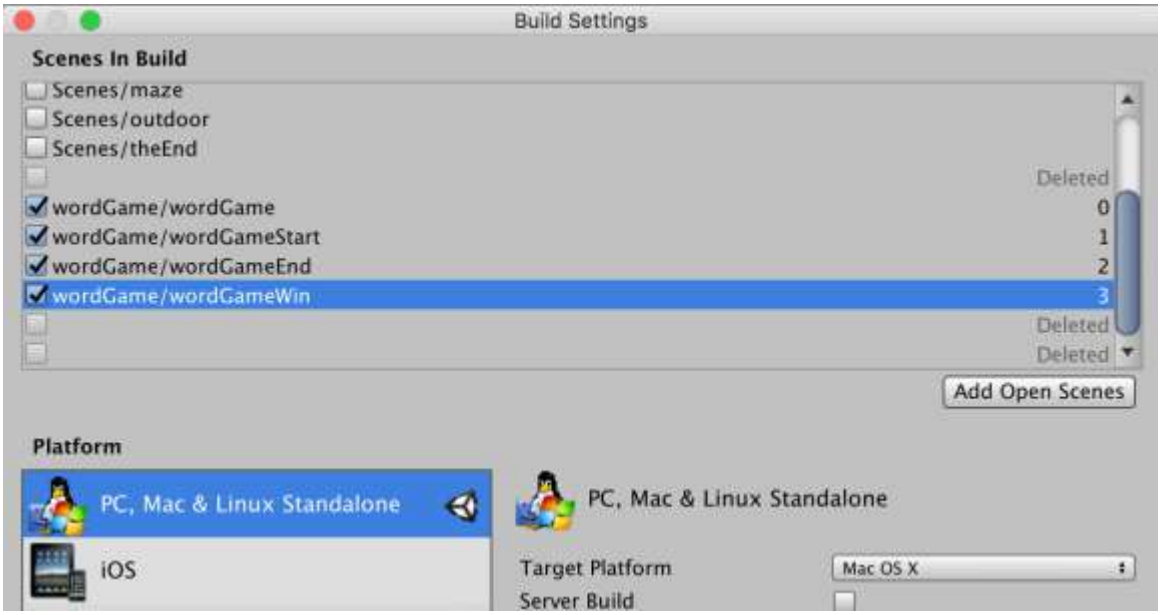


Figure 8-44: Adding the last Scene to the Build Settings

We can now open the main scene test the transition to the wordGameWin scene.

Figure 8-45: Displaying the win screen

Choosing words from a file

At this stage, the word game is pretty much functional with words selected at random; this being said, the words that we are using are part of an array that we need to update manually; if you were to include 100

words, you would need to enter them manually, which could be time-consuming. So, in this section, we will use a technique that consists of selecting a word from a pre-existing list of words saved in a text file. Because such files are available on the Internet, you could virtually create a word guessing game in several languages, by just modifying the file that contains these words.

So we will proceed as follows:

Import a file with that includes a list of words.
Add this file to a folder in Unity, where it can be accessed from a script.
Access this file from our script.
Pick a random word from this file.

So let’s get started.

Please create a new folder in the Assets folder, and call it select the Assets folder in the Project window, and then select Create | Folder from the Project window.
Rename this folder

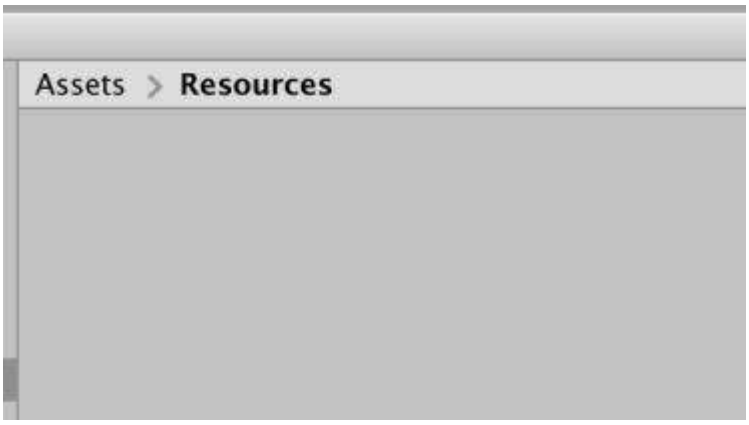


Figure 8-46: Adding a new folder

Once this is done, please import the file called words.txt from the resource pack that you have downloaded from the companion website to the folder Resources that you have just created.

Figure 8-47: Importing the word file

Next, we will modify our code:

Please open the script
Add the following code just before the end of the class.

```
string pickAWordFromFile()
{
TextAsset t1 = (TextAsset)Resources.Load("words",
typeof(TextAsset));
string s = t1.text;
string[] words = s.Split ("\n"[0]);
int randomWord = Random.Range (0, words.Length + 1);
return (words[randomWord]);
}
```

In the previous code:

We create a variable of type TextAsset that will point to the text file that we have just imported.
We then store the text from this file in a variable called

Since this file consists of one word per line, we split this string into lines, so that the variable words now contains an array of all the words (i.e., lines) included in the file.

Note that the command string.Split() will split a string based on a specific character; in our case it is the end of line which is symbolized in computer terms by

The syntax words = s.Split (“\n”[0]); means that we split the string called s based on the separator of (i.e.,

Last but not least, we just need to pick one of these words at random.

Please modify the function called initGame as follows.

```
//wordToGuess = wordsToGuess [randomNumber];
wordToGuess = pickAWordFromFile ();
```

That’s it.

You can now try the game again and check that it works.

There are a few things that you could then modify, for example:

You could set the number of attempts to the number of letters in the word to be guessed.

You could also display the word that was to be guessed in the scene

Finally, you could also use sound effects to provide feedback on whether the letter that was selected was correct.

Level roundup

In this chapter, we have learned to create a simple word guessing game where the player can guess the letters of a particular word. Along the way,

we have learned a few interesting skills including: generating random numbers, reading words from a file, using player preferences to save information between scenes, or detecting the player's input. So, we have covered considerable ground to get you started with your first word game!

Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Quiz

It is now time to test your knowledge. Please specify whether the following statements are TRUE or FALSE. The answers are available at the end of the book.

Please specify whether the following statement is TRUE or

The following code will declare an array of integers.
int [] i = new int [];

Please specify whether the following statement is TRUE or

The following code will declare and initialize an array of string variables:
string [] wordsToGuess = {“car”, “elephant”,”autocar” };

Please specify whether the following statement is TRUE or

The following code will check whether the player has pressed the key called A.
if (Input.GetKeyDown(A))

Please specify whether the following statement is TRUE or

The following code will display the number of characters in the string

```
string s = “Hello”;
print(s.Length);
```

Please specify whether the following statement is TRUE or

A char variable can be used to store a name with more than two letters.

Please specify whether the following statement is TRUE or

A string variable can be used to store a name with more than two letters.

Please specify whether the following statement is TRUE or

The following code will generate a random number between 0 and 100.

```
float randomNumber = Random.Number (0, 100);
```

Please specify whether the following statement is TRUE or

The first element of an array starts at the index 1.

Please specify whether the following statement is TRUE or

The first element of an array starts at the index 0.

Please specify whether the following statement is TRUE or

The following code will store the score in the player preferences:
PlayerPrefs.SetInt (“score”,score);

Challenge 1

Now that you have managed to complete this chapter and that you have created your first level, you could improve it by doing the following:

Set the number of attempts to the number of letters in the word to be guessed.
Display the word that was to be guessed in the scene

Challenge 2

Another interesting challenge could be as follows:

Create a text file of your choice with a word on each line.
Use this word file instead for your game.

9

Saving and Loading Information with local files in C#

In this section, we will learn to save information to a file, as this can be very useful in several applications, including games. So this will include features such as:

Saving a number or text to a file.
Saving information about a player such as its score, the las level achieved or its last location in the game.
Saving multiple record (i.e., for several players) into one file.
Saving more complex information using JSON files.

So, after completing this chapter, you will be able to:

Save and load single and multiple records to and from a text file.
Save and read more complex information to and from a JSON file.
Save and read more complex information to and from an XML file.

The code solutions for this chapter are included in the resource pack that you can download by following the instructions included in the section entitled and Resources for this

.

Saving single records

In this section, we will start by saving simple information about a user/player in the form of text or numbers.

Please create a new scene, and rename it
Next, please create a new C# script called SaveAndLoad and open it in your editor.
Please add the following code at the start of the script

```
using System.IO;
```

This code ensure that we can use libraries to read or write files

Please add the following function to it.

```
void SaveData1()
{
print("Saving to File");
File.WriteAllText(Application.dataPath + "/gameData.txt","test");
print("Saving complete");
}
```

In the previous code:

We create a function that will save text to a file called gameData.txt
We write messages in the console window before and after the data has been written
As we write the data, we use the built-in function and specify the location of the file where the data is to be written, in relation to where the current application is saved

Next, we need to create the code that will read this file to ensure that we have written the information correctly.

Please add the following function to the script:

```
void LoadData1()
```

```
    {
    print("Reading Data from File");
    string savedData = File.ReadAllText(Application.dataPath +
"/gameData.txt");
    print ("Data=" + savedData);
```

_____

```
}
```

In the previous code:

We declare a function called LoadData1 that will be used to read the data that we have written in the text file.
We print information in the Console window before and after reading the file.
We read the file using the function to read the content of the file gameData.txt and save this content in the variable called
Once this is done, we display a message that includes the data read in the text file.

So at this stage, we have created two function to respectively save data to, and read data from a file. The next thing we need to do is to call these functions when the user presses a specific key. So, please add the following function to the script (new cold in bold).

```
    void Update()

    {
    if (Input.GetKeyDown(KeyCode.S)) SaveData();
    if (Input.GetKeyDown(KeyCode.L)) LoadData();
    }
    void SaveData()
```

```
{
SaveData1();
}
void LoadData()
{
LoadData1();
}
```

In the previous code:

We write code in the function that is by default called every frame; so every frame, we check whether the player has pressed the keys S or
If they press the key then the data is being saved through the functions SaveData and
If they press the key then the data is being loaded through the functions LoadData and

You can now save your code, and check that it has compiled properly.

So now that we have created the necessary code to save and load data, it will need to be attached to an object in the scene so that it can be executed.

Please select: GameObject | this will create an empty object and add it to the scene.

You can rename that object saveLoadData and also drag and drop the script that you have just created So, at this stage, our scrip is linked to an object that is part of the scene.
Please play the scene by pressing the Play button, click once inside the Game view (so that clicks are detected), then press successively the S key

(to save the data) and then the L key (to load the data) and you should see the following messages in the Console window.

Saving to File
Saving complete
Reading Data from File
Data=test

So, if you have managed to see these messages, it means that you have successfully written your code to save and load data from a text file.

So, as you have seen in the previous section, we can write text in a text file, but we could also easily write a number; Let's see how:

Please stop the scene.
Open the script.
Add the following code.

```
void SaveData2()
{
print("Saving to File");
int nbLives = 2;
File.WriteAllText(Application.dataPath +
"/gameData.txt",""+nbLives);

print("Saving complete");
}
```

In the previous code, we do the same as in the function SaveData1, except that a number (the number is saved instead of a string.

Please add the following script:

```
void LoadData2()
{
print("Reading Data from File");
int savedNbLives  = int.Parse(File.ReadAllText(Application.dataPath + "/gameData.txt"));
print("Loaded nbLives: " + savedNbLives);
}
```

In this function we do the same as in the function except that we save the result in an integer variable

We just need to modify the code in the functions SaveData and LoadData now (new code in bold):

```
void SaveData()
{
//SaveData1();
SaveData2();
}
void LoadData()
{
//LoadData1();
LoadData2();
}
```

You can now save your code, check that it compiles properly and play your scene.

Click once on the Game view and then press, successively the letters S and

You should see the following messages in the Console window:

```
Saving to File
Saving complete
```

Reading Data from File
Loaded nbLives:2

Saving multiple Data for 1 player

While in the previous section we have saved data separately (a string, and then a number), we will, in this section, save multiple data for a player; The ide is that, often in a game, you will want o save data related to the game played such as the player location, its name, or its score; so instead of saving this in different files, we will use one file to save multiple data about a player.

Please add the following function to your script.

```
void SaveData3()
{
int newScore = 30;
string playersName = "Patrick";
string [] multipleData = new string [] {""+newScore, playersName};
string dataToSave = string.Join("|",multipleData);

File.WriteAllText(Application.dataPath +
"/gameData.txt",""+dataToSave);
print("Saving complete");
}
```

In the previous code:

We declare a variable of type integer, named and set its value to
We declare a variable of type string, named playersName, and set ist value to

We declare an array of strings made of two strings: the value of the variable score (that will be converted to a string), and the value of th variable playersName (Patrick).
We then use the function string.Join to create a string that will include the value of the variables and separated by a pipe sign (for example
Finally, we write a message that indicates that we have saved the data.

Now that we have written the code to save the data, we can create a function that will read this data, bearing in mind that the data is written in the format

Please add the following function to the script:

```
    void LoadData3()
    {
    string dataRead = File.ReadAllText(Application.dataPath +
"/gameData.txt");
    string [] multiDataLoaded = dataRead.Split("|");

    int newScore = int.Parse(multiDataLoaded[0]);
    string newName = multiDataLoaded[1];
    print ("Data Loaded: score ="+newScore+", Name="+newName);
```

_____

```
}
```

In the previous code:

We read the content of the file, and save the data read inside the variable called

We then use the Split function to split the string variable so that each of the strings found around the | sign within, are saved as part of a string array called in other word, if we had the data test1|test2 in the file, we would create an array that includes two elements: test1 and
Since the array now includes two elements, the first element (at the rank 0) is saved in the variable while the second element (at the rank 1), is saved in the variable
Finally, we display the data that has been loaded (i.e., the score and the name).

We just need to modify the code in the functions SaveData and LoadData now (new code in bold):

```
void SaveData()
{
//SaveData1();
//SaveData2();

SaveData3();
}
void LoadData()
{
//LoadData1();
//LoadData2();
LoadData3();
}
```

You can now save your code, check that it compiles properly and play your scene.
Click once on the Game view and then press, successively the letters S and
You should see the following messages in the Console window:

Saving complete
Data Loaded: score =30, Name=Patrick

_____

Saving multiple Data for several players

In the previous section, we have managed to save multiple data for one user; following from this, we will use a similar technique to save multiple data for several users.

Please open the script and add the following code:

```
void SaveData4()
{
string multipleData = "Patrick|2*Mary|3";

File.WriteAllText(Application.dataPath +
"/gameData.txt",""+multipleData);
print("Saving complete");
}
```

We create a new string variable that includes information about two players; the * sign is used to separate each user, and the | sign is used to separate every information about each user. So, in this example, the first player has the data Patrick and
We then write this data to the file
Finally, we write a message in the Console window that specifies that the saving is complete.

Next, we need to create a function that will read the content that we have just written in the file.

Please add the following function to the script:

```
void LoadData4()
{
string dataRead = File.ReadAllText(Application.dataPath +
"/gameData.txt");
string newRecord;
for (int i = 0; i < 2; i++)
{
newRecord = dataRead.Split("*")[i];
print("Record " + i + ": name="+ newRecord.Split("|")[0] + "score=" +
newRecord.Split("|")[1]);
}
}
```

In the previous code:

We create string variable called dataRead and store the content of the data read in the file gameData in that variable.
We create a new variable called
We then create a loop that will make it possible to go through the data of the 2 users included in the file (i.e., Patrick and
We record the data related to each player in the variable newRecord by using the function Split.
Then, for each individual (using the variable we extract the corresponding name and score.

We just need to modify the code in the functions SaveData and LoadData now (new code in bold):

```
void SaveData()
{
//SaveData1();
//SaveData2();
//SaveData3();
SaveData4();
}
void LoadData()
{
//LoadData1();
//LoadData2();
//LoadData3();
LoadData4();
}
```

You can now save your code, check that it compiles properly and play your scene.
Click once on the Game view and then press, successively the letters S and
You should see the following messages in the Console window:

```
Saving complete
Record 0: name=Patrick score=2
Record 1: name=Mary score=3
```

Saving Multiple Data (Continued)

In the previous section, we have seen how we could save multiple records in one file so that information such as name and score can be stored and retrieved easily for each player. In this section, we will expand on this principle, and add ways to facilitate and clarify the data we want to

access using constant variables instead of indexes, that can be misinterpreted.

Please add this code at the beginning of the class:

```
const int PLAYER_NAME = 0;
const int PLAYER_POSITION = 1;
const int PLAYER_SCORE = 2;
const int PLAYER_LAST_LEVEL = 3;
const int PLAYER_DIFFICULTY_LEVEL = 4;
Vector3 playerPosition;
string playerName;
```

In the previous code, we declare constants that will be used to access the player's name, position, score, last level, or difficulty level, along with the position of the player (a vector) and the name of the player.

Please add this code, just after the code that you have just yped

```
public GameObject playerCharacter;
```

In the previous code, we declare a public variable of type as we will instantiate an object at the position stored in the file, as you would instantiate the player at its previous position as s/he resumes the game.

Add this function to the script:

```
void SaveData5()
{
string multipleData = "Patrick|2,1,2|100*Mary|5,1,1|100";
File.WriteAllText(Application.dataPath +
"/gameData.txt",""+multipleData);
print("Saving complete");
}
```

In the previous code:

We create three records this time, each with three types of information: a name, a set of 3 coordinates (i.e., the position of the player), and a number (the score).
We then save this information in the file
Finally, we write a message to specify that the saving is complete.

Now, lets create the function that will read this information:

Please add the following function to the script:

```
void LoadData5()
{
string dataRead = File.ReadAllText(Application.dataPath +
"/gameData.txt");

string newRecord;
for (int i = 0; i < 2; i++)
{
newRecord = dataRead.Split("*")[i];
string playersName = newRecord.Split("|")[PLAYER_NAME];
string coordinates = newRecord.Split("|")[PLAYER_POSITION];
int score = int.Parse (newRecord.Split("|")[PLAYER_SCORE]);
}
}
```

Please add the following code within the loop, just after the code that you have already typed previously:

```
float x,y,z;
```

```
x = float.Parse(coordinates.Split(",")[0]);
y = float.Parse(coordinates.Split(",")[1]);
z = float.Parse(coordinates.Split(",")[2]);
Vector3 newPosition = new Vector3(x,y,z);
```

In the previous

We create three float variables x,y and z that correspond to the coordinates of the player.
We then read the variable called coordinates which is in the format data1, data2, to extract each coordinate at the index 0 (for x), 1 (for y) and 2 (for z).

Finally, we create a new vector of type Vector3 using these 3 coordinates.

Finally, please add this code within the loop:

```
print ("Players Name=" + playersName);
print ("New Position="+newPosition);
print ("Current Score = " + score);
GameObject t = Instantiate (playerCharacter, newPosition, Quaternion.identity);
t.name = playersName;
```

In the previous code:

We print information abou the player (name, position and score).
We create a new GameObject at the position defined earlier.
We also set the name of this object to the name of the player.

Before we can use this new code, we just need to do two things: modify the function LoadData and SaveData, then create an object that will be used to represent the player at its new position.

Please modify the code in the functions SaveData and LoadData now (new code in bold):

```
void SaveData()
{
//SaveData1();
//SaveData2();

//SaveData3();
// SaveData4();
SaveData5();
}
void LoadData()
{
//LoadData1();
//LoadData2();
//LoadData3();
//LoadData4();
LoadData5();
}
```

You can now save your code, check that it compiles properly and play your scene.

Now, we just need to create a new object that will be used to represent the player, so please do the following:

Create a new Cube: GameObject | 3D Object |
Rename this object
Drag and drop this object to the Project window, so that it become a prefab.
Delete the object that you have created in the Hierarchy view.
Drag and drop the prefab called playerTemplate from the Project view to the place holder called playerCharacter for the script attached to the empty object

We are now ready to test the scene:

Play the scene
Click once on the Game view and then press, successively the letters S and
You should see the following messages in the Console window:

Saving complete
Players Name=Patrick
New Position=2,1,2
Current Score=100
Players Name=Mary
New Position=5,1,1
Current Score=100

One last thing: you may not want to display all the information for all users, but instead to display information specifically for a user, using its name as a way to identify him/her.

In that case, we could easily modify our code so that we do just that: display information of a user based on its name. In the next example, we will also save the difficulty level used when the player played before.

Let’s make that change:

Add the following function to the script:

```
    void SaveData6()
    {
    string multipleData =
"Patrick|2,1,2|100|week5Level2|level1*Mary|5,1,1|100|level3|level1";
    File.WriteAllText(Application.dataPath +
"/gameData.txt",""+multipleData);
    print("Saving complete");


    }
```

In the previous code we do as we have done for the function except that we also store the name of last level (or scene) for each player.

Please add the following function to the script:

```
    void LoadData6(string newName)
    {
    string dataRead = File.ReadAllText(Application.dataPath +
"/gameData.txt");
    string newRecord;
    for (int i = 0; i < 2; i++)
    {
    newRecord = dataRead.Split("*")[i];
    string playersName = newRecord.Split("|")[PLAYER_NAME];
    string coordinates = newRecord.Split("|")[PLAYER_POSITION];
    int score = int.Parse (newRecord.Split("|")[PLAYER_SCORE]);
    string lastLevel = newRecord.Split("|")[PLAYER_LAST_LEVEL];
    float x,y,z;
```

```
    x = float.Parse(coordinates.Split(",")[0]);
    y = float.Parse(coordinates.Split(",")[1]);
    z = float.Parse(coordinates.Split(",")[2]);
    playerPosition = new Vector3(x,y,z);
    if (playersName == newName)
    {
    playerName = playersName;


    SceneManager.LoadScene(lastLevel);
    }
    }
    }
```

In the previous code, we do the same as in the function except that we create a conditional statement where we check that the name of the player for which we will display information, is the same as the one we are looking for. In that case, we load the corresponding scene.

Please modify the functions LoadData and SavaData as follows:

```
    void SaveData()
    {
    //SaveData1();
    //SaveData2();
    //SaveData3();
    // SaveData4();
    //SaveData5();
    SaveData6();
```

```
}
```

```
void LoadData()
{
//LoadData1();
//LoadData2();
//LoadData3();
//LoadData4();


//LoadData5();
LoadData6("Patrick");
```

So at this stage, we can save data, read data for a specific player (in our case, the data for Patrick), and consequently open the last scene accessed by this player.

So, for this to work, we need to do several things:

Create a scene called level1 that should be loaded after pressing the L key. Ensure that this script will still be present in the new scene.
When the new scene is loaded, use the data to move the player object to the location stored in the data file.

So let's proceed:

Please add the following code at the beginning of the script, so that we can load new scenes:

```
using UnityEngine.SceneManagement;
```

Please create and add a tag called "saveData" to the object saveData that you have created for te hcurrent scene.

Add the following function to the scene:

```
void Awake()
{
```

```
    GameObject[] objs =
GameObject.FindGameObjectsWithTag("saveData");
    SceneManager.sceneLoaded += OnSceneLoaded;
    if (objs.Length > 1)
    {
    Destroy(this.gameObject);


    }
    DontDestroyOnLoad(this.gameObject);
    }
```

In the previous code we just ensure that the object tagged as "saveData" is kept even if we load a new scene. We also ensure that whenever a new scene is loaded, the function OnSceneLoaded (that we will write in the next section, is called).

Please add the following function to the script:

```
    void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
    if (scene.name != " SavingAndLoading ")
    {
    GameObject t = Instantiate (playerCharacter, playerPosition,
Quaternion.identity);
    t.name = playerName;
    }
    }
```

In the previous code: if we have loaded a new scene (i.e., if we are not in the original scene called we instantiate the player object at the location specified in the data file and also rename the player accordingly.

Please save your script.

Open the Build Settings add the scene level1 to the Build Settings
Play the scene
Click once on the Game view and then press, successively the letters S and

You should see that a new scene (level1) has been loaded and that an object called Patrick is present in the scene at the position

Using JSON and XML for more complex data

So, in the last section, we have seen how it is possible to store and read multiple data from multiple users, and this feature is quite useful because of the amount if data that we can store for each user, making it possible for the player to stop and resume the game at any stage; This being said, sometimes you may need to save more complex data, that would make the usual text file a not-so-easy format to do so. On the other hand, we could use a format that allows for more complex data to be solved, with minimum effort to save/read, and the format that can be used in that case is JSON.

So, in this section, we will see, how JSON, a common format used in the software industry, can be employed to save and read multiple information for players.

So the next steps will consist in:

Creating a class that gives a template for the data to be stored in the JSON file.
Create the code to save data in a JSON file based on that class.
Create the code to read the data present in the JSON file.

So first let's create the class to be used to store the player's information:

Please add the following code to the script:

```
private class DataToSave
{
public string name;
public int score;
```

```
public Vector3 position;
public string lastLevel;
}
```

In the previous code We create a new class called This class includes four member variables: a string variable called an integer variable called a Vector3 variable called and a string variable called

Next, well write the function that will save the data in a JSON file.

Please add this code at the start of the class:

```
DataToSave myData;
```

In the previous code, we create a new variable of type that will be used to save the data about the player.

Please add the following code to the script:

```
void SaveDataJSON()
{
myData = new DataToSave{name = "Patrick", score = 100, position = new Vector3(10,0,10), lastLevel = "level1"};
jsonText = JsonUtility.ToJson(myData);
File.WriteAllText(Application.dataPath + "/game.json",jsonText);
print (“Data Saved”)
}
```

In the previous code:

We save the data about the player Patrick in the JSON format in the variable called

So, for this object myData (an instance of the class we set the member variables and lastLevel as we create this object.
Next, we use the class JsonUtility to format the data in the JSON format. This data is then written to the file
Finally, we write a message that specifies that the data has been saved.

Next, we need to create a function that will read the JSON data.

Please add this function to the script:

```
void
    {
    string newText = File.ReadAllText(Application.dataPath +
"/game.json");
    DataToSave mySavedData = JsonUtility.FromJson(newText);
    print ("Saved Data: Name=" + mySavedData.name + "score= " +
mySavedData.score + "Position: " + mySavedData.position);
    playerPosition = mySavedData.position;
    playerName = mySavedData.name;
    SceneManager.LoadScene(mySavedData.lastLevel);
    }
```

In the previous script:

We read the JSON file and save its content inside the variable
We create a new object of type DataToSave and save the information from the file in that object.

We print a message that displays information about the data read and saved.
We set the name and the position of the player.
The corresponding scene is loaded.

The last thing we need to do is to modify the function saveData and LoadData so that we just the code that we have just written.

Please modify the functions LoadData and SavaData as follows:

```
void SaveData()
{
//SaveData1();
//SaveData2();
//SaveData3();
// SaveData4();
//SaveData5();
//SaveData6();
SaveDataJSON();
}
void LoadData()
{
//LoadData1();
//LoadData2();
//LoadData3();
//LoadData4();

//LoadData5();
//LoadData6("Patrick");
//LoadDataJSON();
```

Play the scene
Click once on the Game view and then press, successively the letters S and
You should see that a new scene (level1) has been loaded and that an object called Patrick is present in the scene at the position
You should also see the following messages in the Console window.

Saving Complete
Saved Data: Name=Patrick score=100 Position: (10,0,10)

Level Roundup

In this chapter, we have learned some concepts related to saving and loading information in text files, including the JSON format. Along the way, we also managed to keep scripts and objects, even when a new scene is loaded. So, we have covered considerable ground to get you started with files handling.

Checklist

Checklist Checklist

Checklist Checklist

Quiz

It is now time to test your knowledge. Please specify whether the following statements are TRUE or FALSE. The answers are available at the end of the book.

Using C #, it is possible to load and save data from and to a file.
The method Split can creates an array of strings.
The method File.WriteAllText can be used to write data in a text file.
The function Input.GetKeyDown can be used to detect the user's inputs.
An array can store several variables at a time.
A class can include several member variables.
It is possible to save multiple data about a player in a text file by using only one line that contains the characters | or *.
JSON can be used to store only text.
JSON can be used to store only numbers.
A class cab be used to specify the format and the type of data included in a JSON file.

10

Accessing and Updating a Database in C#

In this section, we will learn how to interact with a database from Unity, using some simple but effective techniques.

After completing this chapter, you will be able to:

Understand basic database concepts.
Understand how online databases can be accessed through scripting.
Understand how to access a database from Unity.
Save and access information about a player.

Introduction to Online Databases

As it is, you may know how to save information from a game so that it is kept between scenes using the player preferences. This being said, in many situations, there is data that you may like to keep centrally, so that it can be accessed by players (or Unity) regardless of the computer or device used by the player. This can, for example, consist of a list of high scores, players' preferences, or players' details. In this case, as for many web-based applications, these details are stored on a server in an online database.

So what's a database?

To put things simply, a database is a collection of tables; each table contains specific information arranged in columns and rows. Each column corresponds to a specific attribute (e.g., name, score, etc.), while a row refers to a particular data set (each data set has the same type of attributes, but the values may differ between rows). For example, you could have a

table that includes details about a player such as: login, password, high-score, last room explored, etc. So, to access this information, you would usually need to access the database and then select this particular table; once the table is selected, you would then look at the row that includes the information for a particular user or player. Each player could be identified by an id, or by their nickname. So, that's it in a nutshell.

Now, in web development, accessing the database is usually performed with what is called a server-side script; a server-side script is a script, or a piece of code, that is saved on the server; when called, it is executed on the server and the results are then sent to the client (for example, your browser). These scripts can be written in several types of languages including PHP (Personal Home Page or Hypertext Preprocessor), a very popular server-side scripting language.

So, in a typical setting, you would do the following:

Write a PHP script.
This script would then be saved on a server.
When executed (e.g., when a url that points to the script is entered in a browser), this script can do several things, including returning text (e.g., displayed onscreen), performing calculations, or accessing a database hosted on the same server.
In the latter case, the database can be a MYSQL database.

So let's look at what a PHP script looks like:

```
echo "Hello";
?>
```

In the previous script:

The code indicates that we start PHP commands.

The second line uses the command echo to print (or output) the text
The third and last line indicates the end of the PHP commands.

Since PHP is a web-based server-side scripting, it is possible to mix both PHP commands and HTML code in a given PHP script; to differentiate between these, PHP commands are usually included between the tag (or and the tag

We could then usually save this script on a server and give it a name (e.g., hello.php)
We would then typically execute this script by requesting the page hello.php and typing its url in a browser, for example:
yourserver.com/hello.php
You would then see the text in your browser

In other words, by entering the url of the script, you are asking the server to execute the script; and in this case, the script will display, through the command the text "Hello"

Now, of course, more complex tasks, including database access, can be done through PHP, but this is the general idea behind executing server-side scripts.

When you want to access an online database with Unity, the process is quite similar in the sense that you do the following:

Create a C# script in Unity.
Within this script, provide the url of the PHP script to be called.
Call this url from Unity, so that the script is executed.

Record the result sent from the PHP script in Unity (e.g., by using C#).

So now that you have a clearer idea of what server side scripting is, let's have a closer look at the nitty gritty of accessing a database, before we can translate this skill to Unity.

Accessing a database through PHP

To access a database, you usually need the following:

an IP address.
a user name.
a password.

When a database has been set-up, it is often associated with a user, its password, along with access permissions. The permissions for this user define what this user can or can't do with the database (e.g., read or write). Once a database and the corresponding user have been created, tables can be created for this database; put simply, tables are similar to a spreadsheet with columns and labels for each column, and a new row for every new record. For example, we could create a table that is labeled player and that includes details about each player; details about each player would then be recorded in a corresponding row; these details could be, for example, an id, a password, or a high-score. As for the variables used in C# or JavaScript, each of these rows have a type: the id and the score could, for example, be integers, and the login could just be text.

So let's look at a code example that illustrates how a database could be accessed through PHP. You don't need to write this code yet; this is just provided as an example, and we will get to create such scripts later in this chapter.

```
function connect()
{
$host="localhost";
```

```
$database="mydatabase";
$user="user1";
$password = "mypassword";
$error = "Cant connect";
$con = mysqli_connect($host,$user,$password);
mysqli_select_db($con, $database) or die("Unableto connect to database");
}
?>
```

In the previous code snippet:

We create a function called connect that will perform a connection to the database of our choice.
We declare variables that will be used to establish a connection with the database (note that each variable starts with the $ sign in PHP).
this variable defines the address of the host or the server that we would like to connect to; in our case, we will use the server where the PHP script is stored; this is usually referred as the
this variable refers to the database that we want to access; we will see, later in this chapter, how to set-up this database along with users that can access it.
this refers to the user we have defined for this database.

Note that several users can be granted access to a database, all with different levels of permissions. It is usually a good idea to define at least two levels of permissions; for example, one user with read-only and another user with read and write

this refers to the password defined for the user (s) stated above.

this will be the error displayed if PHP can’t manage to connect to the database.
We then create a connection to the server using the variable defined previously through the command is a built-in PHP function.
We finally select the database defined previously; if this is not possible (i.e., error when trying to access the database), we display an error message.

So at this stage, we can understand some of the code that can be used to connect to a server and a corresponding database; we will now look at how it is possible to read from a database in PHP.

In PHP, it is possible to request information from a database using what is called a query. This query is usually performed using what are usually called SQL SQL commands make it possible to select a database and a table to, for example, read information from or write data to a table. These commands can only be processed once a connection has been established with the server and the database.

So let’s say that we’d like to read all the records from a specific table; we could then use the following code:

```
$query = "SELECT * FROM players";
$result= mysqli_query($con, $query);
$n = mysqli_num_rows($result);
for ($i = 0; $i < $n; $i++)
{
$name = mysqli_fetch_assoc($result)["name"];
$score = mysqli_fetch_assoc($result)["score"];
echo $name."\t";
echo $score."\n";
}
```

In the previous code:

We create a new string called this string includes an SQL query that basically selects all records from the table called The star character * indicates that all records are selected (we will see, later-on, how we can focus on or select a particular record).
We then execute this query using the command the result of the query (the information returned by the database) is then stored inside the variable called
Because the table may include several records, we could obtain several records in the results and the number of records, in this case, is stored in the variable called

We then loop through all the records sent back from the database (in response to our query), and for each record we access a particular attribute (or column). So the attribute name is saved in the variable and the attribute called score is saved in the variable called $score using the command The mysqli_fetch_assoc command includes two parameters: (1) the result (records) returned by the database based on the query, and (2) the attribute that we are interested in.
Finally, we display the corresponding name and score.

Note that using SQL commands, it is also possible to sort the results based on specific criteria; for example, we could ask the database to return the name and score, and to order these in ascending or descending order of scores.

Let’s look at the next code snippet to illustrate this principle.

$query = "SELECT * FROM players ORDER by score DESC";

In the previous code, the SQL query requests all records from the table players; it also requests that these should be ordered in descending order of scores.

We could also, to make this even neater, limit the result to the top three scores using the following query.

$query = "SELECT * FROM players ORDER by score DESC LIMIT 3";

In the previous code, we add the command LIMIT to limit the number of records returned from the database to

One of the other interesting things we can do is to write data to a database; for example, you may register a new player in the database and save or update its score; this, again, can be done with SQL queries. So let's see how this can be done.

The following example illustrates an SQL query that inserts a new player as well as its score in the database.

$name = "Paul";

$score = 25;

$query = "insert into players values ('$name', '$score');";

In the previous code:

Two variables are declared for both the name and the score of the player. We then create a query that will insert a new record in the table called for this new record, the two columns for the player's name and score will be set with the values previously defined.

Now, the previous scenario may happen only when the user registers for the first time in the database; what may happen next, is that the player may need to save or update its score after each subsequent game. In this case, because the player's information have previously already been recorded in the database, we could use another type of SQL command, the command as illustrated in the next code.

$name = "Paul";

$score = 25;

$query = "UPDATE players SET score = '$score' WHERE name = '$name'";

In the previous code:

We define the variables $name and
We then create a query that updates the columns score in the table called players fort the record with the name So here, we need to be more specific so that we can access a particular row; this row is identified by its attribute called so it is assumed that all players have a different name in this case.

So, as you can see, using SQL commands you can do several things including reading a table, writing to a table, or even updating a table. For the latter, you need to specify an attribute that will identify the row that you need to specifically modify; this is often an id, but it can consist of other types of variables if need be.

Passing data to a PHP script

One of the last things that will be of help when transferring data between Unity and a database, is the ability to pass information to a PHP script; in many cases, you will want, not only to receive information and data from the script, but also to be able to pass data to the script, so that this data can be processed by the script and used to, for example, update the database or check for some information (e.g., login details). For example, you may want to check the login or password for a particular user, or specify what record should be updated and what data should be used in this case.

In PHP you can pass variables along with their values by adding them to the url of the PHP page as follows:

http://www.mysite.com/index.php?playerName=john&score=100

In the previous code:

We first use the url of the PHP page and then add variables after the url of the page.

http://www.mysite.com/index.php

A question mark is added to the url.

http://www.mysite.com/index.php?

Then the name of the variable and its value are passed using the syntax variable =

The & character is used between each pair of variables and values.
So here, we have two variables and along with their corresponding values and

So once this information is passed to the PHP script, it can be captured (and used) by the PHP script as follows:

```
$name = $_GET['playerName'];
$score = $_GET['score'];
?>
```

In the previous code:

We define two variables $name and

The first variable is initialized with the value of the variables playerName passed to the PHP script through the url (as we have seen earlier).
The same is done for the second variable, except that this time we use the variable

So that's pretty much it as far as PHP and MYSQL are concerned; there is, of course, more to SQL and PHP programming; however, to be able to access the database, the previous sections should be sufficient to understand how this works.

Accessing PHP from Unity

So when your PHP scripts have been created and your database set up, the last thing you will need to do will be to connect to the PHP script through Unity. This can be done using the WWW class, a class that makes it possible to retrieve content from a url, as illustrated in the next C# code snippet.

```
String url =
WWW www = new WWW(url);
yield return www;
```

In the previous code:

We create an object of type WWW that will point to a specific url.
We then wait until the content has been downloaded from this url.

Setting up your server

Great. So I hope that the whole principle behind accessing a database from Unity is clearer now (and if it's not, the next section will help you to put this knowledge into practice). So in the next sections, we will create a simple system whereby:

A player will be asked for his/her id.

If the id is already in the database, then the database will be updated with the new score.
If the id is not in the database, then a new record will be created for this player with an id of his/her choice, along with the new score.
Once the score has been saved, the top 5 scores (i.e., the high scores) will be displayed onscreen with their corresponding names.

So the workflow will be as follows:

Set-up the database and associated users.

Create a new table with some players and corresponding scores.
Create PHP scripts that will be able to either insert data in, update, or read from the database we have created.
Save these scripts on the server.
Design a simple user interface in Unity whereby players can enter their id or register.
Access the PHP scripts that we have created previously in order to perform the previous tasks.

So first, let's setup your server. In order to execute the PHP scripts, these scripts need to be saved on a dedicated server. This server will also make it possible to create and set up your database. If you already have your own website, the chances are that you also have different packages installed that make it possible to set-up a database, as described in the next figure.

Figure 10-1: Database tools included in web packages

For example, in the previous figure you can see some of the standard tools offered as part of your control panel (if you have a control panel included in your website package), to manage databases, including and MySQL The former will make it possible to create new databases, while the latter will make it possible to create and update new tables.

Now, you may not have a website with these tools, and that's perfectly fine too. In fact, if you have never done any server-side scripting before, I would suggest that you use a local server for the time being. You could, in this case, use a tool called WAMP (if you are using a windows machine) or MAMP (if you are using a Mac). This software includes an Apache server, one of the world's most used web servers, and PHP (so that you can run PHP scripts), along with MySQL. MAMP is available from the official website:

AMP stands for (the web server), and It is a package that includes all the elements needed to run PHP scripts and access a MYSQL database. I would suggest that you use this package (instead of a web server) for the next sections of the chapter, as it should make the creation of your database much easier.

When you are using this software (i.e., M-AMP or W-AMP), the only difference is that your server is hosted on your own computer (rather than online), which makes testing easier, because you do not need to be connected to the Internet in order to access the server or connect to the database. Once your code has been tested locally (i.e., using WAMP or MAMP), you can then create the database on the remote server and upload your PHP code so that the application runs "live", and so that it is accessible from anyone through the Internet.

Whether you are using a local server (i.e., MAMP or WAMP) or a remote server, you will connect to the server using the address localhost or 127.0.0.1 because the pages will be hosted where the server is (either on the remote or the local server).

So, if you are using a remote server (but again, I suggest that you use W/MAMP instead, as explained in the next sections), you will need to do the following:

Create a new database using the MySQL Databases tool

Create a user with sufficient rights to access the database and complete some commands such as READ, WRITE or
Add tables and their content to the database using PHPMyAdmin.

Creating your database using a remote server

You can skip this section if you are using (or prefer to use) a local server (e.g., W/MAMP). In fact, I would strongly suggest that you skip to the next section, called Creating a database using a local if you don't already have some experience with PHP or PHPMySQL or setting-up a remote server.

So, let's get started!

Please note that the look and feel of the control panels may differ across hosting companies; however, the principles and the steps explained in this section should remain relevant.

To create your database:

Connect to your website.
Open your control panel.
Click on MySQLDataBase tool.

Figure 10-2 :Selecting MySQL Databases on the remote server

Once the new page opens, enter the name of your database and click on Create

Figure 10-3: Creating a new database on a remote server

The following page will then be displayed to acknowledge that the new database has been created.

Figure 10-4: Confirming that the database has been created

You can then click on the button labelled Go to return to the previous page.

At this stage you should see that your new database has been created, as described on the next figure.

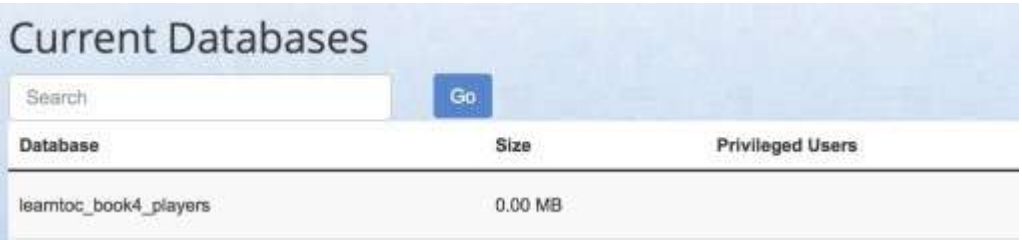


Figure 10-5: The new database has been created

However, you will see that it doesn't have any associated users who could be using it, or corresponding access rights. So we will need to add such users:

Please scroll down to the section called MySQL
Create a new user by setting a name and a

Figure 10-6: Creating a new user for the database

Please take note of the name of the user, along with its password, as we will need this information later-on when accessing the database through PHP.

Once this is created, a confirmation page should be displayed as follows.

Figure 10-7: Confirming the creation of a database user

You can then click on the button labelled Go to return to the previous page.

The last thing we need to do now is to associate this new user to the database that we have just created:

Please scroll down to the section called Add User to

Figure 10-8: Adding a user to our database

Make sure that the correct user and database are selected (using the drop-down menus). For example, in the previous figure you can see that I am setting privileges for the user learntoc_admin (created previously) to be linked to the database
Then click on the button labelled
This will open a new window labelled Manage User

Figure 10-9: Managing user privileges

In this window, select the privileges INSERT and UPDATE, as described on the previous figure. These are the key commands that we will need to perform.

Once this is done, click on the button labelled

The following window should then appear:

Figure 10-10: Confirming changes to privileges

Please click on the button called

So at this stage, if you are using a remote server (i.e., your web host), you have managed to create a database, an associated user, along with access rights for this user: so we are almost there :-).

Creating a database using a local server

So if you are using a local server (i.e., W/MAMP), the steps to create your database and associated user will be slightly different; however, the idea and principles will remain similar.

Please download the software MAMP (if you are using a Mac) or WAMP (if you are using a Windows-based computer) from the following site:

https://www.mamp.info/en/downloads/.

Install it on your computer following the instructions.

Once it is installed you can launch it, and the following (or similar window) will appear.

Bear in mind that the examples provided in the next section are from a Mac OS (i.e., using MAMP), so the screen on Windows computers may differ slightly; however, the layout should be similar.

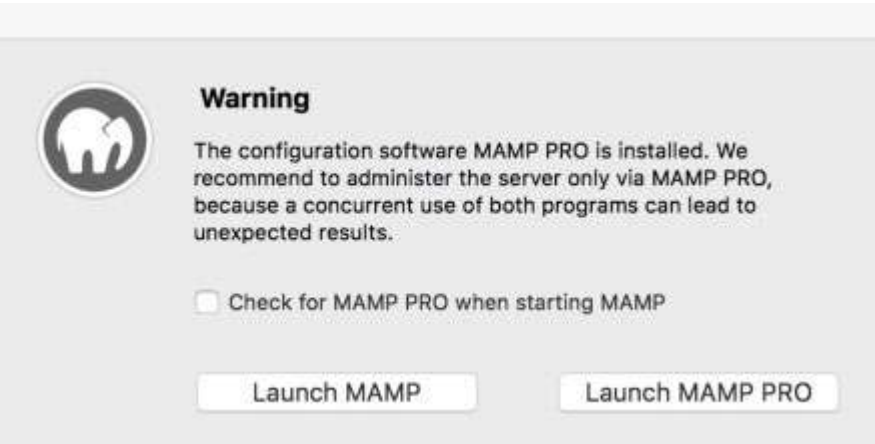


Figure 10-11: Launching your local server

Launch the application by clicking on

Figure 10-12: Selecting the servers to launch

In the next window, click on Start Servers (as described in the previous figure).

After a few seconds, you should see that the icons located in the top-right corner of the window are ticked (green) indicating that the Apache Server and MySQL servers have started.

A new page will also open in your browser, as described in the next figure.

Figure 10-13: Displaying the welcome page after a successful installation

This page includes some important information, including a default host name, a user name, and password for this user to access the server

through PHP, along with a PHP example.

Figure 10-14: Default settings for the database

Figure 10-15: Example PHP script

So, at this stage, our servers are running; the only thing that we need to do is to create a database; note that because this set-up will be used locally, we can use the default user root which has, by default, all access privileges to the database (e.g., READ, WRITE, UPDATE, and SELECT, etc.). So let's create a new database:

Select the tool PHPMyAdmin from the drop-down menu located at the top of the browser in the page that is already open (as described on the next figure).

If the MAMP window is not opened yet, you can open it using the url http://localhost:8888/MAMP/?language=English

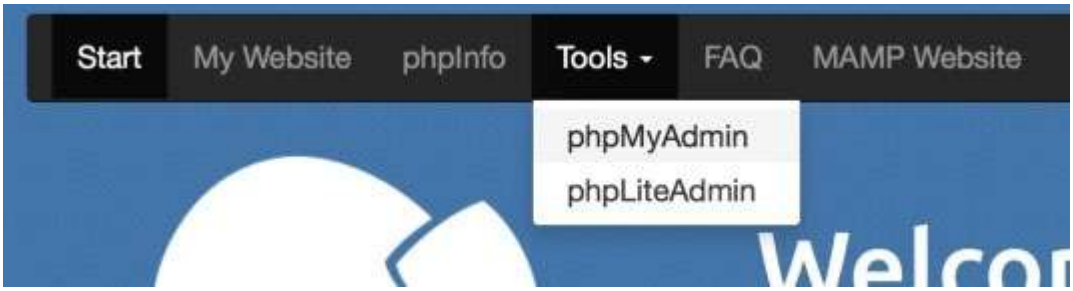


Figure 10-16: Accessing PHPMyAdmin

Note that you can also use the address http://localhost:8888/PHPMyAdmin/ to access PHPMyAdmin on your computer. This will open a new window with information on your MYSQL database.

Click on the tab labelled

Figure 10-17: PHPMyAdmin's front page

Give a name to your database, for example and click on the button labelled Create (leave the Collation default option).

Figure 10-18: Creating a new database on the local server

At this stage, as the database has been created, you will see it listed in the left frame of the window, as well as at the top of the window.

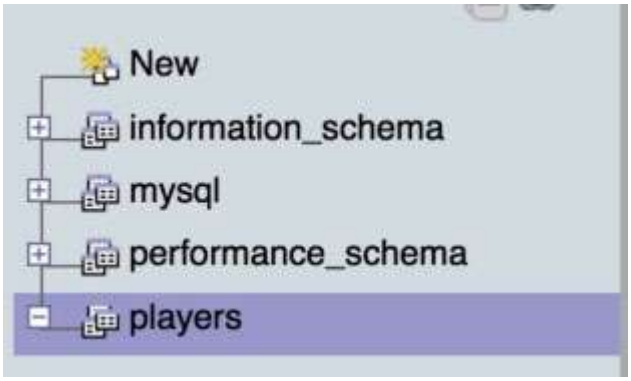


Figure 10-19: The new database listed in the left frame of the window

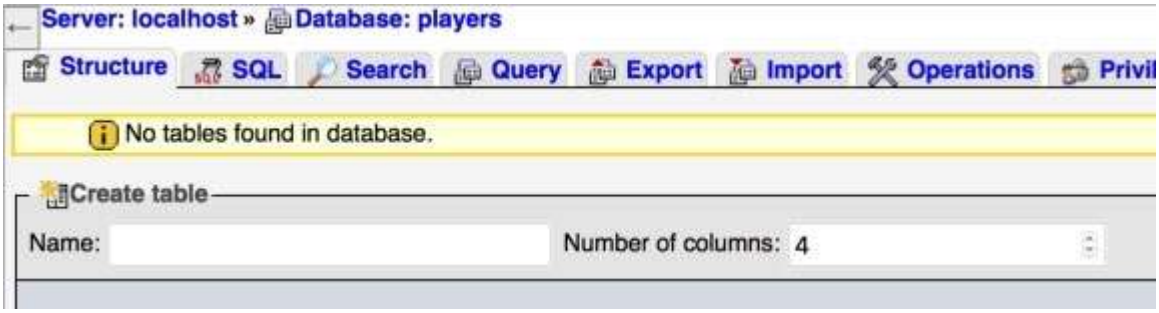


Figure 10-20: The new database listed at the top of the window

Please click on the tab called

Figure 10-21: Creating new privileges

You should see that, by default, a user root has already been created; the default password for this user is

Figure 10-22: The root user

So at this stage, we have managed to create a database on the local server (or the remote server if you have followed the previous section), and it is time to create a table that will hold information about the players' score.

Creating new tables

The following can be performed whether you use a local or a remote server. The only difference is the way you access PHPMyAdmin; you can use the address http://localhost:8888/PHPMyAdmin/ if you are using a local server.

Please open

Select your database (i.e., click on players in the left-hand menu).

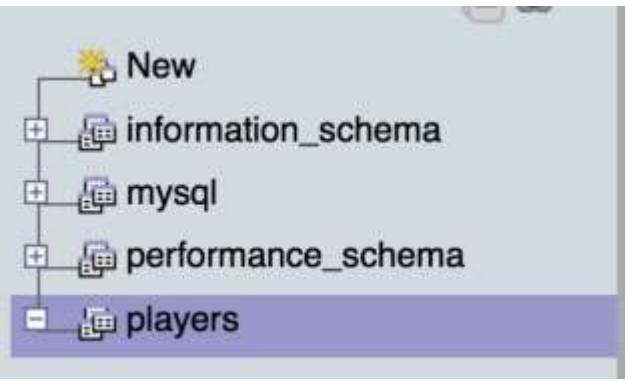


Figure 10-23: Selecting your database

Select the tab called This section makes it possible to define the structure of a table including the name and the type of the columns within.

Figure 10-24: Modifying the structure of the database

Within this tab, specify a new name for the new table, for example and a number of columns one for the name and one for the score), as illustrated in the next figure.

Figure 10-25: Initializing the table

Click on the button labelled located at the bottom of the page, to complete the creation of the table.
In the new window, that will be used to specify the name and type of the variables stored in the table, please enter the following information:

Figure 10-26: Defining the structure of the table

Row1:

Name = name
Type = TEXT
Length/Values = 100
Leave all other options as default.

Row2:

Name = score
Type = INT

Leave all other options as default.

Once this is done, please click on the button labelled located in the bottom-right corner of the window, as illustrated in the next figure.

Figure 10-27: Saving the table

Then the following window will appear, indicating that the table was successfully created; it should include the columns that we have defined previously (e.g., name and score).

Figure 10-28: The table has been created

So at this stage, we just need to add information to this table so that we can access it and test it accordingly from PHP.

Please select the tab called Insert from the top of the window; this section is used to insert rows (i.e., data) into our table.

Insert

Insert two new records as follows:

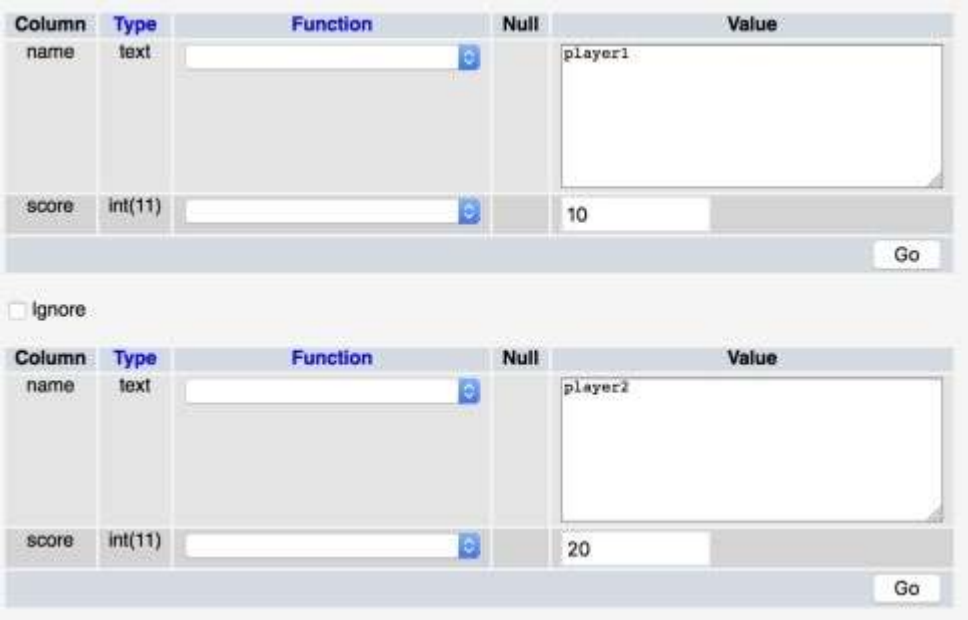


Figure 10-29: Adding record to the table

Record1:

Name = player1
Score = 10

Record2:

Name = player2
Score = 20

You can then click the button labelled Go located after the second record.

Note that you can add one or several records (or rows) at a time. This can speed-up the process of setting up your tables.

Although we don't need it now, for those used to SQL commands, the following could have been used within the SQL window (i.e., the SQL tab). This will have the same effect as creating the previous record manually.

INSERT INTO `players`.`high_scores` (`name`, `score`) VALUES ('player1', '10'), ('player2', '20')

So now that this has been done, it is time to use some PHP code to access the database.

Creating and running your PHP script

In this section, we will locate where to create and execute your PHP script on your computer, and then write the necessary code to access our database from your script.

If you are using a remote server, your script can be added anywhere within the public_html or www folder; this folder is usually the folder where all webpages to be accessed by Internet users can be seen and/or executed. So, if your script is called accessDB.php and is stored in the public_html folder, then it will be accessible using: In order to add this file to the public_html folder you can either use file managers included in your web package or an ftp client.

An FTP client is a software that makes it possible to connect to your website and to transfer files from or to it. If you are using an ftp client, you will need to connect using the details provided by your host (the name of the server, along with an ftp user and password).

If you are using WAMP or MAMP, the location for your PHP files (also referred as the www folder) will be as follows;

For MAMP (Mac OS) the folder is: Applications/Mamp/htdocs

For Wamp (Windows) the folder is: C:\wamp\www\
From now, this folder will be refereed as the www folder, whether you are using a local or remote server.

So, if your script is called accessDB.php and is stored in the www folder, it will be accessible using: if you are using a remote server, and if you are using a WAMP or MAMP server.

When using a local server through MAMP or WAMP, the url used to access your PHP script includes the address of the server (i.e., as well as a default port for AMP The port can be compared to a phone line through which the server listens to and replies to queries.

Next, we will be specifying the PHP version that we will be working with; for the code introduced in this chapter, the version 5 of PHP will be used, although if you read this book a few months or years after it has been released, this version may or may not be available. There are slight differences across PHP versions; this being said the concepts introduced in this chapter shall remain similar across versions (e.g., +/- minor necessary changes).

If you are using a local server:

Open the MAMP admin.
Click on the icon called

Figure 10-30: Opening the MAMP admin

In the new window, select the tab called
Select the Standard Version

Please note that these PHP versions may differ over time; however, the principle explained in the next pages should remain similar.

Figure 10-31: Selecting the PHP version

Click OK.
The server will then restart; you can then check that the correct version is running by opening the url: localhost:8888/MAMP/ in your browser, and

then by clicking on the tab called

Figure 10-32: Opening the PHP information page

This should display your PHP version at the top of the page.

Figure 10-33: Displaying the current PHP version

Next, we will also make sure that any PHP error is displayed onscreen, so that we can see if and where these occur; to do so, we will need to modify a file called the location of this file is provided in the phpinfo window in the section labeled Configuration So if your open the url and then click on the tab called you should see the corresponding section, as described in the next figure.

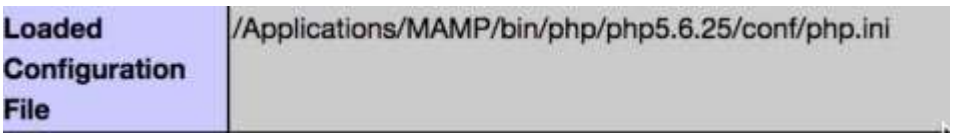


Figure 10-34: Finding the location of the PHP.ini file for a Mac computer

Please note that the previous figure shows the path to the file PHP.ini on a mac computer; the path for a windows machine should be different.

So we can now access and modify this file using the path provided by

Please open this file (i.e., using a text editor of your choice.

Search for the text

Once you have found it, please replace the text display_errors = Off with display_errors = On, as described on the next figure.

```
272  ; Print out errors (as a part of the output).  For production web sites,
273  ; you're strongly encouraged to turn this feature off, and use error logging
274  ; instead (see below).  Keeping display_errors enabled on a production web site
275  ; may reveal security information to end users, such as file paths on your Web
276  ; server, your database schema or other information.
277  display_errors = On
278
```

Figure 10-35: Modifying the PHP.ini file

Please save the file.

So that this change can be applied, we need to restart the server:

Please stop the server using the AMP control panel.
And restart it again.

Figure 10-36: Stopping the server

So let’s test our PHP settings:

Please create a new file with the text editor of your choice, and save it as updateScore.php inside the www folder.

For Mac OS computers, the www folder is located in Applications/Mamp/htdocs

For Windows computers the www folder is located in C:\wamp\www\

Add the following code to it.

```
echo "Hello World";
?>
```

Once this is done, save the file, and open this page in your browser, for example, by typing the following address.

http://localhost:8888/updateScore.php

If you are using a remote server, the url may look like:
This should display the text

Figure 10-37: Checking PHP

Once this is done, we can try to connect to our database. Note that if you are using a remote server, you will need to use the information that you created when initially you set up the database.

Please add the following code (in bold) to the file

```
echo "Hello World";
```

```
$host="localhost" ;
$database="players";
$user="root";
$password = "root";
$error = "Cant connect";


$con = mysqli_connect($host,$user,$password);
mysqli_select_db($con, $database) or die("Unableto connect to database");
?>
```

In the previous code:

As we have seen in the previous sections, we connect to our database using the default user and password for our localhost.
If no error message is displayed, then the connection has been successful.

Next, we will try to read some records from the database.

Please add the following code to the PHP script:

```
$query = "SELECT * FROM high_scores";
$result= mysqli_query($con, $query);
$n = mysqli_num_rows($result);
while ($row = mysqli_fetch_assoc($result))
{
$name = $row["name"];
$score = $row["score"];
echo "Name:".$name;
echo "Score:".$score;
}
```

Save your PHP file.
Switch to your browser and refresh the following page.

http://localhost:8888/updateScore.php

The following should be displayed:

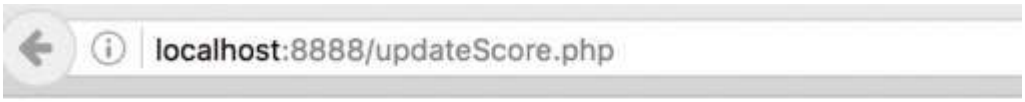


Figure 10-38: Displaying the output from the database after reading it

If you see this output, you have successfully managed to access and read data from the database. Congratulations!

Gathering data from Unity

Now that we know that our PHP script works and that we can access the database, we will modify it and create a new C# script in Unity so that we can read and display this information in Unity.

Please comment the following line in your PHP script, as follows:

//echo "Hello World";

In PHP, as for C# or UnityScript, you can comment your code using // to comment only one line or /* and */ at the start and the end of the section that you would like to comment.

Now that it is done, let's work on the Unity side of things and try to access this PHP script and gather its output (i.e., the list of players and their score) from Unity.

Please launch Unity.
Create a new project | New
Create a new empty object and rename it accessDB or a name of your choice Object | Create

Using the Project window, create a new C# script | C# called
Add the following code to it.

```
using UnityEngine;
using System.Collections;
public class AccessDB : MonoBehaviour {
string url = "http://localhost:8888/updateScore.php";
// Use this for initialization
IEnumerator Start()
{
WWW www = new WWW(url);
yield return www;
string result = www.text;
print("data received"+result);
}
void Update () {
}
}
```

In the previous code:

We declare the class
We then create a string called url that stores the address of the PHP page that we will access.
We declare a function called Start using the keyword IEnumerator beforehand. This keyword is used to specify that the function Start has

become a co-routine, which means that it now has the ability to be paused until a condition has been fulfilled.

In our case, it is necessary to declare this function as a co-routine because the code that we use to gather information from the PHP script will need to send a request to the server and then wait for the answer. However, we don't want the whole programme to stop while this data is on its way (e.g., we still need to update the screen and perform other important tasks). So in that sense, this function does not act like a usual function, in that it doesn't just perform actions and return to where it was called from; instead, because part of its tasks is to gather (and possibly wait for) information from the server, which may involve delays, as a co-routine, this function will fetch for the server's data and pause itself until the data has been received. Meanwhile, other functions (such as the Update function, for example) will be able to run in the meantime; then, when the data is received from the server, the Start function is called again just after the point where it had been paused.

We will see more about co-routines in the next chapters; however, in a nutshell: co-routines are a bit like a soccer team where the ball is passed to a player; this player takes the ball and starts to run, however, s/he needs information from the team doctor to know whether s/he can go ahead and what s/he can do to heal a recent injury; so s/he just passes the ball to another team mate and freezes (yield) until s/he receives instructions from the doctor. When s/he receives this instruction, the ball his passed to him/her again and s/he resumes to play. So in this case the players share the ball, but only one player has the ball at one particular time. So co-routines are a way to collaboratively run a program with only one function running at a time. If you have used threads, co-routines and threads differ in that threads run in parallel whereas co-routines work collaboratively to

freeze one function and give it the focus again when criteria have been fulfilled (only one co-routine running at any given time). Co-routines are usually referred as concurrency as they pass control to each-other.

We then declare an object of type WWW that will be used to run the PHP script using the url variable defined earlier.
We use the command yield to wait (or "freeze" this function) until the data has been returned by the server.
When the data has been received, it is saved as text, using the variable www.text and printed in the Console window.

When this is done, you can make sure that the MAMP server is running, add the script to the object accessDB (i.e., drag and play the scene, and check the Console window which should look like the following figure.

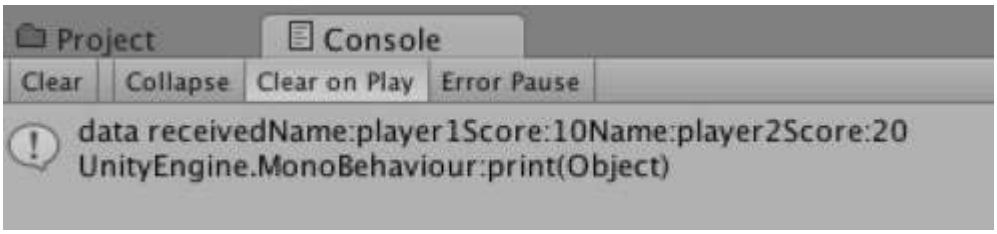


Figure 10-39: Receiving data from the server (through PHP)

So if you see this message, you have successfully managed to connect to the database and to read its content from Unity. Well done!

Now, while we have managed to collect this information, we could try to display it in a text field; this will mean that we need to create a field, write this confirmation to it, but also format our text, as for the time being, the text gathered from the PHP script may not be easily readable.

Please open the PHP script that you have created earlier.
Modify it as follows.
Change the code...

```
    echo "Name:".$name;
    echo "Score:".$score;
```

to...

```
    echo $name."\t";
    echo $score."\n";
```

In the previous code:

We have modified the line of code that displays the name by adding the character “\t” which is a tabulation.
We have modified the line of code that displays the score by adding the character “\n” which adds a line break.

Now that this change has been made, we can save the PHP script and focus on our C# script.

Please save the PHP script.
Switch back to Unity.
Create a new Text UI object | UI |
Rename it
Change its height to
Move it to the center of the screen using the Move tool (you can switch temporarily to the 2D mode, and then click on this object in the Hierarchy window).
Using the you can empty its text field.

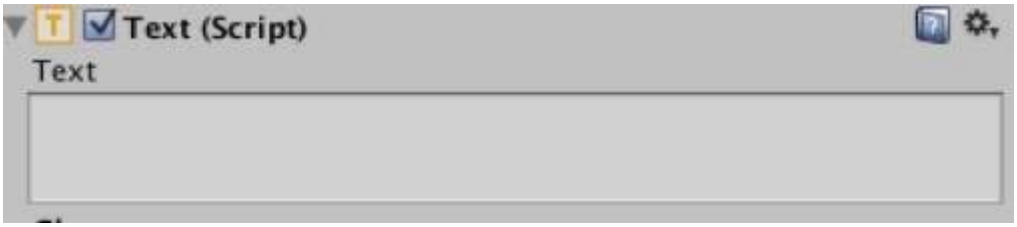


Figure 10-40: Emptying the text field

Modify the C# script as follows:

Add the following code at the beginning of the class.

```
using UnityEngine.UI;
```

Modify the code in the Start function as follows (new code in bold)

```
IEnumerator Start()
{
WWW www = new WWW(url);
yield return www;
string result = www.text;
print("data received"+result);
GameObject.Find ("high_scores").GetComponent ().text = result;
}
```

Please save the C# script, play the scene and check that the name of the players (and their scores are displayed in the text filed that you have created).

Now that we know that our PHP script works and that we can access the database through Unity, we will modify our code so that we can obtain two types of information: (1) whether a user is actually present in the database, and (2) the score for this player, if it already exists in the database.

Updating the player's records

In this section, we will create a simple interface that will be used to capture the player's name or nickname. Once this information is captured, we will access the database, and check if the user exists. If this is the case, then we will just update the database, otherwise, we will create a new user with the corresponding score. The score, for this example, will be hard-coded, for the time being.

Let's start by creating the interface.

You can switch to the 2D mode for now, if you wish, using the 2D button located in the top left corner of the Scene view.
Please create a new Input Field | UI | Input
This should create an object called Input Field within the Canvas object; within this object, you will also see two objects called Placeholder and as described in the next figure.

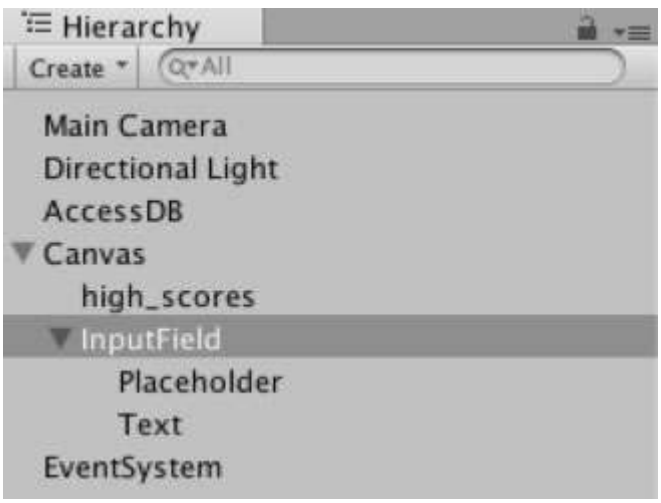


Figure 10-41: Creating the InputField object

Play your scene: you should be able to see the input field as described in the next figure.

Figure 10-42: Testing the Input Field object

At this stage we can type text in this field; however, we need to find a way to store this information when the user presses the “Enter” key or another key or button of our choice.

So, to be able to check whether the user has validated its entry (e.g., by pressing the “Enter” key), we will create a script that detects when this happens; this script will then store the value that has been entered in the text input field.

Please create a new C# script.

Name this script SaveScore for example (or any other name of your choice).
Add this script to the object
Open this script to edit it.

Add the following code at the top of the script, so that we can refer to UI object easily:

```
using UnityEngine.UI;
```

Please add the following code in the Start function:

```
gameObject.GetComponent().onEndEdit.AddListener(saveScore);
```

In the previous code, we do the following:

We access the InputFieldComponent for the object linked to this script (i.e.,
We then use the event which means that we will be tracking when the user has finished editing his/her text.
When this happens, we will call a function called saveScore (that we yet have to create).
So to summarize, we create an event listener for the event onEndEdit that will be linked to the function So this function will “handle” the event.

Last, we just need to create this function:

Please add the following code within the class (e.g., after the Update function).

```
void saveScore(string textInField)

{
print ("Starting to save score for user "+textInField);
}
```

Please save your code.

You can now test the scene by doing the following:

Play the scene.
Enter text in the text field.
Press the “Enter” key on your keyboard.
Look at the Inspector and see whether the text is displayed there.

So, if this is working, we know that we can enter text in (and gather text from) the input field; we just need to be able to send this text to a PHP script that will check whether this user exists.

Please duplicate the file updateScore and rename the duplicate updateScore_b (please keep the duplicate file in the same folder as the original PHP file).
Open this file (i.e.,
Comment or delete the following code.

```
/*$query = "SELECT * FROM high_scores";
$result= mysqli_query($con, $query);
$n = mysqli_num_rows($result);
for ($i = 0; $i < $n; $i++)
{
$name = mysqli_fetch_assoc($result)["name"];
$score = mysqli_fetch_assoc($result)["score"];
echo $name."\t";


echo $score."\n";
}*/
```

Add the following code at the end of the script (new code in bold) instead;

```
mysqli_select_db($con, $database) or die("Unable to connect to database");
$name = $_GET['name'];
echo "name: ".$name;
```

In the previous code:

As we have seen in the previous sections, the function $_GET[‘name’] will get (or obtain) the value for the variable called name that was passed as part of the url when the PHP page was called.
This variable is then displayed onscreen.

We can now check our code:

Save the PHP code and open your browser.
Make sure that M/WAMP is running and that the servers (i.e., Apache and MySQL) are active.

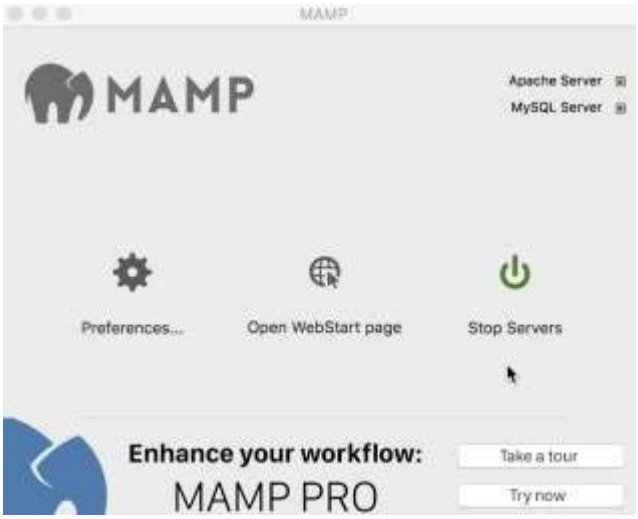


Figure 10-43: Checking that the servers are active

Open the url
The page should display the following text:

name: myname

So at this stage we know that we can successfully pass a variable (i.e., to the PHP page.

Now, the next step is to query the database for this particular name. In other words, we will try to access a record/row that corresponds to a user

with the name that we have stored earlier.

Please add the following code to the PHP script:

```
$query = "SELECT * FROM high_scores WHERE name = '$name'";


$result= mysqli_query($con, $query);
$n = mysqli_num_rows($result);
if ($n > 0) echo "Found 1 record"; else echo "Sorry not registered";
```

In the previous code:

We create a query statement.
For this query, we select all records from the table called and then focus on the record for which the name is the one stored in the variable $name (if you remember, the value was passed as part of the url). :-)

We can now test the code:

Please save your code.
Open this url in your browser:
http://localhost:8888/updateScore_b.php?name=test
The page should display:

name: Sorry not registered

The mention “Sorry not registered” is displayed because we have made a mistake on purpose by asking the server to display a record that does not exits (i.e., none of the records currently available in the database and table have a name called “test”.

Please open the url http://localhost:8888/updateScore_b.php?name=player1
The page should display:

name: player1Found 1 record
Did you get the same results??

Yes..?
Perfect!
So now we just need to write code that either update an existing player's record or adds a new player to the list.

Please add the following code (new code is bold) to the PHP script (i.e.,

```
echo "name: ".$name;
$score = 100;
$query = "SELECT * FROM high_scores WHERE name = '$name'";
$result= mysqli_query($con, $query);
$n = mysqli_num_rows($result);
if ($n > 0)
{
echo "Found 1 record";
$query = "UPDATE high_scores SET score ='$score' WHERE name = '$name'";
}
else
{
echo "Sorry not registered";
$query = "INSERT INTO high_scores VALUES ('$name','$score')";
}
$result= mysqli_query($con, $query);
```

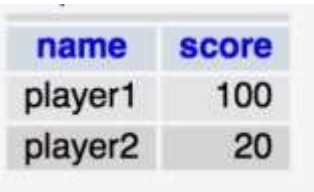
In the previous code:

We temporarily set the score to

If a record is found with the same name, then we can update the record accordingly with the new value for the score. Here we use the keyword UPDATE in the query.
If no corresponding record is found, we then create a new record (or row) in the table, by adding both the name and the corresponding score. Here we use the keyword INSERT in the query.

We can now check our code:

Please save your code.
Open the url http://localhost:8888/updateScore_b.php?name=player1
Check the database, using you should see that the score of player1 is now 100 (remember, this value of 100 was hard-coded in the PHP script).



name	score
player1	100
player2	20

Figure 10-44: Updating the score of an existing user

Open the url http://localhost:8888/updateScore_b.php?name=newUser

Check the database; you should see that a new player has been added

name	score
player1	100
player2	20
newUser	100

Figure 10-45: Adding a new user and a new score

One of the last thing we need to do is to pass not only the name of the player but also the score; for this, we will use the same techniques as previously using the statement

Please modify the PHP script as follows.
Replace this code....

```
$score = 100;
```

...with the following code, so that the score is now obtained from the url.

```
$score = $_GET['score'];
```

Please save your code.
Open the following url:
Check the database using you should see that the player newUser has an updated score of

So at this stage, we are almost ready to go; our PHP script works perfectly; we just need to connect it to Unity, so that whenever the name is entered in the Input field and the return key pressed, the PHP page is called using (or passing) the proper name and score.

Please switch to Unity.
Open the script
Modify the beginning of the class and the Start method as follows (new code in bold):

```
public class SaveScore : MonoBehaviour
{
string playerNname;
int score;
// Use this for initialization
void Start ()
{  gameObject.GetComponent().onEndEdit.AddListener(saveScore);
score = 1000;
}
```

Please modify the saveScore function as follows:

```
void saveScore(string textInField)
{
playerNname = textInField;
print ("Starting to save score for user "+textInField);
StartCoroutine(connectToPHP());
}
```

Then create a new function connectToPHP as follows:

```
IEnumerator connectToPHP()
{
string url = "http://localhost:8888/updateScore_b.php";
```

```
url += "?name=" + playerNname + "&score=" + score;

WWW www = new WWW(url);
yield return www;
print ("DB updated");
}
```

In the previous code:

We create a co-routine called
Within this function, we create a new url that includes the address of the PHP page that will connect to the database.
We then open this url with the www object and wait (using the yield command) until we receive data.
Once this is done, we display the message “DB

Once this is done, we can test our code:

Please save your code, and play the scene.
Enter a name in the text field.

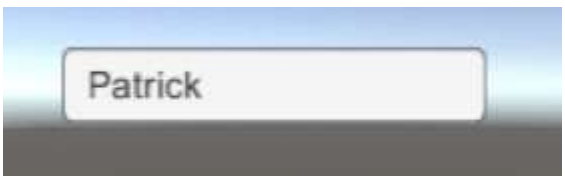


Figure 10-46: Entering a name in the text field

Press return.
Using please check that the dataset now includes a new player with a score of
In Unity, please empty the text field to enter the name then press return and check that the database has been updated accordingly using

Please note that if you want to use a connection to a real server, you will probably need to set what is called a cross-domain policy. So you will need to create new file called crossfomain.xml and then add it to your remote server.

The structure of this file may look as follows:

version="1.0"?>

domain="*"/>

For more information on cross-domain, see the following page.

https://docs.unity3d.com/Manual/SecuritySandbox.html

Level Roundup

Well, this is it!

In this chapter, we have learned about communicating with a database, setting up a server, creating a database and tables, and finally reading from, writing to or updating a database through Unity. In the process we were also introduced to PHP and MYSQL commands, and we finally managed to create a system whereby the player's score is saved in this database. So yes, we have made some considerable progress, and we have by now looked at some simple ways to store and access information about the game and the players on a server.

Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist
Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist Checklist
Checklist Checklist Checklist Checklist Checklist Checklist Checklist

Checklist Checklist Checklist Checklist Checklist Checklist

Quiz

It’s now time to check your knowledge with a quiz. So please try to answer the following questions (or specify whether the statements are correct or incorrect). The solutions are included at the end of the book.

The keyword when added just before the name of a function, means that it is a co-routine.
To be able to access a database a user with proper privileges needs to be set-up.
Crossdomain restrictions apply if you are trying to access a database on your local server.
The keyword SELECT can be used in SQL Queries to select specific rows.
The keyword mysqli_queries can be used to perform SQL queries through PHP.
The keyword IEnumerator when added before the name of a function indicates that it is a co-routine.
The keyword localhost can be use to refere to the server where the PHP pages are hosted.
Data can be passed to a PHP script using its url.
The PHP function $_GET can be used to receive (or get) data from a MYSQL database.

Challenge 1

Now that you have managed to complete this chapter and that you have improved your skills, let’s modify the scripts to add more interaction.

Add a new column to the database that you have already created to save a number (i.e., int) that will represent the last level achieved by the player. In Unity, create a splash-screen, along with three other scenes.

In the splash screen, ask the user for its name, and either load the first scene if s/he is not registered in the database, or load the corresponding last level achieved stored in the database for this user..

11

Reading Files and Creating Scenes Procedurally with C#

In this section, we will learn how to create your game levels from scripts and external files rather than by adding all objects manually to each scene. This will have the advantage of saving you a lot of time creating your levels; it will also make it easier to modify your levels relatively quickly too. Creating your environment from a script or "procedurally" can be achieved using a wide range of techniques from simple arrays to XML files, or prefabs. So, after completing this chapter, you will be able to:

Instantiate objects based on an array or a text file.
Create a level from an array.
Create multiple levels using simple text files.
Create more complex scenes by reading and implementing the content of an XML file.

Some of the skills you will also learn along the way include:

Creating and accessing a Resources folder in your project where you can store and access resources for your game (e.g., images or text files).
Reading text files from your game.
Reading and parsing an XML document.

Building your environment from an array

The first and simplest way to create a game environment procedurally is by using a simple array, so to setup our first procedural environment, we

will generate an indoor level using a combination of C# scripting and arrays.

We will proceed as follows:

Create an array that represents the environment.
Read the array.
Instantiate objects based on the numbers read in the array.

So let’s get started:

Please create a new scene | New and rename it gameLevelAuto (or any other name of your choice).
Create a new cube Object | 3D Object | and rename it
Make sure that the scale property for this object is (100, 1, 100) and that its position is (0, 0,
You can also create a material for (and apply it to) the ground object if you wish. To do so, you can either import a texture and apply it to the object, or create a new color material through the Project window (i.e., Create |

Next, we will create an object that will be used to instantiate the walls.

Please create a new cube Object | 3D Object |
Rename it
Set its scale property to (10, 2,
You can also create and apply a blue Material to this object or use any other texture of your choice.

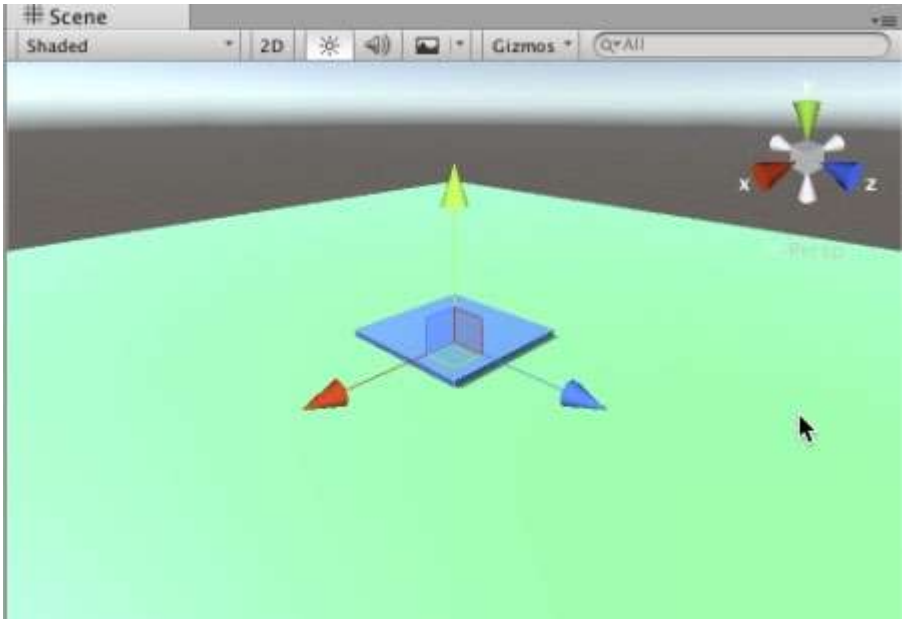


Figure 11-1: Creating a wall

Once this is done, you can create a prefab from this object, rename this prefab

To create the prefab, you can either drag and drop the object wall to the Project window, or create a new prefab from the Project window | and then drag and drop the object wall from the Hierarchy to this prefab.

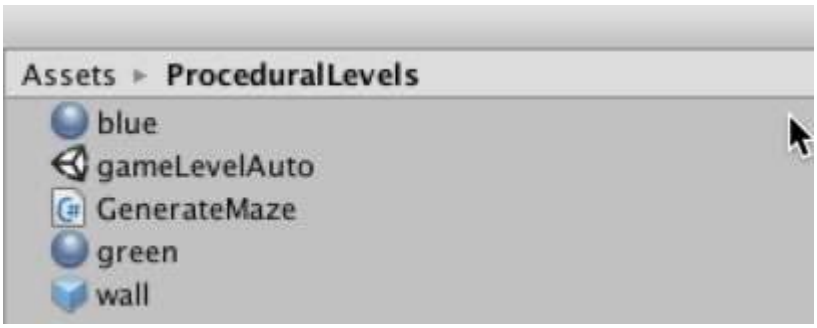


Figure 11-2: Creating a new wall prefab

Once this is done, you can deactivate the object wall located in the Hierarchy window, using the

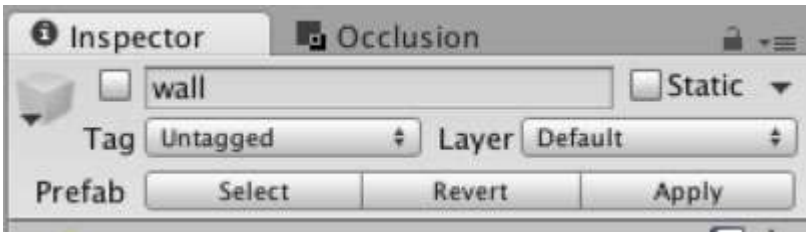


Figure 11-3: Deactivating the wall from the Hierarchy window

Next, we will create a script that will generate our maze.

Please create a new C# script (i.e., from the project window select: Create | C# and rename it

Open this script and add the following code at the beginning of the class (i.e., just before the Start method).

```
public GameObject wall;
private int [,] worldMap = new int [,]
{
{1,1,1,1,1,1,1,1,1,1},
{1,0,1,0,0,0,0,0,0,1},
{1,0,1,0,1,0,1,0,0,1},
{1,0,1,0,0,0,0,0,0,1},
{1,0,1,1,1,1,0,0,0,1},
{1,0,0,0,0,0,0,0,0,1},
{1,0,1,0,1,0,1,1,1,1},
{1,0,0,1,0,0,0,0,0,1},
{1,0,1,0,0,0,0,0,0,1},
{1,1,1,1,1,1,1,1,1,1},
};
```

In the previous code:

We declare a public GameObject variable that will be used a placeholder in the Inspector window to set the object to instantiate with the

corresponding prefab.
We then declare a multi-dimensional array (i.e., a two-dimensional array) of integers. The structure of this array mirrors the structure of the maze that we would like to create; for example, the top row could be the north wall, and the bottom row could represent the south wall, etc. So each 1 represents a wall, and each 0 represents an empty space.
Each row of the array is defined using opening and closing brackets with values within separated by commas.
Please add the following code to the Start method:

```
int i,j;
for (i = 0; i < 10; i++)
{
for (j = 0; j < 10; j++)
{
GameObject t;
if (worldMap [i,j] == 1) t = (GameObject)(Instantiate (wall, new Vector3 (50-i*10, 1.5f, 50-j*10), Quaternion.identity));
}
}
```

In the previous code:

We declare two integers i and these will refer to specific rows and columns in our array. For example, if i is 1 and j is we will be looking at row 1 and column Because each array starts at 0, these will effectively be the second row and the second column in our array.
We then create two loops; these loops will go through each row of the array that we have created.
We then check the value of each element present in the array.
If the value is we instantiate a wall prefab accordingly.

Now, we just need to finish our setup:

Please save your script.
Check that it is error-free in the Console window.
Create an empty object and rename it
Drag and drop the script GenerateMaze to the object
Once this is done, select the object
Make sure that the Inspector window is active.
In the Inspector window, identify the parameter called wall for the script GenerateMaze attached to this object, and drag and drop the prefab wall from the Project window to this variable.

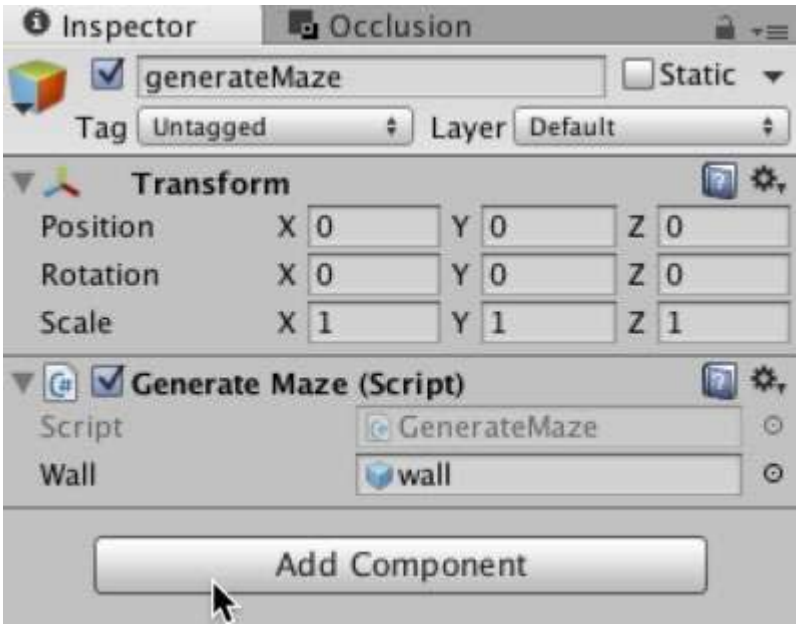


Figure 11-4: Using the wall prefab for the variable wall

So that we can see the layout in the Game view, you can change the camera position to (0, 90, 0) and its rotation to (90, 0,
Once this is done, you can play the scene, and check the layout either from the Scene view or the Game view.

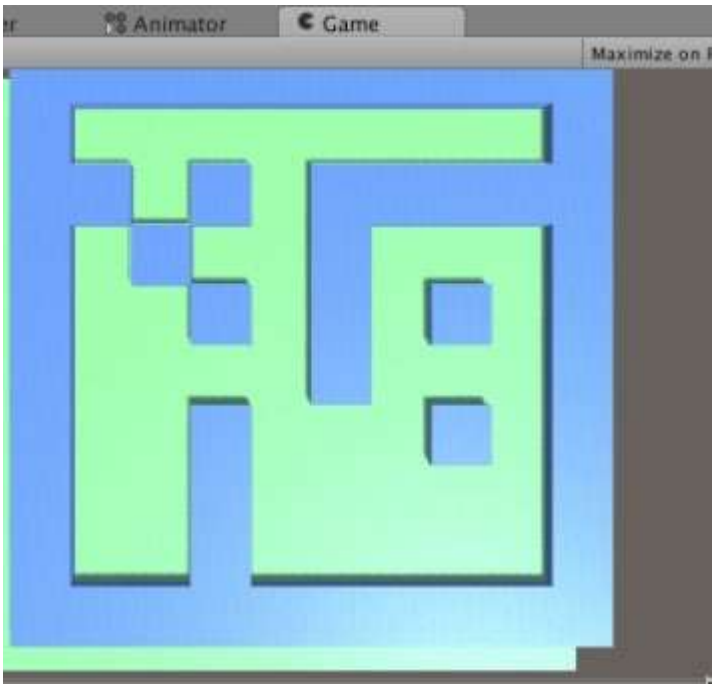


Figure 11-5: The new scene viewed from the main camera

Creating an environment from a text file

Now, this works well and we could take it a notch further by creating a file that includes all the information about the maze.

You see, using arrays is great; however, it may be more convenient to use a specific file for each level. This will at least do two things for you: (1) it will make it possible to modify the structure of the level without having to modify the code, and (2) it will make it possible to create (and load) individual files for each level.

So you could virtually create a text file for every level and then load it accordingly.

So, for this purpose, we will be using a new method called This method makes it possible to load resources (e.g., textures or text) from your Unity project. So first, we will create such a text file and then access it through our script.

Please create a new folder called Resources within the folder called Assets in your project (from the Project window, select: Create |

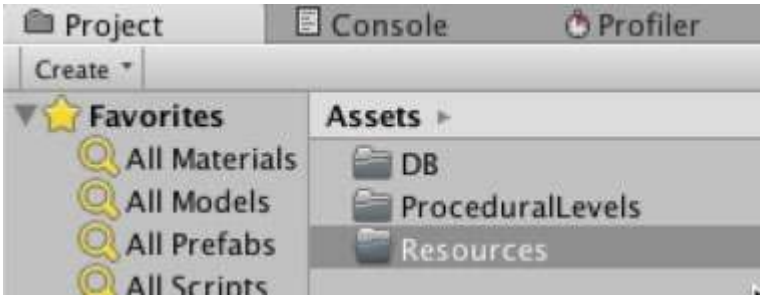


Figure 11-6: Adding a new Resources folder

Create a new text file using the editor of your choice.
Add the following text to it.

1111111111
1010000001
1010101001
1010000001
1011110001
1000000001
1010101111
1001000001
1010000001
1111111111

Save it as, for example, maze.txt within your Resources folder.

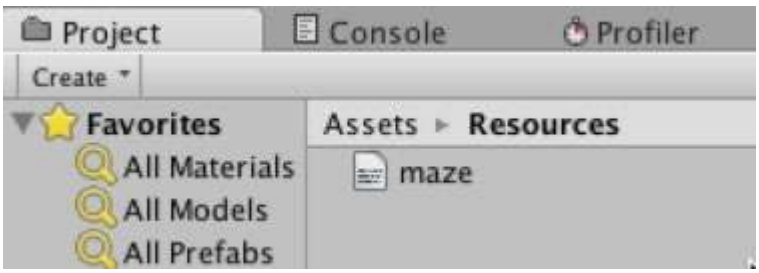


Figure 11-7: Saving the file maze.txt in the Resources folder

If you don’t know where you Resources folder is on your file system, you can locate it by right-clicking on it in Unity and selecting the option Reveal in Finder (for Mac OS) or Show in Explorer for Windows computers. You can also save your file on your computer system and then drag and drop it to the Resources folder.

Figure 11-8: Locating the Resources folder in your file system

Once this is done, we can modify our C# script so that it reads information from the text file rather than the array.

Please open the script
Comment (or remove) all the code already present in the Start method.
Please add the following code to the Start function instead.

```
void Start ()
{
TextAsset t1 = (TextAsset)Resources.Load("maze",
typeof(TextAsset));
string s = t1.text;
int i;
s = s.Replace("\n","");
for (i = 0; i < s.Length; i++)
{
if (s [i] == '1')
{
int column, row;
```

```
    column = i%10;
    row = i / 10;
    GameObject t;
    t = (GameObject)(Instantiate (wall, new Vector3 (50 - column * 10,
1.5f, 50 - row * 10), Quaternion.identity));
    }
    }
    }
```

In the previous code we do the following:

We declare a variable of type TextAsset that will be used to store the content of the file we have created as a resource.
We then access its content and save it to a string variable.
When this is done, we replace all the of characters (i.e., by empty strings; these (the end of line characters) were initially present in our text file for convenience and to better tell each row apart; however, we don't need this information anymore.

We then loop through the content of the string gathered from the text file, and we instantiate an object whenever the number 1 is read. Columns and rows for our maze are also identified by either dividing the counter (i.e., by 10 or by using the modulo operator (i.e.,

The modulo operator provides the remainder of a division.

We then do as previously to locate and rotate the object that has been instantiated.

Once you have added this code, please save your script and run the scene; you should see that the scene has been generated as previously, but

with the difference that the content is now read from a text file that is saved within your project.

Following this principle, you could have several files that correspond to each level, each with a different name, and you could then load these depending on the level to be displayed.

Creating an environment from an image file

While the previous techniques are quite interesting and useful, there is another way to create your level, using a more artistic approach; that is: by drawing the outline of your levels as an image, and then by reading this file and instantiating objects, based on the color of each pixel present in the image.

Let’s see how this can be done:

Please create a simple jpeg image, using an image editor of your choice, for example Adobe Photoshop, Microsoft paint or Gimp.
As you create your image, you can use the brush tool, and ensure that its size is 1 so that you can draw pixel-by-pixel.

For this particular application, we will leave empty spaces white, and any other pixel painted using the color of your choice.

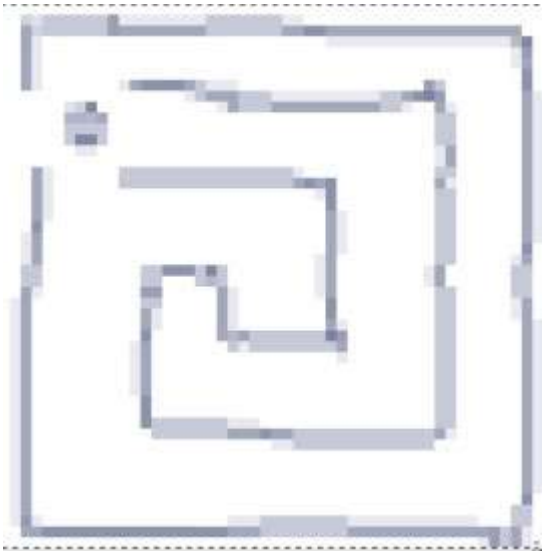


Figure 11-9: Creating the outline of the level.

The previous figure is my outline; again, it is very simple for the time being; it uses white pixels for empty areas and colored pixels for walls. This image is 500 by 500 pixels, simulating an area that is 500 meters wide and 500 meters long. If you wish, you can find and use this outline from the resource pack (i.e.,
You can save this image to any format of your choice; in my case, I have saved it to .png

Once this is done, we will need to import this texture in Unity and make sure that it can be read from our code.

Please import the texture in Unity (i.e., drag and drop the image to the Project window or use the option Assets | Import New
The asset can be saved in the Assets folder and does not need to be saved in the Resources folder.

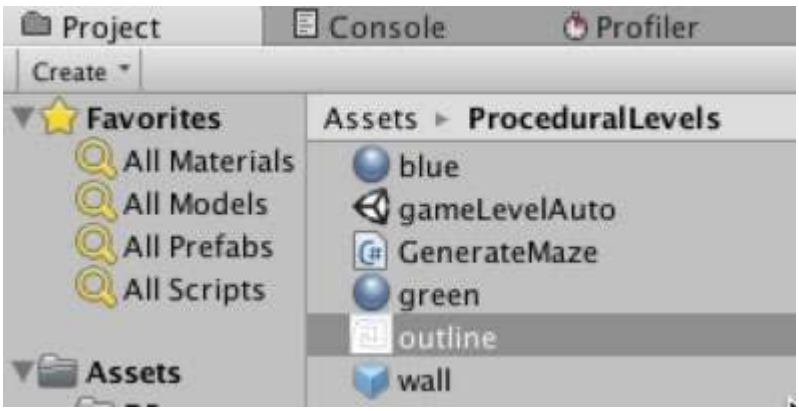


Figure 11-10: Importing the outline

Select the texture that you have imported in the Project window.
Using the Inspector window, set its Texture Type attribute to Advanced, as described in the next figure.

Figure 11-11: Setting the Texture Type attribute

Then select the option Read/Write Enabled to true (i.e., check the corresponding box).

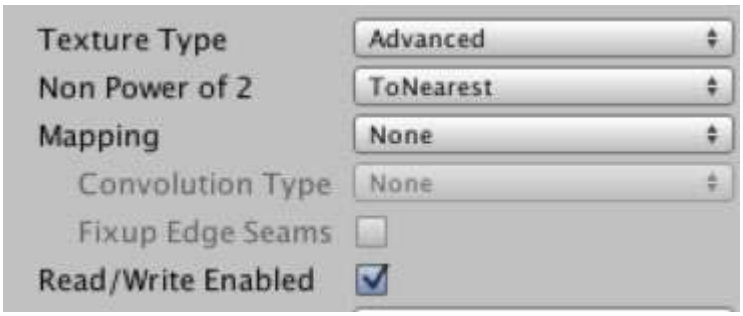


Figure 11-12: Making the image readable (from the code)

Press the button Apply that is located at the bottom right corner of the Inspector window.

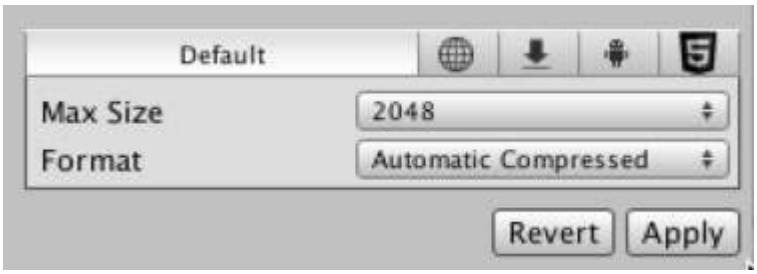


Figure 11-13: Applying the changes to the texture

At this stage, we have an image that is ready to be used and read; so all we need to do is to write the code that will create the environment based on this image.

Please create a new C# script and call it GenerateFromImage (or any other name of your choice)
Please add the following code at the start of the class.

```
Color[,] colorOfPixel;
```

```
public GameObject wall;
public Texture2D outlineImage;
```

In the previous code:

We declare an array of colors; this array will store the color of each pixel present in the image, so that it can be used for the outline of our maze.
The object called wall will be used as a placeholder (in the Inspector window) to drag and drop the wall prefab from the Project window.
The Texture2D outlineImage will be used as a placeholder (in the to drag and drop the outline image from the Project window (i.e., the image that we have imported).
Please modify the Start function as follows:

```
void Start()
{
colorOfPixel = new Color[outlineImage.width, outlineImage.height];
for (int x = 0; x < outlineImage.width; x++)
{
for (int y = 0; y < outlineImage.height; y++)
{
colorOfPixel[x, y] = outlineImage.GetPixel(x, y);//check tramsparency
if (colorOfPixel[x, y] != Color.white)
{
GameObject t = (GameObject)(Instantiate (wall, new Vector3
((outlineImage.width/2 * 10) - x*10, 1.5f, (outlineImage.height/2 * 10)-
y*10), Quaternion.identity));
}
}
}
}
```

In the previous code:

We initialize the array it is a multidimensional array; its size corresponds to the size of the image imported (i.e., width and height).
We then read the image, one pixel at a time, using two loops.
For each pixel present in the image and determined by its x and y coordinates, we save its color using the method GetPixel to the array that we have defined earlier.
If the color of this pixel is not white, then an object is created accordingly, as we have done in the previous sections.

Once the code has been saved; please check that it is error-free; you can then do the following:

Deactivate the script GenerateMaze that is currently attached to the empty object
Attach the script that we have just created to this empty object.

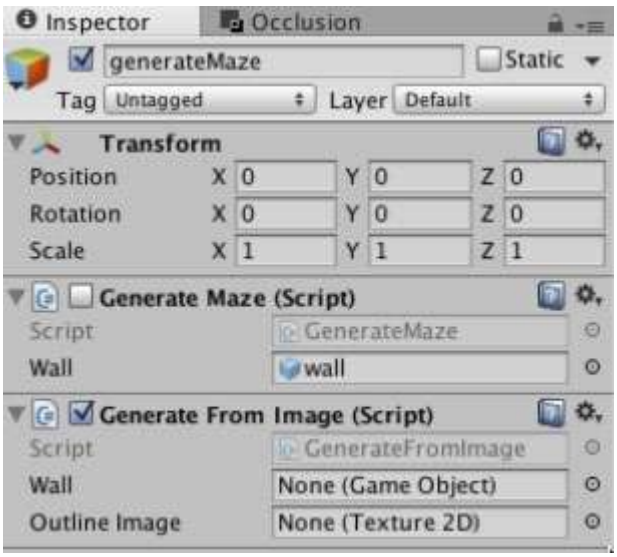


Figure 11-14: Adding a new script to the object generateMaze

Once this is done, you may notice that the fields wall and outlineImage from the script are empty; so we can set them by dragging and dropping the prefab wall to the variable and the outline texture to the variable

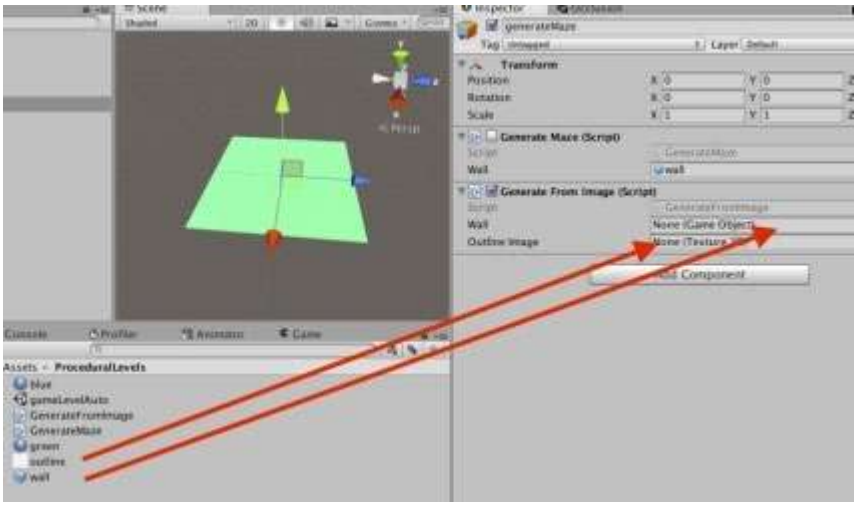


Figure 11-15: Adding the wall prefab and the outline texture (part 1)

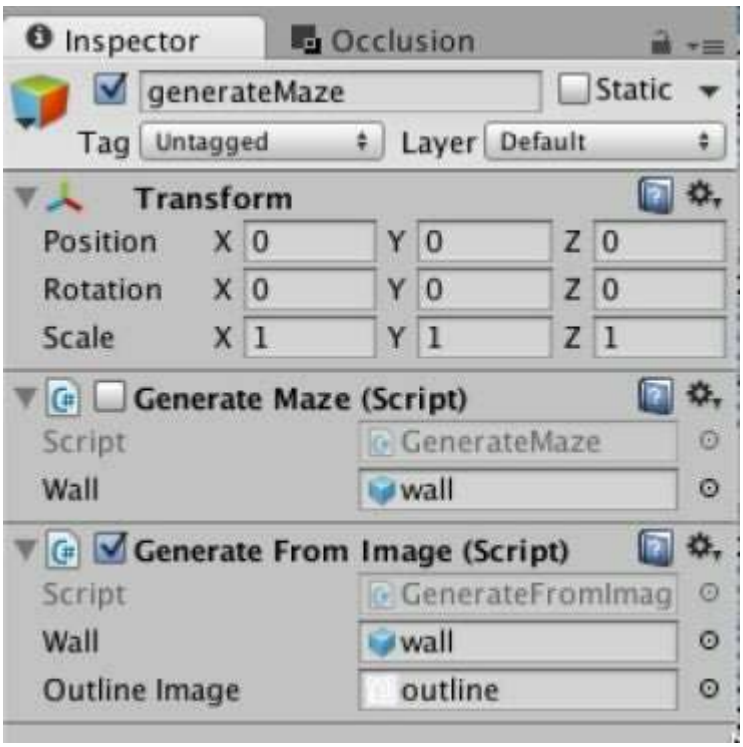


Figure 11-16: Adding the wall prefab and the outline texture (part 2)

Last but not least; since our image is 500 pixels by 500 pixels, the ground will need to be adjusted; if your image has a different size, you can adjust the dimension of the ground accordingly.

Please change the scale property of the ground to (700, 1, You may also adjust the position of the Main Camera to (0, 611,
Once this is done, you can play the scene, and look at it from the Scene view.

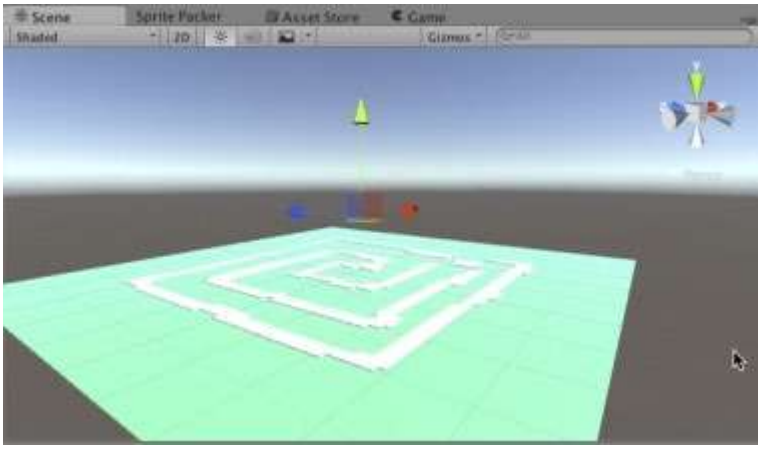


Figure 11-17: Displaying the maze generated from the image

So this seems to work :-)

For this technique and all the other procedural generation techniques covered in this chapter, you can of course navigate the scene by adding a First-Person This, for example, can be done when you edit the scene by doing the following:

Import the Character package | Import Package |

Drag and drop the prefab FPSController to the Scene view from the folder Assets | Standard Assets | Characters | FirstPersonCharacter |
Change the position of the FPSController to, for example, (0, 5, so that it is above the ground.

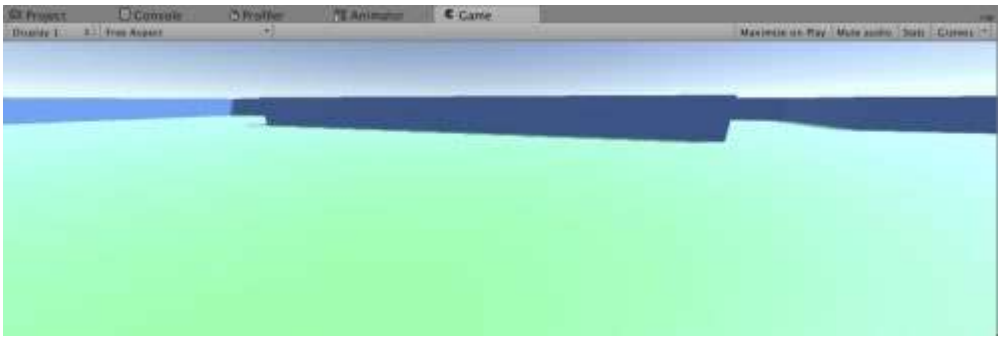


Figure 11-18: Navigating the scene with a First-Person Controller

Now, while this works, there are many ways in which we could optimize this script; we could for example, include specific prefabs when a color is found in the file. For example, we could use red for doors, and blue for walls with different colors, on so on.

Using XML files for content creation

Now, using text files to generate an environment at run time is great; however, this could be limited if we wanted to have more control over the properties of the objects to be instantiated; for example, we could use numbers in our file, and employ 3 for a tree and 2 for a wall; but what if we would like to instantiate different types of walls with each a different texture, size, or rotation. You can see here that a text file, while useful, may be limited; instead, we could use something a bit more advanced such as XML files; an XML file will make it possible to provide more information about the object (or nodes) that we want to create in our game. The nice thing about this feature is that it can also be used for complex visualization techniques to visualize data that is stored in the XML format, including data on weather, financial transactions, scientific measurements, or news information. Many of these XML files are freely available and can be accessed and processed. So, for example, you could create a 3D environment that simulates scientific data such as temperatures, or streams in oceans, and so forth; and since these files are usually updated on a regular basis, you could provide a 3D application that simulates real life phenomena.

Now, before you can move to these complex applications, we will just create a simple application that reads data from an XML file and that uses it to configure the game; we will then create another simple file to add objects to be included to the levels, along with settings their properties, all of this from an XML file.

So first, what is an XML file?

XML stands for It was originally designed to store data in a way that could be read by both humans and computers. As you will see in the next code examples, these files use the extension .xml and have a common structure that makes them easy to read.

Let's look at an XML file that we could create to describe a scene; it could look like the following.

Please note that, given that you follow simple XML rules, you can create and set a structure of your choice for your XML file, which makes them very versatile.

version="1.0" encoding="UTF-8"?>

number ="1">

number ="2">

So, an XML file includes a succession of nested elements delimited by their tags. For example, in the previous code, the game elements start with and ends with Each element also includes attributes. For example, in the previous example, we have levels nested within each game elements; for each of these levels, a number is defined. Similarly, for each object within a level, attributes such as location or scale are defined also.

The first line of the XML file is optional, but its good practice to add it. It basically mentions that the version 1.0 of XML is used and that the encoding used is UTF8 (i.e., this is the default encoding for XML).

When creating XML documents, there are other rules that must be followed, including:

Each element is delimited by and includes an opening and a closing tag. As you can see, there is a tag along with a tag. The back slash "/" marks a closing tag for an element.

These tags are case-sensitive; so using followed by is incorrect; but using followed by is proper.
All elements need to be nested properly. For example, the following nesting is correct (e.g., the first tag open is the last tag closed):

While the next nesting is not correct:

Elements can have attributes and these must be defined using quotes.

The beauty of this file format is that you can create your own XML files using a structure of your choice to best reflect and serve the requirements of your game or application; you could save information about each scene, about the NPCs (e.g., what paths they can use), or the weapons. So yes, you could virtually save any type of information with these files, using an easy-to-read format.

Now that the XML format is clearer, let's see how we can read an XML file to create a simple scene.

In the next section we will do this as follows:

Load an XML file.
Open this document.
Navigate through each level node or element.
For each of these nodes, create the corresponding game objects defined for this scene.

First let’s look at the XML file that we will be using:
version="1.0" encoding="UTF-8"?>

number ="1">

number ="2">

As you can see in the previous code, the file includes the following:

A first line with information on the version of XML used along with the encoding type.
We then define the root note called game (i.e., the node that contains everything else in the XML file).
For this game element (or node), we have two direct children elements called
For each level element (or node), we have defined an attribute called we also added an object to each level, each with specific attributes called rotation and The idea here is to define where these objects will be within each scene, and to also define their location and appearance.

For each open tag, we also create a closing tag (e.g., and , and or