

Tu Thanh Nguyen

INTRODUCTION TO 2D GAME DEVELOPMENT WITH UNITY

Bachelor's thesis

Information Technology

2021



South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree title	Time
Tu Thanh Nguyen	Bachelor of IT	May 2021
Thesis title		52 pages 4 pages of appendices
2D Game Development in C# with Unity		
Commissioned by		
Supervisor		
Timo Hynninen		
Abstract		
<p>The aim of this thesis is to introduce other developers and programmers to game development with Unity. It requires prior knowledge of C# programming. I introduce them to the game engine Unity, one of the most popular game engines nowadays, and guide them through the process of making a 2D game.</p> <p>The game in the thesis is a 2D-platformer game called "Escape The Cave". Through the game, I introduce some of the functions and components that Unity has to offer and help them get used to the interface of Unity.</p> <p>At the end of my thesis, readers now can create their own 2D game through what I have showed them using Unity and Visual Studio.</p>		
Keywords		
2D Game Development, Unity, Visual Studio, C# programming language, 2D platformer		

CONTENTS

INTRODUCTION	5
1 INTRODUCTION TO GAMES	6
1.1 A brief history of computer games	6
1.2 2D games and 3D games	7
2 GAME ENGINES	8
2.1 Popular game engines	9
2.1.1 Unreal Engine	9
2.1.2 Godot	10
2.1.3 Unity	10
3 PROGRAMMING LANGUAGE C#	11
4 UNITY ENGINE	13
4.1 User interface (UI)	13
4.2 Basic Unity	17
5 ESCAPE THE CAVE	19
5.1 What is Escape The Cave?	19
5.2 Environment	20
5.3 Main Menu Scenes	26
5.4 Character creation	28
5.5 Introduction to game physics	33
5.6 C# Scripts in Unity	36
5.6.1 Player.cs	37
5.6.2 LevelFinish.cs	43
5.6.3 MainMenu.cs	45
5.7 Level Design	46

5.8 Cinemachine.....48

5.9 Scene Management.....49

6 CONCLUSION.....50

REFERENCES52

APPENDICES

Appendix 1. Player.cs script

Appendix 2. LevelFinish.cs script

Appendix 3. MainMenu.cs script

INTRODUCTION

Gaming is becoming more and more popular over the decades and the game business is booming. People play games to relax, to be better at competitive games or simply just to kill time. Therefore, the game industry has become one of the most sizeable (and interesting) businesses in the world. New games are introduced to players every day by indie game companies and corporations, and news about game release dates pop up in every social media and advertisements.

As a result, more programmers and developers are taking an interest in game development and design, but not everyone knows where to start as the skills needed are vast and varied. Most find game development intimidating because of the number of skills and knowledge needed to make a game: programming, animation, sound design, environment art... this is the fundamental knowledge to make a game. Not everyone has all these skills on their hands, and that is why triple-A games are developed by big teams (or game studios) such as Ubisoft, Naughty Dog, Square Enix, CD Projekt Red... but that does not mean an individual cannot make good games by themselves. There are many indie game developers that create astonishing games like Minecraft by Markus "Notch" Persson, Stardew Valley by Eric Barone, Undertale by Toby Fox, ...

Thus, in my thesis project, I want to help newcomers to game development start their first game as I guide them step by step. I will introduce the basic skills of game development with the programming language C# and one of the most widely used game engine out there: Unity. Since this is an introduction to game development, I only cover a 2D game as it is the basic. My goal is that by the end of my thesis, the programmers and developers get a fundamental knowledge on game development and can start to make games independently.

Through my thesis, I introduce thoroughly each of the components I mention above (C#, Unity, 2D games...). During the introduction of each component, I give examples so that readers can easily understand and relate since not every newcomer is acquainted with the words or expressions.

I will also explain what a game engine is, why we need game engine to create games... and introduce readers to Unity Game Engine.

The most challenging component to understand is the programming language C# because it is the backbone of what this thesis is about. Thus, I will go in depth with C# section so that after studying, readers can understand and know how to code in C#

1 INTRODUCTION TO GAMES

1.1 A brief history of computer games

Alan Turing – the one that inspired modern computer science and artificial intelligence – was the first person to make a computer game, a computer Chess program in 1947. It was not his purpose to make it playable for people but to test out artificial intelligence. It was not until 1952 that he tested the program with a colleague, and he pretended to be the computer (Tristan Donovan, 2010). Turing's first game inspired so many mathematicians and computer scientists back in those days that they decided to continue his work.

Later in 1972, Pong was released into the arcade. It became so popular back then that the arcade machine was jammed because it had so many arcade coins in it.

July 1980, a missing slice of pizza inspired Toru Iwatani to create Pac-Man, the first arcade game hit that people can play at home with their console.

Fast-forward to early 21st century, we have news channels about games, streaming services for gamers, online game stores, dedicated merchandises based on popular games, with many companies dedicated to making games like Electronic Arts, Ubisoft, CD Project Red... With all that listed above, it is without a doubt that game industry has become one of the fastest growing and most profitable industries in the world.

1.2 2D games and 3D games

The first generation of computer games was 2-dimensional (2D) since technology back then was limited for 3-dimensional (3D) simulation. But nowadays, the growth of technology allows us to make 3D simulations with the right tools. 3D printing is now possible, 3D simulations allows us to experience things that are normally out of our reach or too risky to try (learn to drive, learn to fly a plane, simulate the solar system,...). So, what are 2D and 3D games?

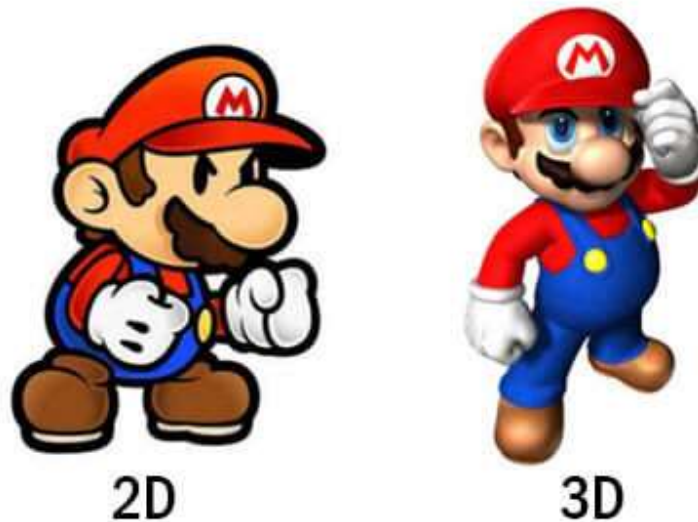


Figure 1. 2D and 3D comparison.

In 2D games, the world inside it is only two-dimensional, which means that it only consists of 2 axes: X and Y. 2D games use flat graphics, called sprites, and don't have three-dimensional geometry. They're drawn to the screen as flat images, and the camera (orthographic camera) has no perspective (Unity, 2021). normally in 2D games, the player can only move left and right, up and down, according to the X-Y axis since the environment in 2D games is visually flat. Some of the popular 2D games examples: Super Mario Bros, Pac-Man, Donkey Kong, ... The limitation on movement however does not mean that 2D games are less enjoyable than 3D games, this depends on the players' preference. Moreover, for players who prefer casual gaming and lightweight games, 2D games is the more suitable choice because 3D games nowadays are often more demanding and take more space on the hard drive.

3D games, however, offer a more immersive experience. 3D games usually make use of three-dimensional geometry, with Materials and Textures rendered on the surface of GameObjects to make them appear as solid environments, characters and objects that make up the game world (Unity 2021). In a 3D game, the player can move and see its world in 3D, which means it is a simulation of the real world. Objects in a 3D environment have depth and volume because the game world operates on an X-Y-Z axis. Players can move up and down, left and right, forward and backward. 3D games have more possibilities and goals than 2D games.

In 2D games, the character only moves in two dimensions, which is easy to control. While 3D games make it more challenging for the players because they have to get used to the 3D movement of the game, for example WASD keys to move, space to jump and mouse to look around.

The reason why I want to write for 2D game development only is because it is the most basic and easy to learn. After learning to make 2D games, readers have the knowledge and foundation in making games so that it is an easy transition to 3D game development should they choose to.

2 GAME ENGINES

A game engine is a software designed specifically to make video games. Developers use game engine to create games for different platforms, such as computer, console, and mobile phone. “It exists to abstract the (sometime platform-dependent) details of doing common game-related tasks, like rendering, physics, and input, so that developers (artists, designers, scripters and, yes, even other programmers) can focus on the details that make their games unique” (Jeff Ward, 2008). The main components of a game engine are renderer for 2D and 3D design, physics engine, scripting function, sound add-ons and animation tools. Many game engines provide reusable components such as character models, basic sounds, template worlds. For this reason, a game engine is powerful and convenient for designing almost anything, not just games. It can be

used to design software (because basically, a video game is a software that is entertaining to interact with), make 2D and 3D animation, some can even be used to make music.

Then why the need for other design tools for animation and music? This is because those tools are made professionally and specifically for those functions. For example, a music maker tool does not need any animation components and drawing software does not necessarily need sound design. It would be easier for professionals to use the tool designed for one's purpose. Thus, game design software, or game engine, is made for game development elements.

2.1 Popular game engines

There are many game engines out on the market, but I will name some of the most widely known so readers can get the concept of different game engines and their purposes.

2.1.1 Unreal Engine



Figure 3. Unreal Engine Logo.

Many triple-A games are developed with Unreal Engine due to its robust graphical capabilities in lighting, shader, etc. The engine is also open-source, so it is being improved constantly by its community. It is the most suitable for making 3D and VR games thanks to its graphic options. Because of its heavy graphical components, the engine requires a more powerful computer to run compared to Unity. Also, many developers declare that Unreal Engine is more suitable for larger projects in which you intend to work as a team.

Popular games: Ark: Survival Evolved, Final Fantasy VII Remake, Gears 5...

2.1.2 Godot

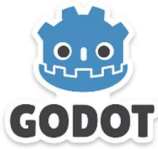


Figure 4. Godot Logo.

The Godot game engine is rather new, it became available in 2014. Only until recently it became popular. It is free and open-source, very suitable for beginner developers. It is suitable for 2D and 3D game development but is lightweight. In comparison with Unity or Unreal Engine, the quality of product is not as good. It does not have the graphical power of Unreal Engine, nor the support community of Unity, but it is easy to learn to use if you want to start your game development journey. In terms of platforms, the majority of games made with Godot are mobile games.

Popular games: Sigil Breakers, Kip, The Kingdom of Avalae...

2.1.3 Unity



Figure 2. Unity Logo.

Unity has been a popular choice for indie game developers around the world since 2005. It is suitable not only for 2D and 3D game development but also great for virtual-reality (VR) and augmented-reality (AR) design. The engine is updated every year with new contents and thanks to a sizeable community, it has an asset store with a huge amount of free and pay-to-use assets. Unity is free so everyone can start using and learning how to make games easily.

Popular games: Cuphead, Inside, Ori and the Blind Forest...

The reason why I chose Unity Engine in my thesis is because it is suitable for making games on your own. Normally, a team is required to make a game because there are so many components of a game that one person cannot handle all of them or take a long time to master all the necessary skills (animation, sound, environment design, coding skills...). Unity provides some components for free so we can focus on programming.

3 PROGRAMMING LANGUAGE C#

C# (pronounced *C-sharp*), inspired by many features of F#, is a programming language that first came to the public in the year 2000. It is a general-purpose language designed for developing apps on Windows and it requires the .NET Framework to work. The language can be used to program almost anything. But it is good for developing Windows applications and games. Some web developers also choose C# to make a web application and it is also popular for mobile development too.

C# is easy to learn, relatively easy to read, but because of its flexibility, it is hard to master all that C# has to offer. C# is a complex language, mastering it may take more time than a simpler language such as Python or HTML, users need to learn a considerable amount of code to create an advanced program.

But it is worth the time users put in to learn it. C# is an in-demand skill that many tech-savvy companies are looking for. In the job market, there are applications every day and everywhere in need of developers with good C# knowledge and experience. Therefore, people who learn C# have great job opportunities.

I chose C# because of its flexibility. Readers can learn C# not only to make games, but also to have better opportunity in their journey to become a developer of any aspects of the IT world.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Unity Script | 0 references
public class Player : MonoBehaviour
{
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        ...
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        ...
    }
}

```

Figure 5. Unity C# Script example.

Above is an example of C# script. The structure of C# is easy to understand: we have a class, and inside it there are methods or functions on what this class should be able to do. A class is like a blueprint for an object. In real life, an object has shape, color, and functions. It is the same with a class in C#. Inside this class, we create methods, declare values and properties. Therefore, a class defines the kind of functionality and data that its objects have.

Inside the example “Player” class, we have methods “Start” and “Update”. These methods already have their defined functions described above them in the green comment lines. We can create new methods and give them names fitted for their functions (PlayerMove, PlayerDie, PlayerIdle...). Names for methods should not have blank space in between. Inside the method, we start with open curly bracket “{” and stop with close curly bracket “}”.

Comments in C# do not have any functions on the code. We use comments simply to explain our code and make it more readable. A single line comment starts with two forward slashes “//”.

The three lines of “using” is for namespace. It is a way for us to save time on writing the namespace every time we need to use a method within that namespace. For instance, normally we have to write `System.Console.WriteLine(“Hello World!”);` to print the phrase “Hello World!”, but

with `using System;` namespace, we can write `Console.WriteLine("Hello World!");` to abbreviate.

These are the absolute basics of C# use in Unity. To fully comprehend C# would take more time and words than I can write in my thesis. Therefore, readers can refer to the [link](#) in reference section for a more comprehensive understanding of C# in Unity. I will go deeper into C# later in chapter 5 where we make our first game.

4 UNITY ENGINE

Unity is a powerful tool for game programming. It has an asset store where we can get game-ready assets and use it for our game. Unity is good for both working alone and in a team. In this chapter, we go in depth with Unity Engine and introduce some of its features.

4.1 User interface (UI)

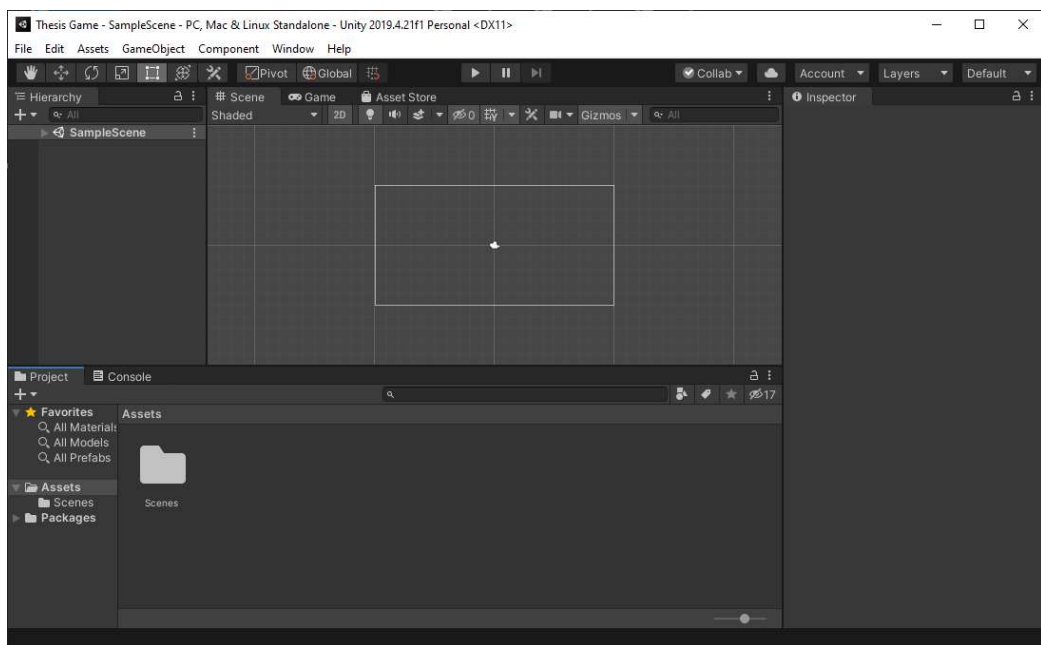


Figure 6. Unity default user interface.

Unity user interface can be daunting for beginners, it might take some time to get used to because of its complexity. In this part, we walk through the layout of the

interface. We can customize the UI windows layout in any way we see fit for our project. I introduce the default layout because it is the layout we see when creating a new project.

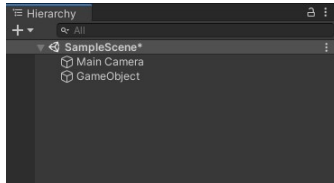


Figure 7. Hierarchy window.

On the top left corner, we can see the Hierarchy window. It displays every game object we have in a scene. Whatever we add or remove in a scene, we also add or remove in the hierarchy window.

Game objects in Unity are fundamental objects that represent characters, properties, and sceneries. They are containers for the components to make sense of the game. An empty game object does not do anything, but if we attach a component to it, it will have function. For example, if we add a collision component to an empty game object, it can now collide physically with other game objects that also have collision component.

A scene contains the objects of our game. Think of each scene as a unique level of the game. We can also create the main menu where the player can start the game, customize settings and quit the game within a scene. Many scenes combine with each other to make a game.

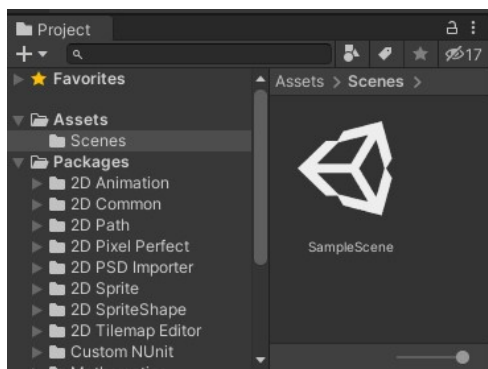


Figure 8. Project window.

Below the Hierarchy, we have the Project window. It not only shows all the files of the project we are working on, it is also the main way to navigate and create new files for our project.

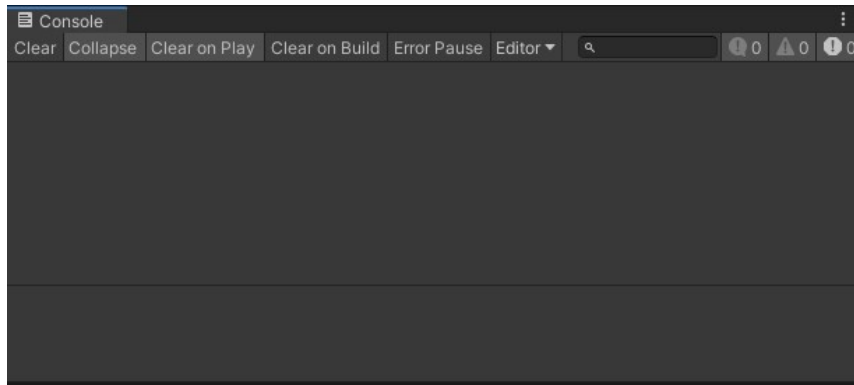


Figure 9. Console window.

Next to the Project window, we have Console window. This window is one of the most important ones we should have on our screen, it shows the warnings, error and other messages related to our project. These messages are vital for our game to work without any problems. It is also the quickest way to find bugs in our project.



Figure 10. Inspector window of Main Camera object.

On the right side of the UI, we can see the Inspector window. This window shows the components within our currently selected game object. We can also add more or remove components, customize the components as we see fit for our project.

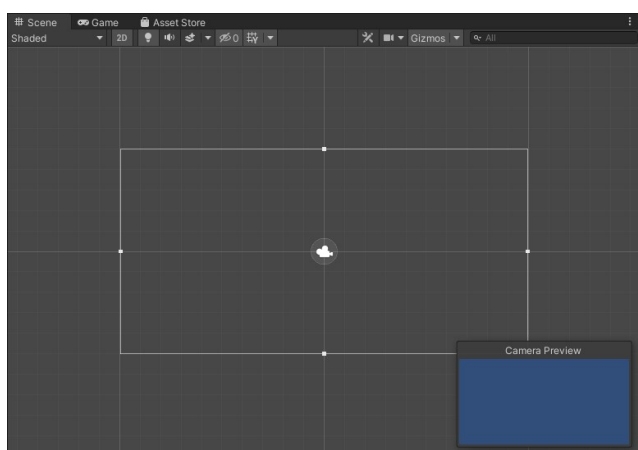


Figure 11. Scene window.

In the middle of the UI, we can see the Scene window. This is the interactive view of the world we are creating within our scene. We can interact with components within our scene, here namely scenery, characters, platforms, environments, and other types of game objects.



Figure 12. Game window.

Next to Scene window, we have Game window. The game window visualizes what the player sees when playing the game. We mostly cannot interact with anything within the game object because its only function is to visualize the game.

4.2 Basic Unity

In this part, we go through some basic features we can do with Unity. I only introduce the features that we use regularly in the next chapter. There are also many ways to do each of the steps I introduce here, so readers can explore the UI and find their own preferences.

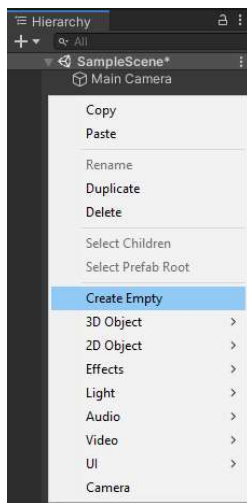


Figure 13. Create an empty GameObject.

To create an empty GameObject, simply right click on the Hierarchy and choose “Create Empty”. As mentioned before, an empty GameObject does not have any function yet, we must add components to it. We can see from the picture that there are also options to create 3D Object, 2D Object, UI objects, ... These are also game objects, but it already has components added to themselves to function. For example, if we click on “2D” and choose “Sprite”, we create a GameObject named “New Sprite” with component “Sprite Renderer” already added to it.

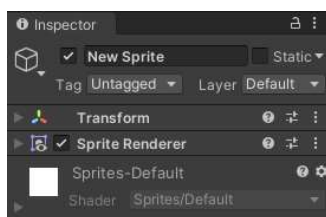


Figure 14. New Sprite GameObject.

To add a component to a GameObject, simply select the GameObject in the Hierarchy window, then in the Inspector window, we can click “Add Component” and search for the component we need.

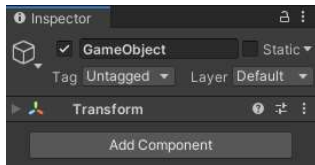


Figure 15. Add Component to a GameObject.

We will create many files in our project, so it is essential that we organize the files into folders. People have different preferences on how they organize their projects, but the way that I found effective is to organize by file types.

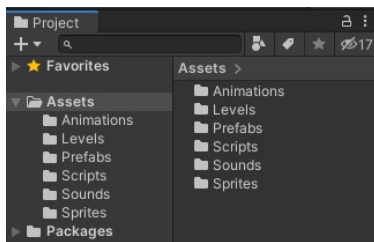


Figure 16. Organize project.

As an example, the way I organize on the picture above is by file types. For example, in the “Sound” folder, I put all the sound files like background music, sound effects. I can also have subfolders for different type of sounds if I have many sound files (character voices, boss music, ambient world sounds...). this way, I can keep track of what I have created and where I can find them.



Figure 17. Tool bar




The tool bar above is what we use to configure our game object in the Scene Window:

- Hand Tool: use this tool to pan around the scene.
- Move Tool: move the selected GameObject.
- Rotate Tool: rotate the selected GameObject.
- Scale Tool: rescale the GameObject on all axes at once.
- Rect Tool: scale, position, size, and anchor for rectangle GameObject.

Those are the basic features of Unity. Of course, there are many more features and even more ways to do things with Unity, but they exist for our preferences on how we want to make our game. For the next chapter, I will introduce more features of Unity because it would be easier to explain them along with making our first 2D game.

5 ESCAPE THE CAVE

This chapter is where we get our hands on making an actual 2D game. After this chapter, we will have the skills and knowledge to make an actual working game and begin our journey into game development. I will walk us through step by step so that you can follow along as you read. We will make a 2D platformer game called “Escape the Cave”. For this to work, we need the latest version of Unity and Unity Hub. Unity Hub is where we can manage our Unity projects and the Unity version we are using. To download the latest version of Unity and Unity Hub, go to <https://unity3d.com/get-unity/download>. For this game, I am using Unity version 2020.3.1f1.

To create a new project, open Unity Hub, in the  Projects tab, click on  button. From there, we can choose the project template that suits our need, in this case, 2D. Then name your project and choose its location on the computer. I name mine “Escape The Cave” and its location is “D:\Repos\Unity2D” on my computer. It is a good practice to put all our projects under the same folder so that it is easy for us to manage when we have many projects later. After that, click on  to create the project. After a while, Unity finishes creating our project and we are greeted with the UI.

5.1 What is Escape The Cave?

Before we dive into developing our game, I want us to think about how we will make the game: what is the theme, what do we want the player to experience, how do we make our game stand out among other games, what kind of game we make, and even more questions that you can think of. Asking these questions ourselves not only helps us shape the game in our minds but also makes us focus on making the game the way we meant for it to be. In my experience, when

making a new game, I tend to create more feature for it than I originally planned. This is not a bad habit, the game might turn out to be better, but sometimes try to remember to ask yourself “Is this what I want the player to experience?” For example, we are making a game about exploration and puzzle-solving, but when creating a level, we make too many obstacles, put in too many puzzles to solve or the puzzle is too hard. This is when we need to look back and ask ourselves “what about the exploration aspect of the game?”

After having the game shaped in your mind, write it down so that you will not forget. For example, in “Escape The Cave”, our character wakes up underground and has no idea where he is, but he knows that he has to get out of this cave first. So now that we know that he is in a cave underground, we would want our level to be vertical. It is a 2D platformer so think of games like Super Mario, Donkey Kong... but instead of going from left to right, we try to lead the character to go up. There should be puzzles and challenges on our levels to keep our game interesting.

In summary, our game is defined:

- Player experience: feeling of mystery
- Core mechanic: 2D platformer
- Theme: escape game

As we have defined our game, we are ready to make our game!

5.2 Environment

One of the first steps to making our game is creating the environment. We need an asset for this to be easy. Normally when making a game, we need a team that each person has a different set of skills: programming, animation, character design, sound design, ... But we are now able to make game on our own thanks to game-ready assets. Unity Asset Store provides us with many free assets that we can import directly to our project such as animation, environment art, sound effects, texture design, ... Of course, there are many more places where we can find good assets, but for this project we use “Sunny Land Forest” 2D animation assets from Unity Asset Store. You can get it free here:

<https://assetstore.unity.com/packages/2d/characters/sunny-land-forest-108124>

After downloading it, we can import the asset to our project. Simply navigate to Window > Packet Manager, then select “Sunny Land Forest” asset, and click “Import”.

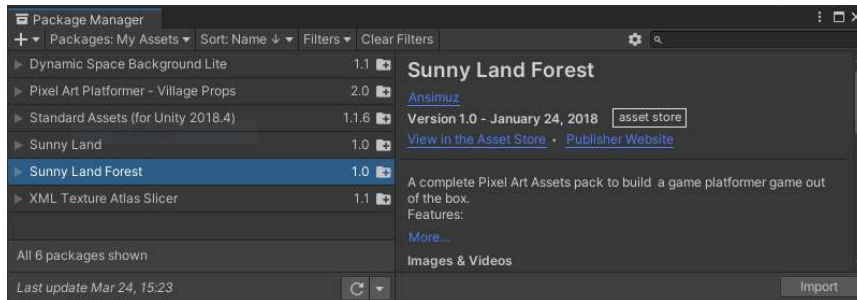


Figure 18. Packet Manager Window.

After a while, we can see that the asset is imported under our “Asset” folder.

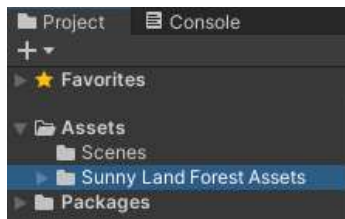


Figure 19. Imported Sunny Land Forest Assets.

Navigate to Sunny Land Forest Assets > Artwork > Environment, we can see the file named “tileset”. We will use this file to draw our environment in Tilemap.

Tilemap is a component that we use to visualize 2D scenes (or levels). It handles and stores tile assets when we use Tile Palette to draw upon it. It also takes care of the relationships between the components such as Tilemap Renderer or Tilemap Collider 2D. In the latest version of Unity, the Tilemap package is already included, however, if it is not included in your Unity, you can always get it from Package Manager.

We will be using square-shaped tiles, so to create a Tilemap in our scene, go to GameObject > 2D Object > Tilemap > Rectangular. This creates a Grid GameObject, with a Tilemap as a child GameObject. With Grid selected, it shows rectangular grid in our scene.

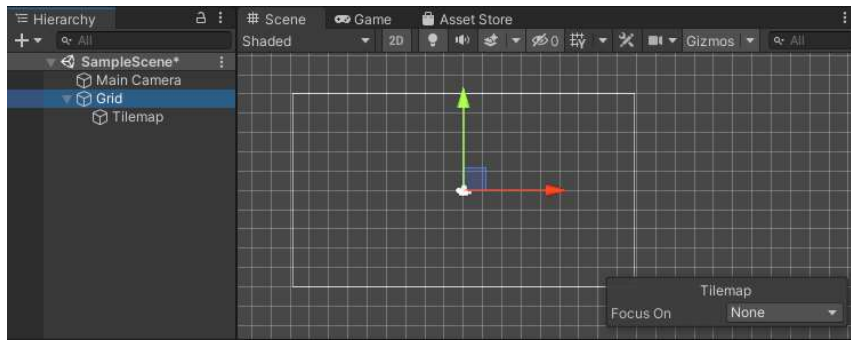


Figure 20. Grid showing in Scene.

Let us change the Tilemap name to “Background”. Then duplicate Background by pressing Ctrl+D. Name the second Background as “Foreground”. After this step, your Hierarchy should look like this:

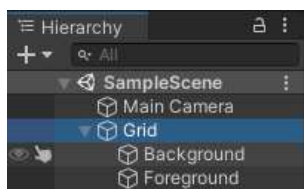


Figure 21. Background and Foreground tilemaps.

The reason why we need to separate background and foreground is that later in the project, we will create collision between the foreground and the player. Because the background only serves as visualization, player cannot interact with the background layer.

Next, to draw on the tilemaps, we need the Tile Palette. Simply navigate to Window > 2D > Tile Palette. This opens the palette that we use to draw in our tilemaps. You should see a Tile Palette window like this:

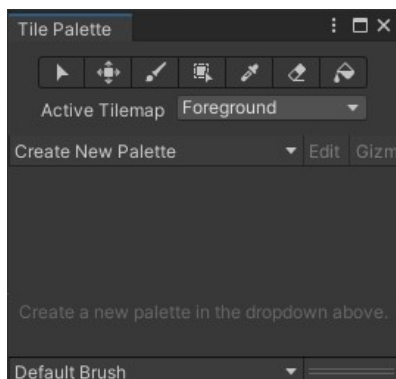


Figure 22. Tile palette.

Click on “Create New Palette” and name it “Main Palette” and place it under a new folder called “Palette and Tiles”. Locate the “tileset” file in Sunny Land Forest Assets. This file is a “sprite”, a 2D graphic objects used for environments, characters, and other elements of 2D game. A sprite can be sliced into many pieces for display purposes. To convert “tileset” sprite into a usable palette, simply drag and drop “tileset” into Tile Palette window with the “Main Palette” on. Then put it in the “Palette and Tiles” folder we just created. We now have the palette to draw our environment.

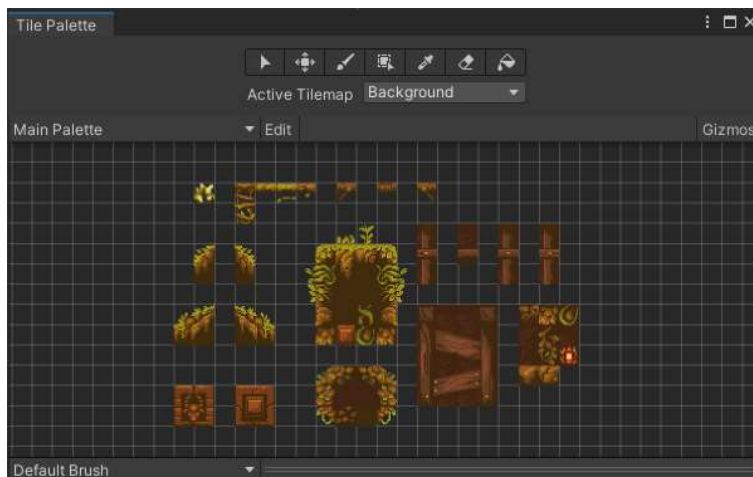


Figure 23. Main Palette ready to use.

Get familiar with the palette and try to draw on your scene, be creative with your design. Then we can use the “Eraser” to erase your drawing. While drawing, keep in mind which layer we are drawing it on. If we accidentally draw a background element onto our foreground, it can be hard to detect and cause problem in our physics later, such as our character suddenly being stopped by an invisible wall of background pieces blended in our foreground. With that in mind, lets draw our background first.

Again, use your imagination on how the background should look like. In the definition of “Escape The Cave”, our character is in a cave and has to get to the surface, since it is usually dark in a cave, our background should have a dark color. Here is an example of what the background should look like.



Figure 24. Background.

The foreground should be things that players can interact with, so it should have brighter color than the background. Select the foreground to start drawing. At this point you might encounter a problem that when you draw it does not appear on the screen. This does not mean the things you draw are not on the screen, but it has rendering problem. If we look at the inspector of both background and foreground, the “Tilemap Renderer” component has “Sorting Layer” setting, and in both grounds, they are rendered at default layer.



Figure 25. Sorting Layer at Default.

To fix this, simply add more sorting layers for background and foreground. Remember to place “Background” layer above “Foreground” because in Unity algorithm, it renders the layer list from the top down, which means layers at the top are rendered first. This makes the “Background” layer visualized behind “Foreground” layer.



Figure 26. Layers' placement.

After assigning the appropriate layers to background and foreground, we can start to draw our foreground. As mentioned before, the foreground is our interactable environment, which means we must assign physics and interactable components to the foreground. This way, when we assign the character in our scene, they can stand on the foreground and interact with the interactable game objects we draw in our scene. Before drawing, remember to select the foreground in the Hierarchy to draw on it, or else we might be drawing on the background Tilemap. After finishing, our game scene should look like this:

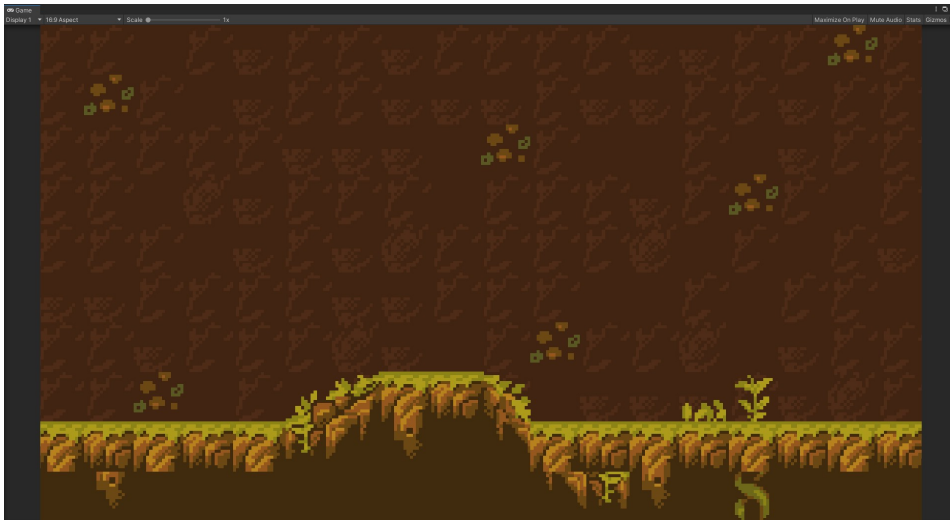


Figure 27. Foreground Example.

This is the end for the environment. As you can see, I make minimal work on the scene and only introduce the fundamentals of environment creation. This is because my game is only a tutorial, I leave the imagination and expansion of the scene to you. You can make the scene larger than mine based on the fundamentals that I introduced in this part and create an extensive level that satisfies your imagination. There is also another concept of layer that we should know about, the physical layer. Newcomers often get confused about this part because the description of the layer types are vague. As can be seen from the picture below, there are “Sorting Layers” and “Layers”.

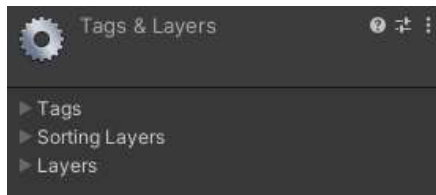


Figure 28. Tags & Layers window.

While we know the “Sorting Layers” function is to render the layers, we have no idea what “Layers” does. “Layers” is for the physical attributes of game objects. We will use the physical layers in later chapter.

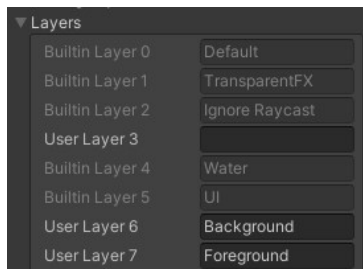


Figure 29. Physical layers.

For now, let us create the same layers as the sorting layer and add them to their corresponding Tilemap.

5.3 Main Menu Scenes

After we are done with the environment, we need to make the “Main Menu” scene and “Congratulation” scene. The “Main Menu” is where the players can choose to start the game or quit, we do not want to force the player to play the game without a way to quit the game. After they finish the game, we greet them with “Congratulation” scene to congratulate them on finishing the game and ask them if they want to play again or quit, which is basically similar to “Main Menu” scene. With that, create a new scene by right click on the project window, Create > Scene and name it “Main Menu”.

In the main menu, we should add the name of the game, a “Start Game” and a “Quit Game” button. To add text into our scene, go to GameObject > UI > Text. This adds a canvas and inside it, a text object. It also creates an “EventSystem”

game object which handles the input and sending events. Change the text name to “Title”, and inside the Inspector, we have the text component where we write the name of our game “Escape The Cave”. Inside this component, there are many customizations for the text object (font, size, color,...), so customize your title however you like.

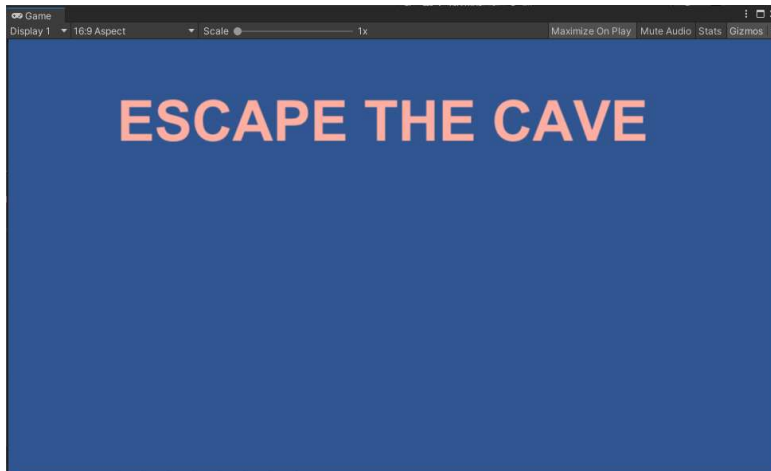


Figure 30. Title “Escape The Cave” Example.

Next, we add the buttons the same way we add text, `GameObject > UI > Button`. Inside the buttons, there is a text object that defines the text showing on the button, name each one “Start Game” and “Quit Game”. there are also many customizations for the text and the button inside the button object, edit your buttons to your preference. The “Main Menu” Scene should look similar to this.

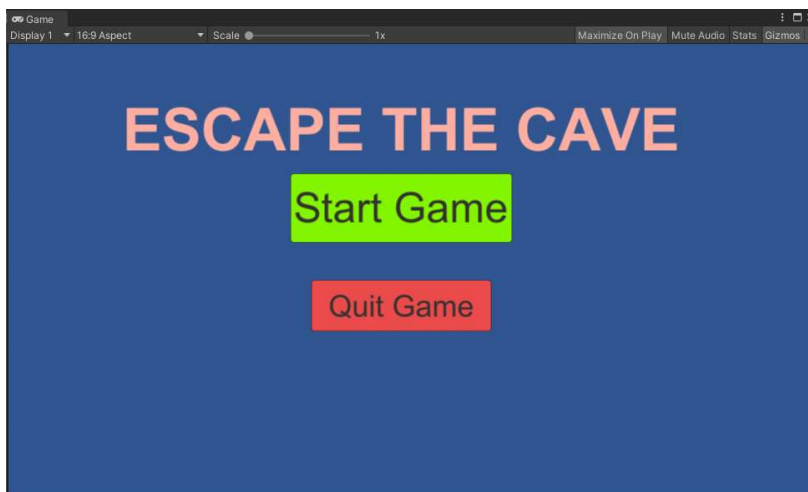


Figure 31. Example of Main Menu.

Duplicate the “Main Menu” scene and name it “Congratulation”. This scene is similar to main menu, but we put this at the end of our game, so there should be a congratulation to our players who finish the game, and the “Start Game” button should be “Play Again?”.



Figure 32. Example of “Congratulation” scene.

Later in *C#* scripts section, we will create functions for these buttons. For now, let us move on to character creation.

5.4 Character creation

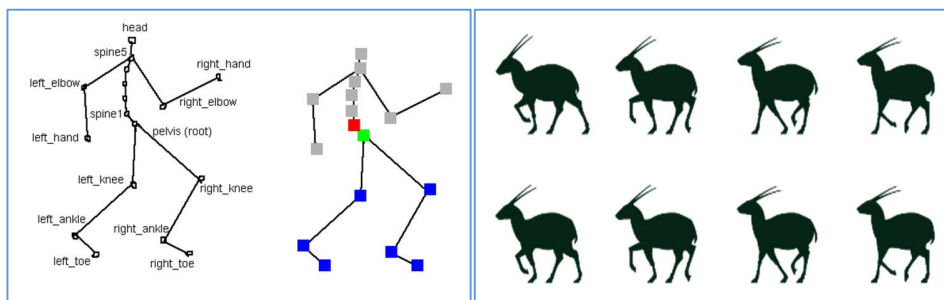


Figure 33. Comparison between Skeletal Animation and Sprite Sheet Animation.

The character is the player’s way to interact with the game world that we created. Players control the character’s actions through their input on the keyboards, game pads, controllers, phones, etc. In this part, we create the animations for our player character and a way for player to control the character.

There are two ways to create character animations:

- Skeletal Animation: the animation is made up of several parts connected to each other. The animator programs how each part moves and relates to each other. The reason why it is called skeletal animation is because it is similar to bone structure, each joint is connected to one another to create a whole skeleton.
- Sprite sheet animation: a sprite sheet consists of many frames of an object with each frame has little movement so that when combining them together, it creates the illusion of movement. This technique of animation is similar to making cartoons where many frames changing fast creates the movement of the cartoon character.

In this project, we use sprite sheet animation as our method because it is easier to implement than skeletal animation. Also, in the “Sunny Land Forest” assets, we have a character sprite sheet provided to us. Navigate to Sunny Land Forest > Sprite >, we can see that there are many folders containing animations for the character’s actions like climb, jump, fall, etc. We can try to drag and drop one of the animations into our scene, but the sprite does not show up. This is where we must track back to our foreground and background problem. The reason why our character does not show on the scene is because of the rendering layers. Our character is now on the default layer, which is rendered first and is behind our background and foreground. To make our character visible, we can simply render it with the same layer as our foreground. But to be more scalable, we can create another layer called “Player”, and move the layer in front of the “Foreground” layer. This is easier for us later in our game development journey as it helps us organize the layer and keep track of what is being rendered on which layer. To understand how animation in Unity works, we must first understand what Animation, Animator Controller and Animator are:

- An animation is a file that comprises of many sprites rendered sequentially so that it creates an illusion of movement.
- An animator controller is a file that controls the logics that trigger the animations, animation transitions and the states of the game object attached to it.
- An animator is a component that links a GameObject with its animator controller.


We can start creating our character animation with the knowledge given above. In the “player” folder, we have many folders for different character movement. I take “player-jump” folder as an example. We need sprites to put in our animation. To view the sprites, simply click the  button in the image and it shows us the sprites. We need to select the sprite in each image, then right click > Create > Animation. We have to do the same with other player animation folders, and then move all the new animations to a new folder named “Animations”. After this step, your “Animations” folder should look like this.



Figure 34. Animations Folder.

Next, we create an animator controller named “Player” in “Animations” folder. If we double click on the newly created animator controller, it pops up an “Animator” window. This is where we will add and work with our animations. But how do we link these animations to our player character? First, we create an empty game object named “Player” in the Hierarchy. We need to add “Sprite Renderer” and “Animator” components to it. Then, we simply drag and drop “Player” animator controller in the “Controller” field of the “Animator” component. Now we can play around with our character animations. To work with animations, we need “Animator” and “Animation” windows open. Navigate to Window > Animation, choose to open both “Animation” and “Animator” windows. To prevent any confusion between the two windows, the “Animation” window takes care of editing our animation files, while “Animator” window oversees when to play those animations.

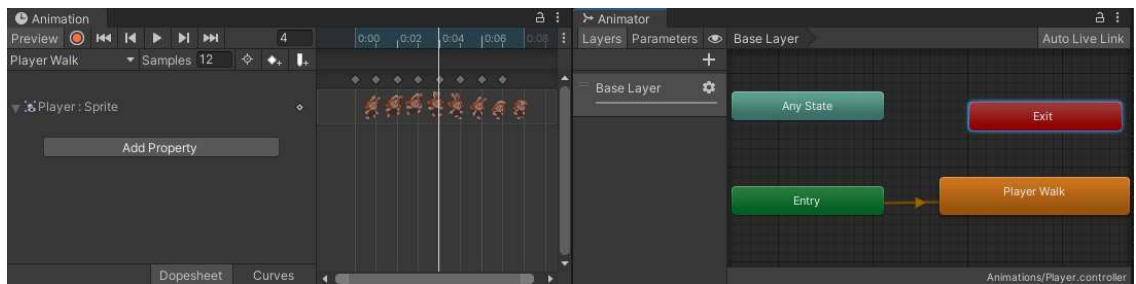


Figure 35. Animation and Animator windows.

As we can see in the “Animator” window, there are states for our character. The states dictate which animations should be played and when. There are three default states in the Animator: “Any State”, “Entry” and “Exit”. These states simplify our states map by removing the need for transition in some states. For example, if a state is connected to “Any State”, it can transit from any state in the states map and vice versa. To create connections between the states, right click on the first state we want to connect, select “Make Transition”, then choose the state that needs connecting, and we have an arrow that connects two states.

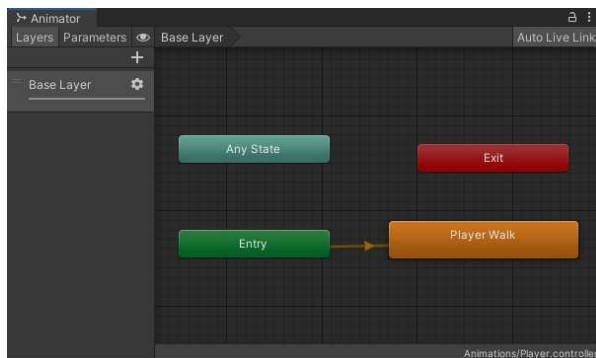


Figure 36. Transition between states.

We can now test our animations on “Player” game object. Remember to select the “Player” game object and try to play some animation that you added to the “Animator” window. The character in your scene should have some movement when you try some animations. At this point we have a basic knowledge of how animation works in Unity, the differences between skeletal animation and sprite sheet animation, how to convert sprite sheet into animation and states transition in animator.

As we have grasped the basic of how animations in Unity works, let us create the transitions between animations. We need to define which animation is for which action, and the transition from that animation to another. For instance, if a character is standing still, we would want it to have an idle state with idle animation, and when the character moves, the animation immediately changes to walking animation. To make it easy for you to follow, I have created my states and transitions in the picture below for reference.

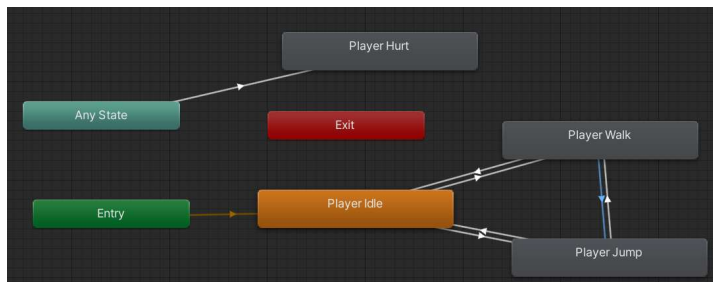


Figure 37. Player states map.

We also need to set the condition to trigger the states because the states do not know when to transit to another state. To set the condition, go to the “Parameters” tab of the “Animator” window, click on **+** button to drop down the list of parameters, as we can see there are 4 parameters we can choose from: float, int, bool, trigger. We choose “Bool” for our parameters. For example, the transition from “Player Idle” to “Player Walk”, we create a bool parameter and call it “isWalking”, then add this parameter to the transition and set its value to “true”.

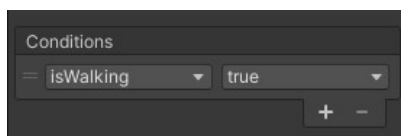


Figure 38. Condition for Player Idle > Player Walk transition.

In the opposite direction, we add this parameter as condition but set its value to “false”, so that when the character stops moving and is standing still, the animation goes back to idle. We do the same steps for our other, create bool “isJumping” and add it to suitable transitions. Later in this chapter, I will explain how to trigger these transitions in C# scripts.

5.5 Introduction to game physics

In this chapter, we learn how to apply physics into our environment and character. Right now, what we have on the screen is like a painting, there is no interaction between objects yet, nothing for the character to touch, jump on or even stand on. This is where physics definition takes place.

Physics in game is very similar to physics in the real world, except we can control and customize the physics in the game to fit our needs. First, we need to understand the physics components in Unity. The two components that applied physics to a game object are “Collider” and “RigidBody”. There are many collider components (Collider 2D, Collider 3D, Tilemap Collider,...), each has its own function suitable for the game object it is attached to. But the main function of a collider is to create collision between two objects like should they bounce off or stick on each other. “RigidBody” component adds gravity definition to a game object. Normally, without rigid body, a game object would standstill in its defined coordinate until we control it. But with “RigidBody” component added, the game object would fall because gravity is now applied to it and it will continue to fall infinitely if there is no collider in its way. Now that we understand the functions of each components, we can add them to our game.

First let us add a collider to our environment. Since our character only interacts with the foreground, we only need to add a collider in our foreground. Select the foreground in the hierarchy and add component “Tilemap Collider 2D”, this component makes our job easier by auto detecting the tiles we draw and only applies collision to the tiles. After adding that, we should see green outline on our foreground tiles, this indicates the collision area of the foreground.

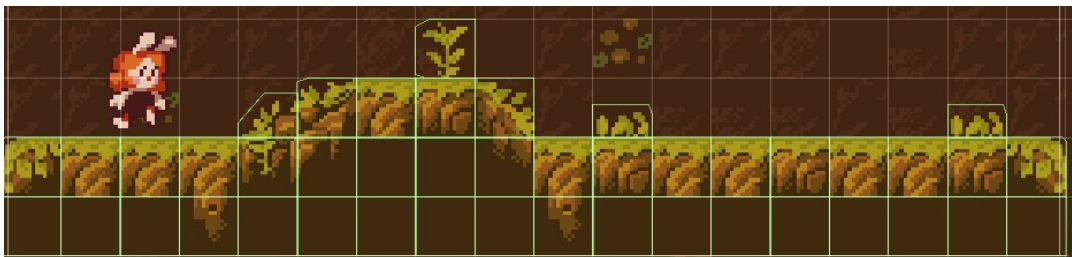


Figure 39. Collision area of the foreground.

But we can see that the plants we draw in the foreground also have collision. We do not want our character to be stopped by a small plant while moving, so this is a problem we need to fix. There are many ways to deal with this problem: draw the plants in the background, create another layer named “plants” and draw the plants on it, or remove the collision area on the plants. The most scalable option, in my own opinion, is to create another tile map and layer for the plants. This way we can manage the plants in the future without worrying messing up other layers. To do this, create another tile map inside “Grid” game object, and name it “Plant”. With the same steps creating the sorting layers, add a new layer named “Plants” and put it in between “Background” and “Foreground” layers, this way it is visible in front of the background, but behind the foreground. We can now draw freely on “Plants” tile map without worrying about the plants colliding with the player.

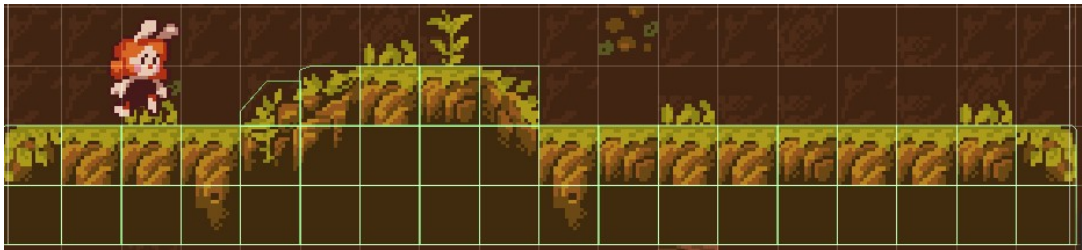


Figure 40. Plants no longer have collision.

The reason why we did not add rigid body to the environment is because it is not necessary for the environment to be pulled by gravity. Had we added rigid body component to it, our environment would fall off the screen because of gravity. Therefore, it is not advisable to add rigid body component to it. Also, it would likely create more problems to solve in the future when our game is more complex.

Now that we are done with environment collision, our character also need collision. There are options on what to use as the collider for our character (box collider, capsule collider, circle collider,...), this is based on your preference, but here I use “Box Collider 2D” component. After adding that to our “Player” game object, we can see the green outline indicating the collision area for our character. We can edit the green outline so that the collision area fit our character. We do not want it too large because that would cause frustration to our players.

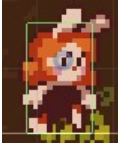



Figure 41. Collision area for character.

We can test the collision of our character with the foreground by clicking  button. Our character should not fall through the ground but stand on it. If successful, we can move on. If not, check on the instructions to see if you have done everything.


Another collision type is trigger collision, this type does not necessarily have any physics property because its main function is to trigger an event when it detects a collision. We use this type to create a way for our character to finish this level and move on to the next one. You can use any image you want as the finish line, but here I use an exit sign  to represent that of the level. Drag it in to our scene and put it at the end of the level, then create another sorting layer and physical layer both named “Interactables”. This way it makes our game more scalable since we can put all the interactable items in these layers if we decide to create more later.



Figure 42. Exit sign represent level finish line.

Then we add “Box Collider 2D” component to it and mark it as “Is Trigger”. This way, when our character collides with the exit sign, it triggers an event to load the next level, which we will define in the next section.



Figure 43. Set the collider to Is Trigger.

5.6 C# Scripts in Unity

So far, we have done environment physics, character collision, character physics, but we have not defined how our character moves based on input of the player or how the environment reacts to the character actions. This is where scripts come in handy. Understanding C# scripting is an invaluable skill for any game developer. Scripting defines the behaviors of our game objects, how they interact with each other, how player controls the character... and because of that, it creates our gameplay. In this part we create C# scripts for our character, I will guide us through the process and introduce C# as we go. First, we need to define what our character can perform. Normally in a game, the basic movements of a character are moving horizontally, jump vertically and diagonally, we can also let our character climb ladder if we decide to add ladder to our game (this is optional). For this to work, we need to have the latest version of Visual Studio on our computers. If you do not have it, open Unity Hub, go to Installs, then on the version of Unity you are using, click Add Modules, then choose Microsoft Visual Studio Community, in my case is version 2019, then click Next and start the installation. Once it is done installing, we can start scripting.

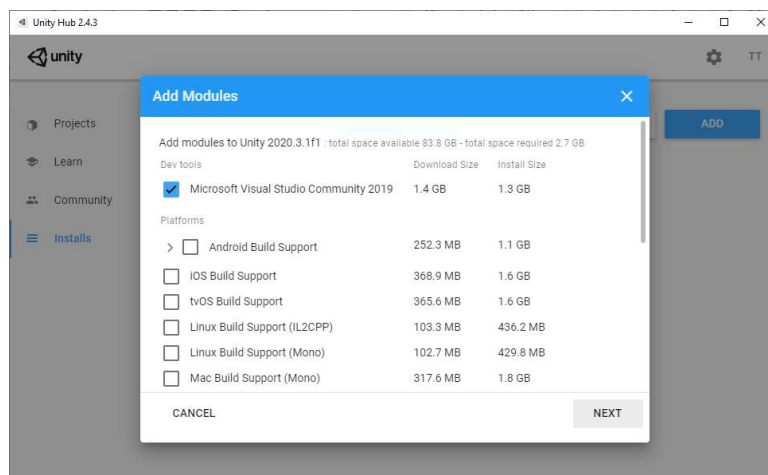


Figure 44. Add Microsoft Visual Studio Module to Unity.

First, create a folder named “Scripts” in our project, this is where we place all our scripts for easy organization. Because scripting is the most challenging part, I will

be exceptionally descriptive about what I do and for reference purposes, I will attach my scripts at the end of my thesis so everybody can have a look should they run into any problems.

5.6.1 Player.cs

In the “Scripts” folder, create a script and immediately name it “Player”. If we do anything before naming the script, it uses the default name “NewBehaviourScript” as its public class and this will cause confusion later in creating “Escape The Cave”. After naming it “Player”, the public class of the script is also “Player”. To add this script to our “Player” game object, simply drag and drop the script in the “Inspector” window of the game object. Your “Player” game object should have the script “Player” added.

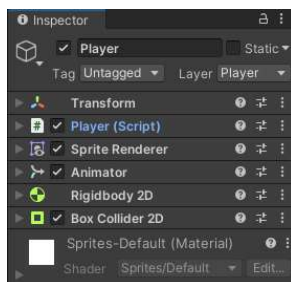


Figure 45. “Player” script added to “Player” game object.

Double click on the script to open it with Visual Studio. We need to create a method for our character to move upon inputs of the player. To do this, write these lines of codes inside public class “Player”. For best practice, write them below Start and Update methods:

```
private void Run()
{
    float controlMovement = Input.GetAxis("Horizontal");
    Vector2 playerVelocity = new Vector2(controlMovement * controlSpeed,
GetComponent<Rigidbody2D>().velocity.y);
    GetComponent<Rigidbody2D>().velocity = playerVelocity;
}
```

`private void Run()` : the declaration of the method “Run”. Private means that this method can only be accessed by its class and none other. Void is the return type of the method that specifies that the method does not return a value.

`float controlMovement = Input.GetAxis("Horizontal");` : float is to declare the floating-point numeric type of the variable “controlMovement”.

“`Input.GetAxis("Horizontal");`” is how we let player use input to control the character movement horizontally. At the end of every line of code in C# ends with semicolon “;”. This whole line means that the variable “controlMovement” returns the float value of the horizontal inputs. At this point, player can give input, but the character does not move yet. The movement control comes in the next lines of code.

`Vector2 playerVelocity = new Vector2(controlMovement * controlSpeed, GetComponent<Rigidbody2D>().velocity.y);` : this line has the same structure as the previous. “Vector2” declares the type of vector for variable “playerVelocity”. Vector2 is a 2D vector type with only x-axis and y-axis. Of course there is vector3 type for 3D use, but here we only need 2D vector. Now we have the variable “playerVelocity” as our vector for the character, and we pass in “controlMovement” from the previous line as the value for x-axis. We also need our y-axis movement to imitate the gravity so we pass in the velocity of y-axis in the “Rigidbody 2D” component. But to be able to move, we need to access the character rigid body component.

`GetComponent<Rigidbody2D>().velocity = playerVelocity;` : here we access “Rigidbody 2D” component inside “Player” game object, and assigns its velocity as “playerVelocity” so that when there is an input from the player, they actually controls the “Rigidbody2D” component of “Player” game object, and thus move the character. The full functionality of method “Run” is now complete. But we have yet to finish. As of now we have the method, but there is no place to use the method yet. To be able to control the character, we need to include this method inside the “Update” method. This would update our input every frame when we play the game. The “Update” method should look like this:

```
void Update()
```

```
{
    Run();
}
```

We can now move our character left and right, but we cannot control its speed. It might be too slow or too fast for our player, so we need to add a speed multiplier that is easy for us to control. For this purpose, we can create a Serialize Field. A serialize field is basically a way for us to control the variable in Unity. We can set an original value for the variable in Visual Studio, with the serialize field, we can change it inside Unity without having to go back to our code to find it. Simply write this code inside “Player” public class.

```
[SerializeField] float controlSpeed = 5f;
```

With this line of code, we created a float variable with value of 5. Together with the serialize field, Unity shows a field inside “Player.cs” script, which allows us to change the value of variable “controlSpeed”. Unity recognizes and automatically add a blank space between words for easy understanding.

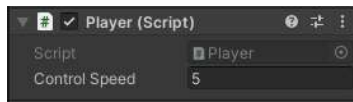


Figure 46. Serialize field “controlSpeed” in Unity.

With this variable, we can now control the player speed by simply add it as a multiplier for “controlMovement” inside the “playerVelocity” vector.

```
Vector2 playerVelocity = new Vector2(controlMovement * controlSpeed,
GetComponent<Rigidbody2D>().velocity.y);
```

Save the script and try the new movement speed, change it to fit your speed desire. But as we move our character, the animation does not change into walking animation or flip into the moving direction. Therefore, we have two problems on our hands: make character flip animation in the suitable walk direction and create means for animation to change. The first problem we fix is the flip animation. As of now, the animation of our character is visually looking at

the right side of the screen, we need to create a condition in our code to tell the animation to flip when moving left.

We will use mathematics classes for flipping function so this requires some knowledge of mathematics:

- `Mathf.Abs(f)` : this code's function is to return the absolute value of `f` whether `f` is a negative or positive number. For instance, if `f = -5`, then `Mathf.Abs(f) = 5`.
- `Mathf.Sign(f)` : this code returns the value of 1 if `f` is a positive number, -1 if `f` is a negative number. For example, `f = -5` then `Mathf.Sign(f) = -1`, `f = 5` then `Mathf.Sign(f) = 1`.
- `Mathf.Epsilon` : this is the smallest value that a float number can have different from 0.

Now that we understand the functions of the codes, we can use them to flip our character. A new method called "FlipSprite" is needed:

```
private void FlipSprite()
{
    bool playerHasMovement = Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x)
> Mathf.Epsilon;
    if (playerHasMovement)
    {
        transform.localScale = new
Vector2(Mathf.Sign(GetComponent<Rigidbody2D>().velocity.x), 1f);
    }
}
```

`bool playerHasMovement = Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x) > Mathf.Epsilon;`: this line is a true-or-false statement. It states that the

"playerHasMovement" value is true when the absolute value of movement on the x-axis is bigger than the epsilon number.

`if (playerHasMovement)` : this is an if statement. An if statement is used when there is a condition for the block of codes inside the statement to run. This line means that when "playerHasMovement" value is true, execute the following code.

`transform.localScale = new Vector2(Mathf.Sign(GetComponent<Rigidbody2D>()`

`.velocity.x), 1f);` : this line is how we flip the sprite of our character. We access the "Transform" component inside the "Player" game object and change the scale on the x-axis of the sprite to flip it. We use `Mathf.Sign()` for the x velocity of the rigid body 2D component so that it only returns 1 or -1.

Now add “FlipSprite” to “Update”, and our character should face the direction it is moving.

```
void Update()
{
    Run();
    FlipSprite();
}
```

Next, we create the jump function for our character with the same principal and structure with “Run” function. But we have to note that when moving, horizontally, the player’s input is constant and the character only stops if the player stop pushing down the move button. While with jumping mechanism, the character only reaches a certain height and then falls back down to the ground, not moving vertically constantly when there is input. For this method, we use `Input.GetButtonDown()` function, which only input the act of pushing down buttons, not constantly take input when the button is pushed. We create a serialize field variable for how height the character can jump and call it “jumpSpeed” so that we can control it inside Unity:

```
[SerializeField] float jumpSpeed = 5f;
```

Then we create the jump method:

```
private void Jump()
{
    if(Input.GetButtonDown("Jump"))
    {
        Vector2 playerJump = new Vector2(0f, jumpSpeed);
        GetComponent<Rigidbody2D>().velocity += playerJump;
    }
}
```

The reason why we use if statement here is because the

`Input.GetButtonDown("Jump")` returns bool type value. When the player input from the keyboard is detected, the character will jump then and only then.

`Vector2 playerJump = new Vector2(0f, jumpSpeed);` : this creates a new vector with “jumpSpeed” variable in the y-axis.

`GetComponent<Rigidbody2D>().velocity += playerJump;` : the line of code means that whatever the velocity of the rigid body component of the character is then, adds with the “`playerJump`” vector to it. Our character is now able to jump. Adjust the jump speed to your liking.

However, there is an exploit in the jump function. Our character can jump whenever the player press jump, which means that it can jump from any position possible and does not necessarily have to fall to the ground to start jumping again. We do not want that to happen because it would make our game less fun when players can just jump out of any challenges. To prevent this, we need to make the player able to jump only when the character is touching the ground. We use “if” statement once again for this logic condition: if the player is not touching the ground, then do not let the player jump. This way, the player cannot jump while floating in the air anymore. We add the “if” statement inside the “`Jump()`” method:

```
if (!GetComponent<Collider2D>().IsTouchingLayers(LayerMask.GetMask("Foreground")))
{ return; }
```

the “!” stands for “not” in the “if” statement. Here we access the component “`Collider2D`” of “Player” `GameObject` and using “`IsTouchingLayers()`”, we can determine whether the character is touching the “Foreground” layer or not. When we want a condition to do nothing, simply write “`return`”. This means that if the character is not touching the foreground, do not let the player use “`Jump()`” function.

So far, our character can move, jump, face the moving direction, but does not play the corresponding animation for each of the actions yet because we have not defined the conditions for these animations to be triggered. We have created the states and the transition conditions “`isJumping`” and “`isWalking`” in the previous “Character Creation” section. now we implement these conditions in our code. To trigger the “Player Walk” animation, we need to set the bool “`isWalking`” to true when our character has horizontal movement. This logic is the same as the one we use for “`FlipSprite`” method. We can use the bool “`playerHasMovement`”

inside the “Run” method. Once we have that, we pass the bool value in the “Animator” component. For that, inside the “Run” method should have two more lines of code:

```
bool playerHasMovement = Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x) >
Mathf.Epsilon;
GetComponent<Animator>().SetBool("isWalking", playerHasMovement);
```

With “isJumping”, the logic is the same as allowing the player to jump. So inside “if” statement of the “Jump” method, set the bool for “isJumping” to true:

```
if
(!GetComponent<Collider2D>().IsTouchingLayers(LayerMask.GetMask("Foreground")))
{
    GetComponent<Animator>().SetBool("isJumping", true);
    return;
}
```

But we also need an “else” statement to change the bool back to false, or the character will continue to run the animation “Player Jump” even when the character is not jumping:

```
else
{
    GetComponent<Animator>().SetBool("isJumping", false);
}
```

With that being implemented, we are done with “Player.cs” scripts.

5.6.2 LevelFinish.cs

In this script, we create a way for our player to finish the level they are in and move on to the next one. We need to create another script called “LevelFinish.cs” inside our “Script” folder, then drag and drop it in the exit sign game object as a component.

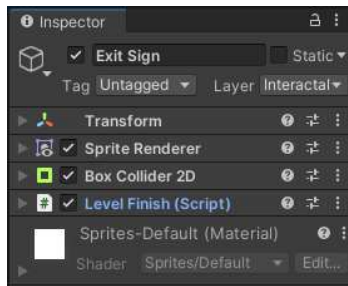


Figure 47. Level Finish script component in Exit Sign GameObject.

In this script, we do not need “Start” and “Update” method, but we need another namespace called “UnityEngine.SceneManagement;” which allow us to use the scene management tool in the script. So, add that to the namespace:

```
using UnityEngine.SceneManagement;
```

We do not want to load the next scene immediately when the character touches the exit sign, it creates a feeling of abruption for the player. Therefore, we need to let the scene wait for a while after player finishes the level, then load the next one. To do this, first we create a serialize field of a float called “delayLevelLoad” and set the value of 2:

```
[SerializeField] float delayLevelLoad = 2f;
```

We will use this variable in a coroutine. A coroutine, in layman’s term, is a way to pause the execution of a function for a certain amount of time and resume execution after the time limit. While a typical function can return any type, a coroutine must return an IEnumerator. Let use create an IEnumerator first:

```
IEnumerator LoadNextLevel()
{
    yield return new WaitForSecondsRealtime(delayLevelLoad);
    var currentSceneIndex = SceneManager.GetActiveScene().buildIndex;
    SceneManager.LoadScene(currentSceneIndex + 1);
}
```

The code “`WaitForSecondsRealtime(delayLevelLoad)`” use the variable “`delayLevelLoad`” as its value for the next code to wait for 2 seconds (value of “`delayLevelLoad`”) before executing. Then we create a new variable called “`currentSceneIndex`” and pass in the current scene build number. The last line “`SceneManager.LoadScene`

(currentSceneIndex + 1);” function is to load the scene after the one the player is currently in.

Next, we need to create a way to trigger the coroutine. In the previous chapter, we use “Box Collider 2D” and set it as a trigger for our Exit Sign. Now we use that trigger in the code with “OnTriggerEnter2D”:

```
private void OnTriggerEnter2D(Collider2D collision)
{
    StartCoroutine(LoadNextLevel());
}
```

The script for the “Exit Sign” is now done. Whenever the character touches the sign, it waits for 2 seconds then load the next scene.

5.6.3 MainMenu.cs

The MainMenu.cs script handles the functions of the “Start Game” and “Quit Game” button. Let us create the script and drag it to both “Start Game” and “Quit Game” buttons. Inside the script, we do not need “Start()” and “Update()” method, so delete that and create the method to start the game. We also use the namespace “UnityEngine.SceneManagement” for scene management functions. The logic here is that the game always starts at level one, so when the player clicks on “Start Game” button, the game loads the scene index associated with the first level:

```
public void StartFirstLevel()
{
    SceneManager.LoadScene(1);
}
```

Then, we need to create a method to quit the game using “Application.Quit()” function:

```
public void QuitGame()
{
    Application.Quit();
}
```

Save the script then drag and drop the script to both of the buttons in “Main Menu” scene. After that, to apply the function for “Start Game” button, in “Button” component in the “Inspector” window, there is “On Click ()” method, set this function like so in the picture and “Start Game” button now has the function to load the first level:

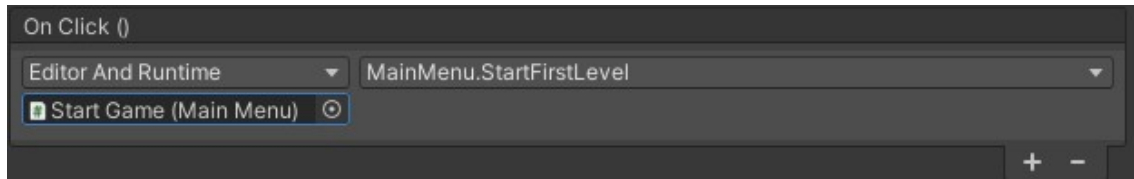


Figure 48. “Start Game” button function.

We can apply the same steps on “Quit Game” button with “QuitGame()” method. Also, the “Congratulations” scene is similar to “Main Menu” scene, apply the same settings to both buttons.

5.7 Level Design

So far, we have only created one level (or scene) for our game. To make the game feel expansive for the players, there should be more levels and challenges, which is why we need to create more scenes, each is different than the others. The level difficulty flow should be from easy to hard, so the players have time to get accustomed to the control. We can easily create another scene by duplicating the one we have now by pressing Ctrl + D, we save a great amount of time by doing this because we do not have to draw the scene or add the components again to the scene, we only need to adjust them. Our level 2 should be somewhat more challenging than level 1, for example, down below is my level 1:

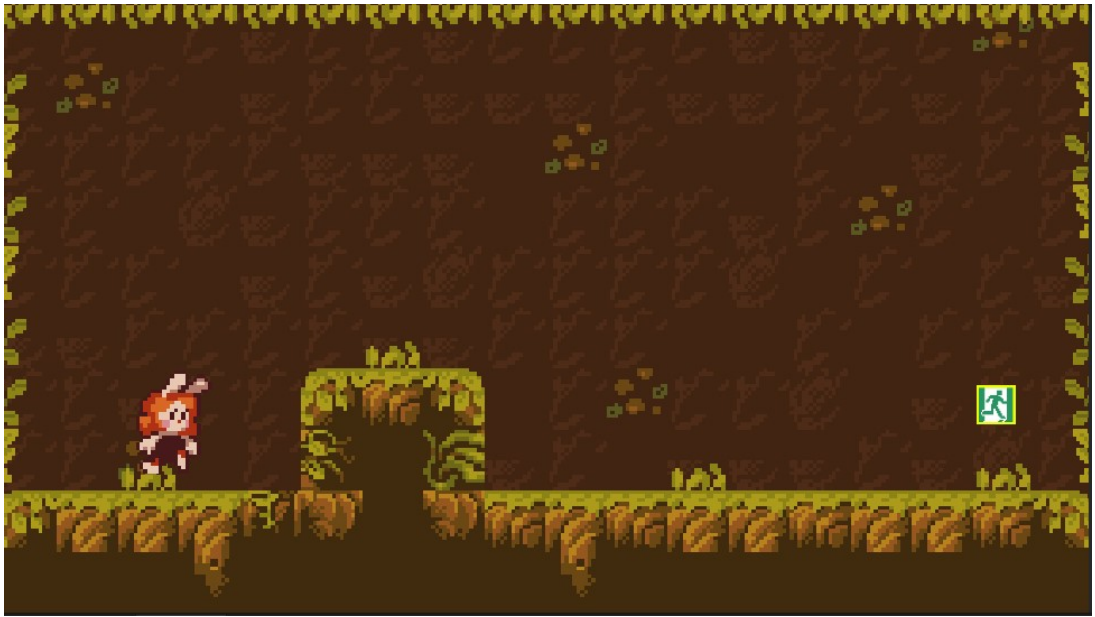


Figure 49. Level 1 scene example.

The player learns how high they can jump when they jump on the higher platform to reach for the exit. Then, on level 2, there should be some challenges involving jumping that player can overcome such as this level below:

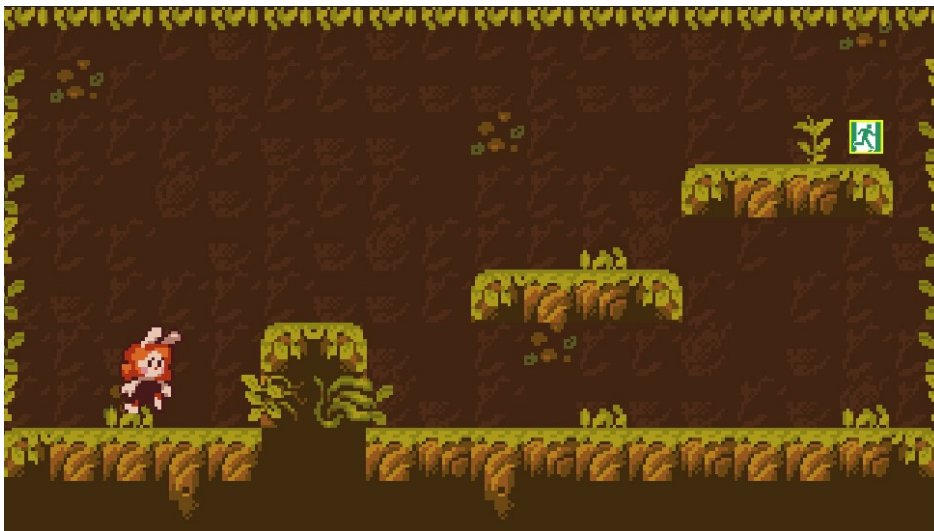


Figure 50. Level 2 scene example.

We can create as many levels as we want in our game, but here I only create 2 levels as an example of how progressive the levels should be. Maybe in later levels you can create more challenges, add more game mechanics, and put

much work into designing your levels, that is up to you. For now, we move on to creating camera movement.

5.8 Cinemachine

So far, we have only created the level in the limit of what our main camera shows. This can be very limited in level designing because we only have so much space that the player can see. In the situation where the level is too big, but our camera does not show the whole level and only stands still, it can be troublesome for the player as they can still move, but the camera does not follow him.

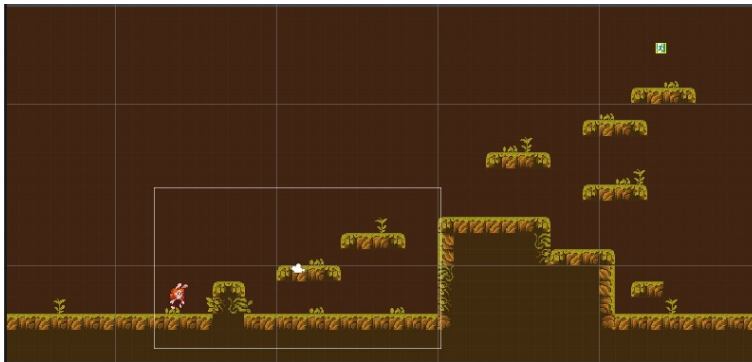


Figure 51. Example of big level with small camera.

Of course, we can expand the camera view to fit the level, but then the character would be too small. Therefore, we need a way for the camera to follow our player in the bigger level that we design with Cinemachine. Cinemachine is an add-on for Unity projects that helps with controlling the cameras we have in play. To add Cinemachine to our project, go to Window > Package Manager, change “Packages” filter to “Unity Registry”, then find Cinemachine and install.

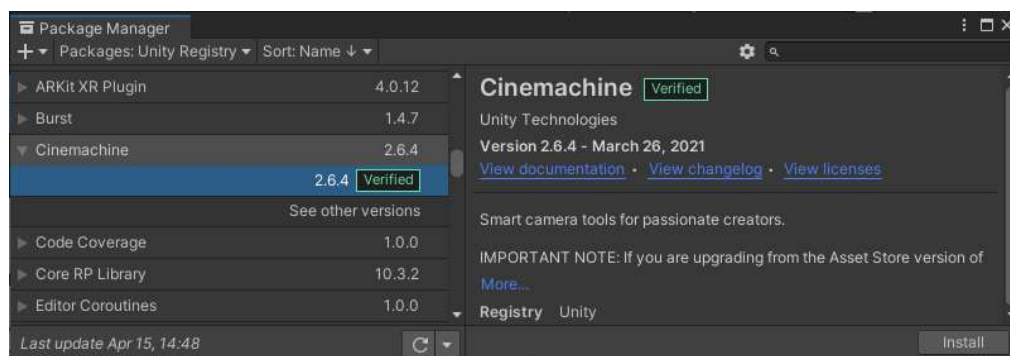


Figure 52. Cinemachine in Package Manager.

After installing Cinemachine, we can see Cinemachine on the tool bar, click on it and select “Create 2D Camera”. This creates a virtual camera in our hierarchy named “CM vcam1”. A virtual camera is an empty object, and it does not appear on our scene. The function of Cinemachine is to apply all the settings from virtual cameras to the main camera. Therefore, we mainly use virtual camera for our settings. In the virtual camera Inspector window, we can see the option to choose the GameObject the camera will follow, choose “Player”. As simple as that, now the main camera will follow our character.

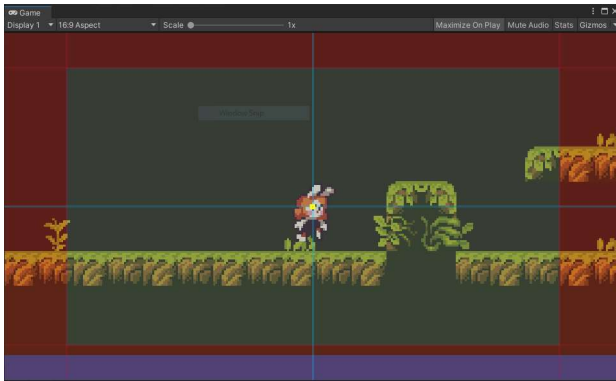


Figure 53. virtual camera follows the character.

5.9 Scene Management

At this point, we have created a game where player can control the character and many levels to play. Therefore, in this section, we build our game into a playable game software. Go to File > Build Settings.

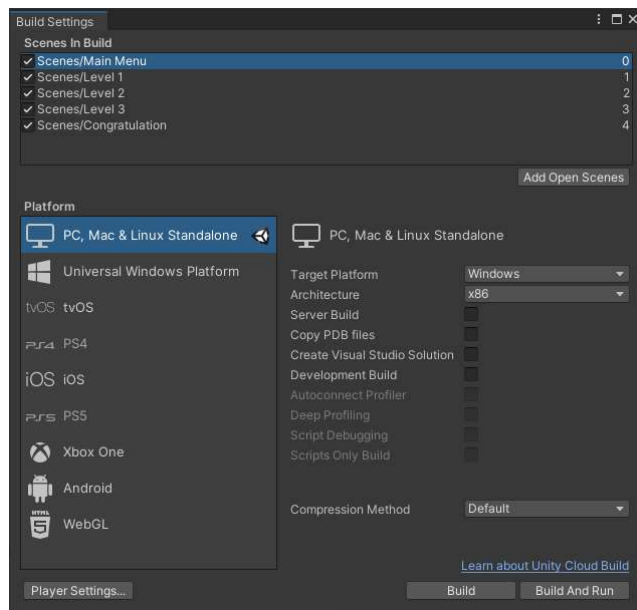


Figure 54. Build Settings window.

This is where we define the order of our level scenes and build our project into a playable game. We can add our scenes here, arrange them in the order that we want, and build the game for the targeted platforms. As can be seen from the list, there are many platforms to build on, so we should decide where we want to publish our game beforehand. For “Escape The Cave”, I decided that it is easier to build it for PC, also because PC platform is included when installing Unity. Drag and drop all your levels (or scenes) in the window, then rearrange them according to your designs so that after finishing a level, the game jumps to the more challenging one. When done, select your preferred platform and click on “Build”, then choose a folder to put all the files in it, and wait for the building process to complete. After build completion, the game is now a standalone software ready to use.

6 CONCLUSION

The purpose of this thesis is to introduce programmers to game development with Unity and show them how easy it is to get started on the journey. Many of Unity’s functions and tools were introduced throughout the thesis, the tutorial is also very detailed to help readers follow without having any difficulty. This way, the purpose of introducing game development with Unity is executed successfully. The possibility for the readers to further their study in game

development on their own is high, in my opinion, after reading the thesis as the thesis demonstrates the ease of developing a 2D game when they already have the programming skills.

Unity Assets store has so many free assets that helps developers save time and effort in making their game. It also has assets that requires payment but the price is reasonable and the amount of free assets makes developers rarely need to pay for other assets.

The Unity's support community is also sizeable. Game developers can visit <https://forum.unity.com/> to seek support form the community when they are in a bind.

Unity is a great game engine that both individuals and large companies use to make their games. It supports the publication on many platforms so that developers have the option to introduce their game on those platforms, which would boost the popularity and revenue of the game.

Unity's popularity shows us the usefulness and the freedom we have when developing games on Unity. Of course, many triple-A game companies have the resources to make their own game engines, but the use of an already great and free game engine like Unity cannot be denied when the gaming industry is on the rise nowadays. Therefore, Unity has proven to be a considerable choice for individuals to make their own indie games.

REFERENCES

Jeff Ward, 2008. What is a Game Engine? Available at:

https://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=1

Tristan Donovan, 2010. Replay: The History of Video Games. Available at:

<https://1lib.sk/book/1103623/723bf9>

Unity, 2021. Full toolset for 2D and 3D video games. Available at:

<https://unity.com/how-to/difference-between-2D-and-3D-games>

Unity Documentation version 2021.1, 2021. Unity Manual, Scripting. Available at:

<https://docs.unity3d.com/2020.1/Documentation/Manual/ScriptingSection.html>

Unity Documentation version 2021.1, 2021. Scripting API, Scene Manager.

Available at:

<https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html>

Unity Documentation version 2021.1, 2021. Scripting API, Input.GetAxis.

Available at: <https://docs.unity3d.com/ScriptReference/Input.GetAxis.html>

Unity Documentation version 2021.1, 2021. Scripting API, Vector2. Available at:

<https://docs.unity3d.com/ScriptReference/Vector2.html>

Player.cs Scripts

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Player : MonoBehaviour
{
    [SerializeField] float controlSpeed = 5f;
    [SerializeField] float jumpSpeed = 5f;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        Run();
        FlipSprite();
        Jump();
    }

    private void Run()
    {
        float controlMovement = Input.GetAxis("Horizontal");
        Vector2 playerVelocity = new Vector2(controlMovement * controlSpeed,
GetComponent<Rigidbody2D>().velocity.y);
        GetComponent<Rigidbody2D>().velocity = playerVelocity;

        bool playerHasMovement = Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x)
> Mathf.Epsilon;
        GetComponent<Animator>().SetBool("isWalking", playerHasMovement);
    }

    private void Jump()
    {
        if
(!GetComponent<Collider2D>().IsTouchingLayers(LayerMask.GetMask("Foreground")))
        {
            GetComponent<Animator>().SetBool("isJumping", true);
            return;
        }
        else
        {
            GetComponent<Animator>().SetBool("isJumping", false);
        }
        if(Input.GetButtonDown("Jump"))
        {
            Vector2 playerJump = new Vector2(0f, jumpSpeed);
            GetComponent<Rigidbody2D>().velocity += playerJump;
        }
    }
}

```

```
private void FlipSprite()
{
    bool playerHasMovement = Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x)
> Mathf.Epsilon;
    if (playerHasMovement)
    {
        transform.localScale = new
Vector2(Mathf.Sign(GetComponent<Rigidbody2D>().velocity.x), 1f);
    }
}
}
```

LevelFinish.cs Script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class LevelFinish : MonoBehaviour
{
    [SerializeField] float delayLevelLoad = 2f;

    IEnumerator LoadNextLevel()
    {
        yield return new WaitForSecondsRealtime(delayLevelLoad);
        var currentSceneIndex = SceneManager.GetActiveScene().buildIndex;
        SceneManager.LoadScene(currentSceneIndex + 1);
    }
    private void OnTriggerEnter2D(Collider2D collision)
    {
        StartCoroutine(LoadNextLevel());
    }
}
```

MainMenu.cs Script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour
{
    public void StartFirstLevel()
    {
        SceneManager.LoadScene(1);
    }
    public void QuitGame()
    {
        Application.Quit();
    }
}
```