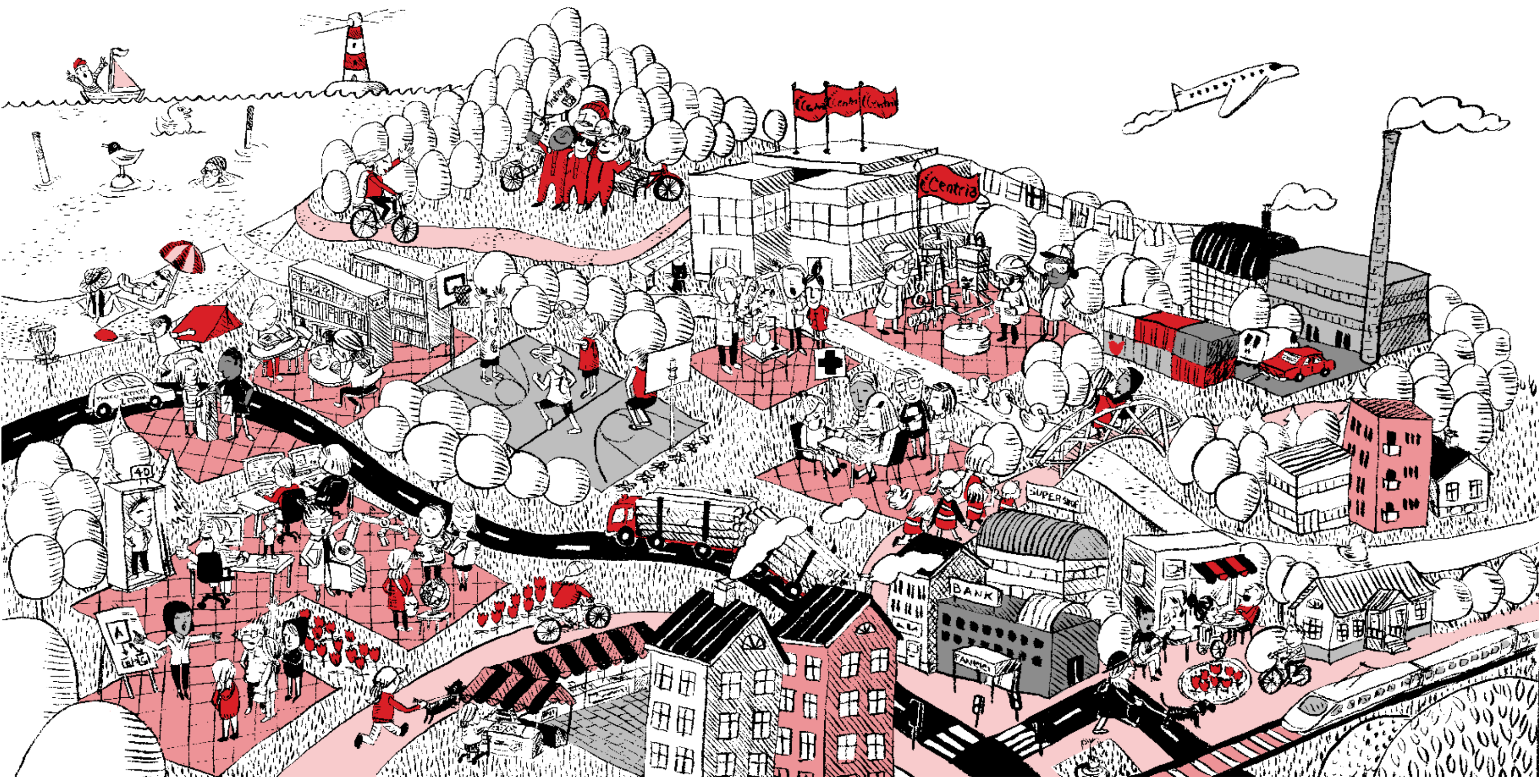
Khanh Van

CREATING A 2D TOWER DEFEND GAME

Developed with Unity game engine

Thesis
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Information Technology
December 2021



ABSTRACT

Centria University of Applied Sciences	**Date** December 2021	**Author** Khanh Van
Degree programme Information Technology		
Name of thesis CREATING A 2D TOWER DEFEND GAME. Developed with Unity game engine		
Centria supervisor Kauko Kolehmainen		**Pages** 44 + 21

The topic of this thesis was to create a 2D tower defend game using Unity game engine. Unity has been constantly updating its version with new features for game development, making it easier for beginner to approach the subject. Cross-platform games are getting more popular every day, therefore; Unity is a good choice to pickup as it is a cross-platform game engine. Switching from one to another is very simple.

The game developed in this thesis was a 2D tower defend game, where the player must defend their base with strategic placements of defenders to stop the attackers. The main topics discussed in the thesis include attacker and defender spawning, health system, and resources system. The aim of this thesis was to introduce beginners to game development through mostly scripting. Other topics such as animations, arts and sound were not discussed thoroughly.

The thesis can be used as guidance for game development, but should not be taken as step-by-step tutorial.

Key words
Game development, 2D, Tower Defend, Unity, Scripting.

ABSTRACT
CONTENTS

1 INTRODUCTION

Game development is getting more and more popular every day. Since there are so many materials to learn on the subject on the Internet, getting started with game development has never been easier. Although there are many game engines that developers can learn, two of the most popular game engines right now are Unity and Unreal Engine. This thesis acts as guidance to 2D game development. The content of this thesis is the process of creating a game through its core features, and to give an introduction to the Unity game engine. Unity offers a free and a paid version of its engine, and the free version is used for this project. Some features are disabled; however, the tools provided in the free version are more than sufficient for game development.

The game mentioned in this thesis is a 2D tower defend game, where the players are required to place down defenders to stop the enemy attackers from breaching their base. The project is an attempt of replication of the game Plant vs. Zombie for the purpose of learning. Unity also offers tools for 3D game development. Choosing which type is purely based on personal reference.

This thesis goes through mostly on how scripting helps to determine the course of actions of the objects in the game. It is explained and desmonstrated through figures. Basic knowledge of C# is sufficient as Unity offers tools and assistance to 2D game development; many features do not need to be created from scratch. The main discussion will be on how the scripts and game mechanics are built and operate. However, other topics including sound designs, animations, art design are not discussed in detail. This is to avoid lengthy documents and unnecessary confusing on the main topic. However, if game development is done by one individual, all these topics should be considered for a larger, more complicated project.

All the terms used in this thesis are identical to Unity's Document to help with further research on the subject.

2 THEORETICAL FRAMEWORK

Unity is one of the most popular game engines today, besides Unreal Engine. Unity is a game engine for multiple platforms that support 2D and 3D graphics. The engine helps the developer with the project by providing built-in features such as physics, 3D rendering, collision detection and more so the developers do not have to make everything from scratch. Unity supports C# and UnityScript, which is similar to JavaScript so it is very easy to comprehend. Recently, from Unity 2017 beta 2, UnityScript is no longer supported. (Adam 2021.)

Unity is chosen over Unreal Engine and other game engines for this process because of its simplicity and beginner friendly interfaces. Additionally, C# is used as scripting language for Unity, which is considered by the majority to be easier to approach than C++ of Unreal Engine.

2.1 History of Unity

Unity Technologies was the developer of Unity Game Engine. The company was founded by David Helgason, Joachim Ante, and Nicholas Francis in 2004. The three aim to create an engine that is affordable and accessible to the public. Unity Technologies was helped by the funds of global companies to achieve this goal. The engine was developed in C and C++ languages. C# and Java languages are used for scripting in the Engine itself. (Corazza 2013.)

Making 2D and 3D game development accessible to many independent developers is a major contribution to Unity's success. Unity launched its initial version in 2005, during Apple's Worldwide Developers Conference. The first version only supports Mac platform but it gained enough success for the engine to keep growing. In September 2010, Unity 3 was introduced, and on June 18, 2012, Unity 4 was released with additional functions and tools for both professional and amateur developers (Corazza 2013). As of today (December 2021), the newest recommended version is Unity 2020.3.5f1(LTS).

2.2 Licenses

Unity has three forms of licensing, Unity Personal, Unity Plus and Unity Pro. Unity Personal has all the basic functions and tools for game development. The Unity Plus and Pro tier include more features and services as they cost more. The most well-known one is the Unity splash screen. There is a "Made with Unity" splash screen in all tiers of Unity licenses, customizable options feature like blurred background image and company logos are under development. While the splash screen is mandatory in Unity Personal, Unity Plus and Pro has the option to turn off splash screen completely. (Downie 2016.)

Unity's EULA Agreement only allows the free version (Unity Personal) users to have annual revenues or funding of 100,000 USD. If the revenue or funding is beyond this, users must upgrade to Plus or Pro version or else their permission will be retracted. Unity Plus raises this cap to 200,000 USD, and Unity Pro has no revenues or funding cap. Unity allows users to upgrade to higher tier freely in the middle of the project while not causing any technical problems, since the upgrades only add new features. (Downie 2016.)

2.3 Unity Interface

Unity interface which is called main editor window consists of tabbed windows which can be resized, moved, grouped, or detached. This allows users to set the windows based on their personal preference making it different for every project. (UnityTechnologies 2021.)

Each tab and window have a specified function, without any initial customization. The main editor windows include: The Toolbar, The Hierarchy Window, The Game View, The Scene View, The Inspector Window, The Project Window, and The Status Bar. (UnityTechnologies 2021.)

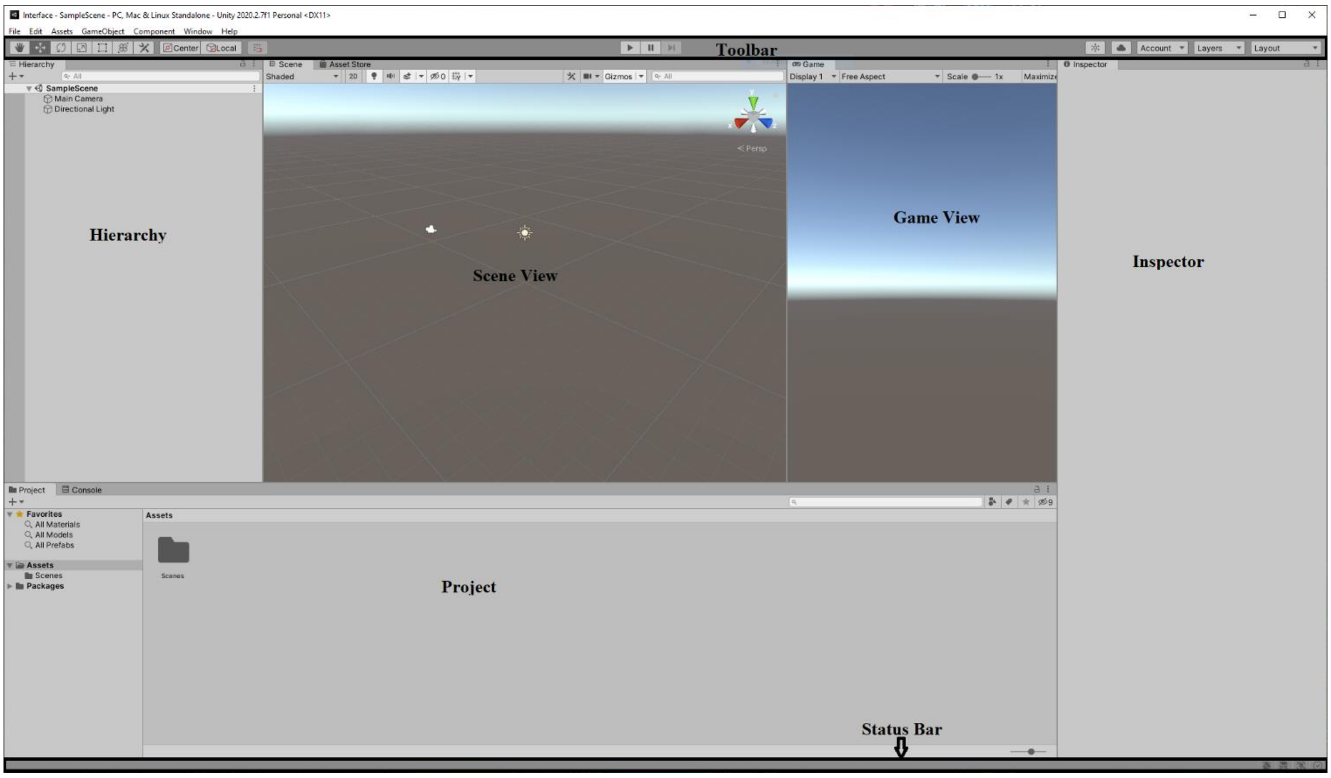


Figure 1. Unity Default Interface

The Toolbar is one of the few elements that cannot be rearranged or moved in the main editor view. On the left side of the bar is the group of control tools for transformations of GameObjects, consisting of Move, Rotate, Scale, Rect Transform, and Transform. These are the most commonly used tools in the Toolbar. If there is a GameObject selected, the controls right next to the Transform group are also displayed. These are used for toggling Transform Gizmo affects in the Scene View. In the middle is the Play, Pause and Step buttons to navigate in Game View. There are also drop-down buttons for modification of Layers and Layout. (UnityTechnologies 2021.)

Every GameObject in the Scene is contained in the Hierarchy Window. GameObjects can be added or removed through this window. If the GameObject is deleted directly from the Scene View, it will also be deleted in this window. GameObjects in this window can also be sorted and grouped. Unity utilizes the concept of parenting to GameObjects. This means a GameObject can contain other GameObjects, the contained GameObjects will have the attributes of the parent GameObject such as rotations, scale, transforms, and more. This will help ease the modification of multiple similar GameObjects. Parenting can be created by dragging a GameObject onto another, making the child of that GameObject. (UnityTechnologies 2021.)

The Scene View is an interactive space used for manipulating GameObjects. GameObjects in Scene view can be selected and repositioned or hide from view. Cameras and light sources are also considered GameObjects that can be manipulated. Getting comfortable with objects modification in Scene View is one of the first and important steps to learning Unity. (UnityTechnologies 2021.)

Game View allows developers to see from the Cameras' point of view of the final application. Game View helps the developers control what the players can and cannot see in the game using one or more Cameras. The group of buttons in the middle of the Toolbar is used to enter, leave, or pause Play mode. All modification made in Play mode are temporary and will be reset when leaving Play mode. The Editor UI is darkened to indicate when Play mode is active. (UnityTechnologies 2021.)

The Inspector Window is used to observe and modify properties of almost anything in Unity Editor. By default, properties of currently selected item will be displayed in the Inspector Window, properties will change if a new item is selected. If users choose to lock the inspector, the contents will no longer be updated if a new item is selected. Items that can be selected and viewed in Inspector Window include GameObjects, Assets, Unity components, Materials and In-Editor preferences and settings. When selecting multiple GameObjects, the Inspector Window will display the component that the GameObjects have in common. If the values are different in each GameObject, a dash (-) is displayed, if the values are the same, the actual value will be displayed. (UnityTechnologies 2021.)

The Project Window contains all the files of the current Project. Users can navigate and find files and Assets of their project from this window. The Assets contained in this window can be used directly by dragging them into the Scene View. Script files can be dragged into GameObjects to add the script component to those GameObjects. The Project Window also allows users to directly search and select Assets from Unity Assets Store. (UnityTechnologies 2021.)

Like the Toolbar, the Status Bar cannot be moved or rearranged. Status bar displays information of many Unity processes, and access to tools and settings. Firstly,this bar displays the most recent message in Console window. User can open Console window by clicking on the message. Secondly, it also contains global progress bar for asynchronous tasks. (UnityTechnologies 2021.)

On the right part of the Status bar is a group of controls, consisting of current optimization mode, and users can switch between debug and release mode on click. Additionally, this group also has the cache server status. Click for the cache server information and to reconnect a lost connection, the automatic lighting generation for Global illumination and activity indicator. (UnityTechnologies 2021.)

2.4 Game Design Theory

The definition of game design is the process of planning the fundamental aspects of a game, including firstly, game mechanics and systems. These are the rules of how the game should work and how the characters and objects should interact in the game. Secondly, gameplay, this is how the developers want the players to interact with the game. Finally, player experience, how players feel while playing the game. Other aspects of the game development process such as art, modelling, programming, sounds and music, even storytelling and character designs are not considered as game design. A game designer might have some influence on these other aspects, but this does not mean that they are a mandatory part of game design. (Burgun 2012, 18.)

To create an interesting game, there are certain basic principles that designers should follow. There are three main elements of game design including the player, the communication, and the appeal. The players should always have a purpose in the game. They should be the one leading the game forward, and the developers need to communicate this purpose to them. If this purpose is unclear, or there is none at all, the players will feel lost and unimportant, leading to boredom. Finally, the most important element of game design is the appeal of the game to the players. None of what were mentioned matter if the players do not like the look or the feel of the game. Appeal is hard to achieve since it varies from game to game and from player to player. Some games resolve around stunning graphics and scenery, while others focus on stories and puzzles. Some players enjoy fast-paced action combat, while other like to enjoy the beautiful views of the world they are playing in. (Brackeys 2018.)

3 GAME DEVELOPMENT

The first step of every game or software development process is to have a clear plan of the overall project. Planning and requirements is an essential part of the development process. For game development in particular, for the game to have a chance to be successful, it must have proper game designs, game art, sounds, and interesting gameplay. Having clear goals and schedules make it easier for the development process and for necessary changes. With the combination of decent planning and the help of the game engine, creating an intriguing game is much easier. (Basu 2017.)

3.1 Planning and requirements

The first step of creating a game is laying out its core requirements. This step is also considered game design (Basu 2017). Firstly, the player experience, as the game developed in this thesis is a tower defend game, strategic thinking is the experience that is aimed for. Player experience always needs to be considered or the project will become some random confusing game. Secondly, the core mechanic of this game is to place defenders to stop attackers. There will be multiple waves of attackers spawning and the player objective is to place down defenders to stop the attackers from reaching its destination. This is similar to Plants vs. Zombies. The main theme of our game will be garden, the defenders are either plants or objects in a typical garden, and the attackers are wild animals. Finally, killing all the attackers then proceeds to next level will be our core game loop.

The second step of planning on a game development is choosing art assets and sound effects for the game (Basu 2017). There are numerous sources of art assets online including assets from the Unity Store, artworks from artists or our own source of artworks. Choosing suitable assets for our game can be a challenging task. Although there are many free assets from the Unity Store to choose from, most of the high-quality assets are not free. This can be a problem to small, independent developers since their budget is usually quite small. Additionally, there might be times when there are no fitting assets for our desired games. All the problems above can be solved by acquiring at least some basic knowledge on graphic design and design tools like Photoshop, Audacity, and more. The similar problems and solutions can be applied to sound effects, sound design, and game music.

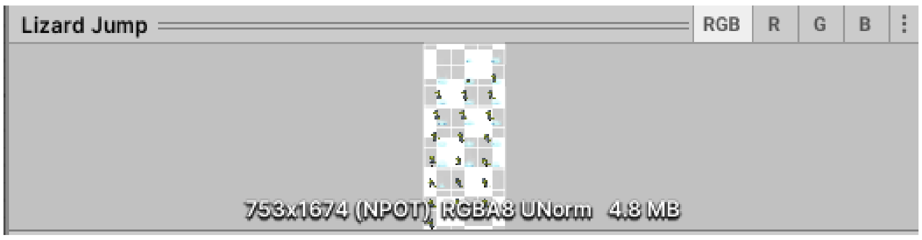
Finally, the last essential part of game development planning is scripting (Basu 2017). Game design and assets can only make the game look nice and appealing, but scripting is needed to operate the game. Scripting in Unity is written in the C# language. Once the scripts are attached to the GameObjects, the game will start to function as we expected it. MonoDevelop is the default built-in Integrated Development Environment (IDE) of Unity. However, for the game developed in this thesis, Visual Studio Code is used as an IDE for scripting. All Visual Studio IDEs are accepted in Unity, we just need to modify the default IDE from Unity's settings.

3.2 Game art and sound

For our game's visuals and sound, we would use the free assets from Glitch. For sound effects, we already have the premade assets from Glitch so implementing sound is simple and direct, manually creating new sound effects is not required. As for visuals, however, animating characters is required. The next section will be on Sprites.

3.3 Sprites

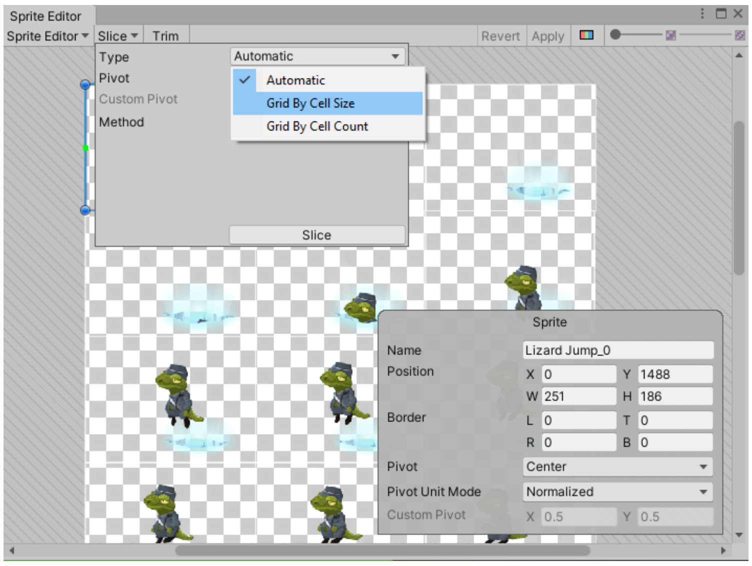
Sprites are 2D graphic objects, which are used for visual representation of characters, objects, projectiles and almost every visible element of a 2D game. Sprites are bitmap images which can both be static or animated. Sprites especially the ones in games usually have a role to play independently on a larger background environment (TechTerms 2012). Sprites can be viewed in Unity in the Scene View through a GameObject component, SpriteRenderer. The SpriteRenderer component allows us to modify the Sprite color, material, its layer order, and flipping its X and Y axes. (UnityTechnologies 2021.)



753x1674 (NPOT) RGBA8 UNorm 4.8 MB

Figure 2. Sprite Image Size

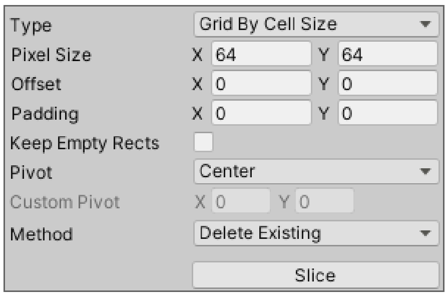
It is possible to create Sprite images ourselves, but it will not be discussed here to avoid redundancy of information. We will focus on how to slice Sprite images provided in the art assets into many Sprite textures using the Sprite Editor tool. The Sprite Editor tool can be found on the Inspector Window when a Sprite image is selected. There is also one more important information, which is the size of the Sprite image. This can be seen at the bottom right corner of the Inspector Window (UnityTechnologies 2021) . As seen from the Figure 2 above, the resolution of the image is 753x1674. This is important to know when slicing complicated images.



Name: Lizard Jump_0
Position: X 0, Y 1488, W 251, H 186
Border: L 0, T 0, R 0, B 0
Pivot: Center
Pivot Unit Mode: Normalized
Custom Pivot: X 0.5, Y 0.5

Figure 3. Sprite Editor

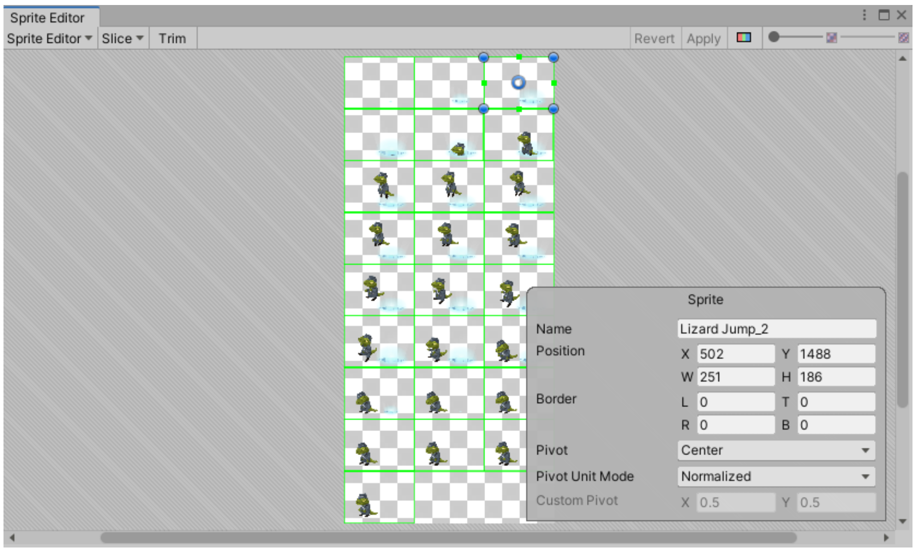
Figures 3 represent the Sprite Editor Window. This is one Sprite image of the “Lizard Jump” action. The image can be slice into multiple textures which complete the whole animation. There are three options for slicing: automatic, grid by cell size, and grid by cell count. The automatic option will try to find anything that is not an empty background and snap them in square or rectangular pieces. This is effective if the image is just a simple walking animation. However, in the jumping action image, the automatic option will create uneven snaps, so it is optimal to use grid by cell size option. (UnityTechnologies 2021.)



Type	Grid By Cell Size
Pixel Size	X 64 Y 64
Offset	X 0 Y 0
Padding	X 0 Y 0
Keep Empty Rects	☐
Pivot	Center
Custom Pivot	X 0 Y 0
Method	Delete Existing
	Slice

Figure 4. Grid by Cell Size window

As seen in figure 4 above, the image can be cropped with the corresponding size of X and Y. There are a total of 3 columns and 9 rows in this image. Using the information we have on the image resolution above; we can slice the image equally by dividing 753 by 3 and 1674 by 9. Figure 5 show the image after cropping, the green lines indicate where the cuts are. Each texture can now be use as a single frame for animating.



Name	Lizard Jump_2
Position	X 502 Y 1488
	W 251 H 186
Border	L 0 T 0
	R 0 B 0
Pivot	Center
Pivot Unit Mode	Normalized
Custom Pivot	X 0.5 Y 0.5

Figure 5. Sliced Sprite Image

3.4 Scripting

Scripting is a crucial part of development. Having a script attached to a GameObject is the only way to make that GameObject respond to player's input. Using scripts is also a way to control character behaviour and even implementing Artificial Intelligent (AI) element to that character. A script can be created easily by right-clicking on the Project Window, choose Create, and choose C# Script as seen in Figure 6 below. (UnityTechnologies 2021.)

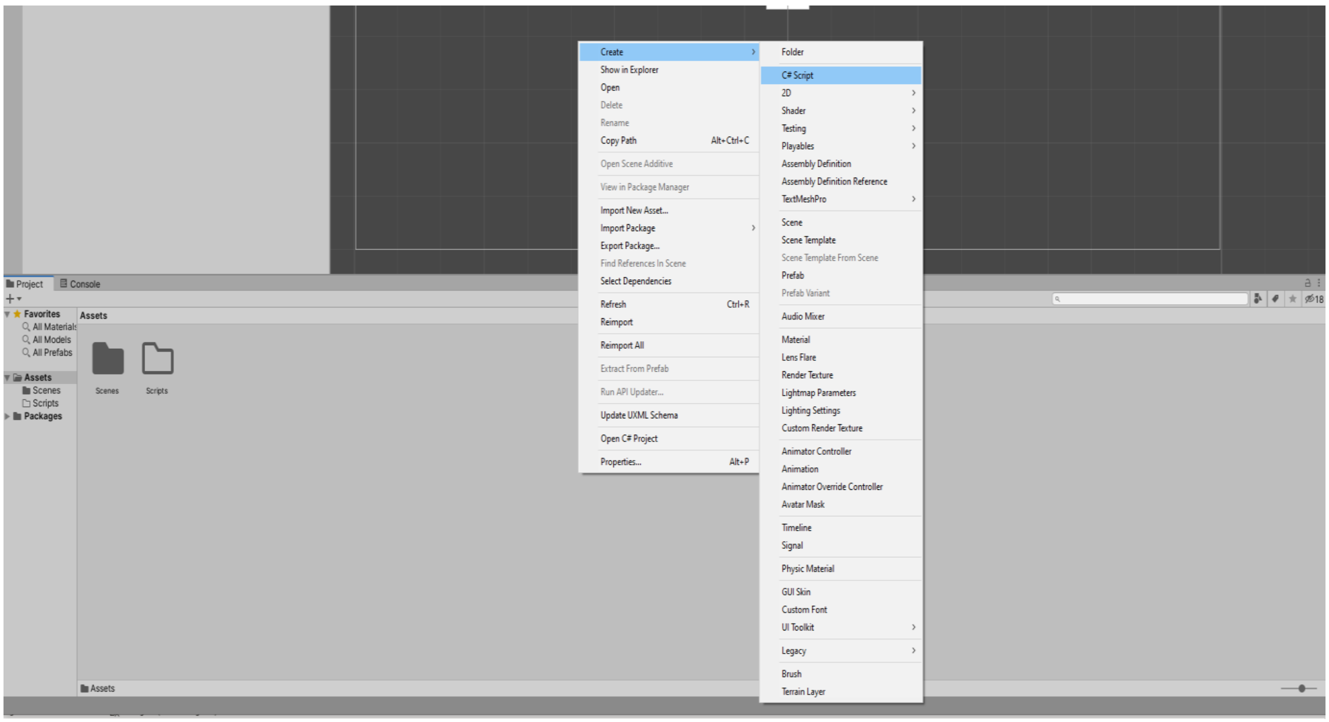
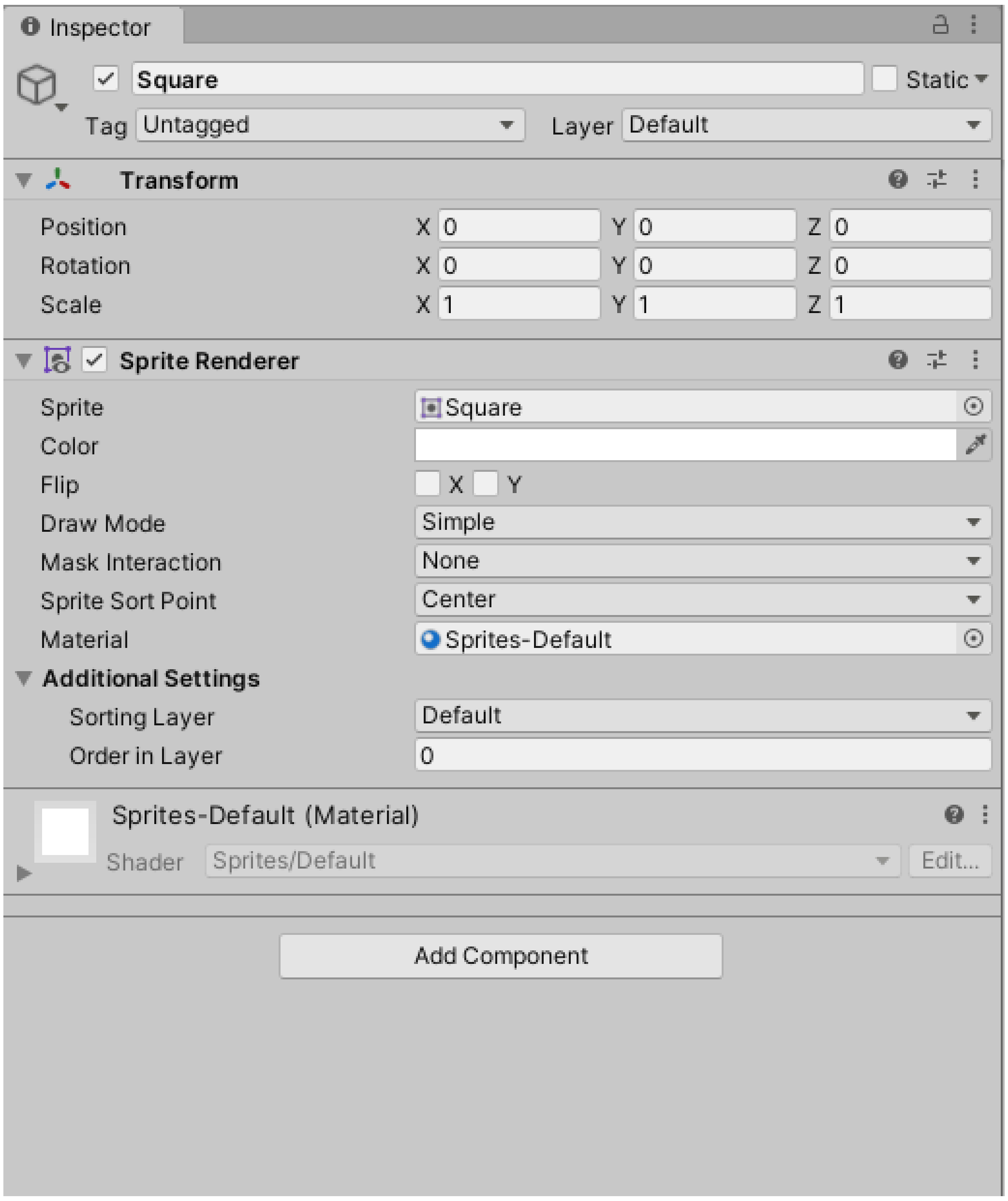


Figure 6. Creating C# Scripts

There are two ways to attach a script to a GameObject. The first way is to add the script as a component to the GameObject.



Inspector

Square — Static

Tag: Untagged — Layer: Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

Sprite Renderer

Sprite	Square
Color	
Flip	X Y
Draw Mode	Simple
Mask Interaction	None
Sprite Sort Point	Center
Material	Sprites-Default

Additional Settings

Sorting Layer	Default
Order in Layer	0

Sprites-Default (Material)

Shader: Sprites/Default — Edit...

Add Component

Figure 7. Adding Script component

As seen in Figure 7, when the GameObject is selected, a list of components of that object is displayed in the Inspector Window. To add a script, simply click on the “Add Component” button, and a dropdown window will appear with all the possible components that can be added. (UnityTechnologies 2021.)

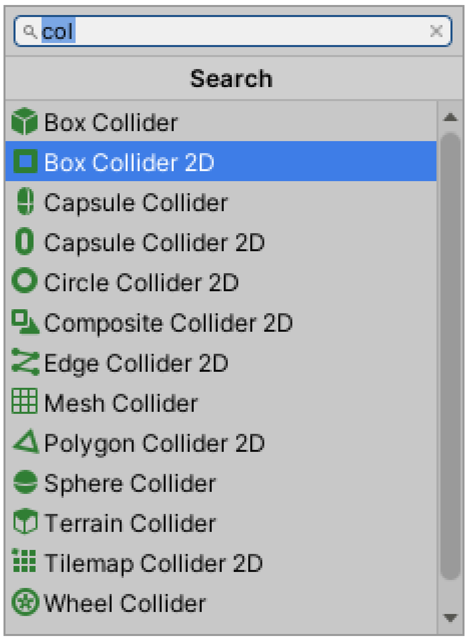
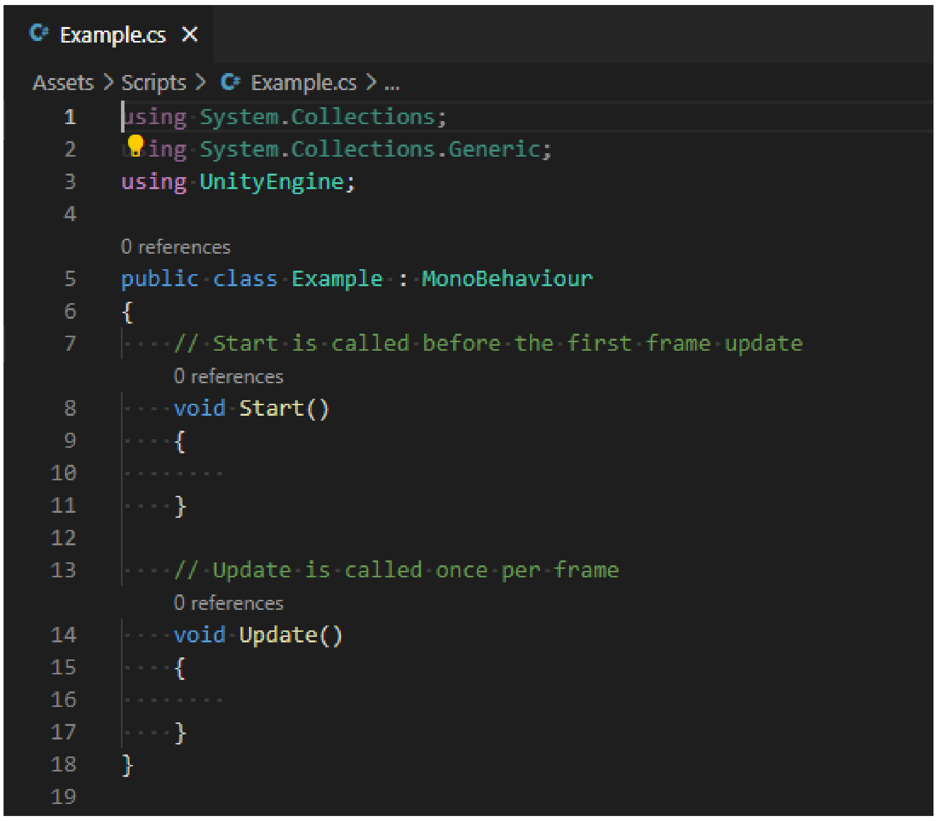


Figure 8. Add component window

Figure 8 displays the “Add Component” dropdown window. Developers can scroll down and look for the name of the desired script , or just type in its name on the search bar. New script can also be created by selecting the ”New script” option from the dropdown window with the search bar emptied. The second way to attach a script is to directly drag that script from the Project Window and drop it on the Inspector Window of the desired GameObject. (UnityTechnologies 2021.)

When a script is opened with our selected IDE from Unity, it is shown that our script is a class derived from the MonoBehaviour class in Unity, and there are two default functions. As seen in Figure 5 below, our script, which is named “Example” is a derived class of the MonoBehaviour class. To avoid unnecessary confusion, definition of MonoBehaviour will not be discussed. However, be reminded that every script used in Unity need to be derived from this class. (UnityTechnologies 2021.)



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

0 references
public class Example : MonoBehaviour
{
    // Start is called before the first frame update
    0 references
    void Start()
    {
        
    }

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

```

Figure 9. Default Unity Script

By default, there are comments on what the functions are. Start function is called immediately before the first frame, and the Update function is called once every frame. There are many more functions including Awake (), FixedUpdate(), OnApplicationPause(), OnDestroy(), Coroutines, and more. What and how these functions are used will be discussed later in this thesis once they are encountered. One important thing we need to consider when renaming scripts is that if we rename the scripts in the Project Window, the name of the derived class in the IDE will remain unchanged. Therefore, to avoid conflicts, we need to make sure that both the names from the Project Window and the IDE are changed, or just recreate a new script entirely if it is empty. (UnityTechnologies 2021.)

4 CREATING THE GAME

The game mentioned in this thesis can be set as an example on creating 2D games. The content in this section can be used as guidance for creating core elements for a lane based 2D tower defend game. The basics and necessary scripts will be explained. All the scripts used for this project can be found in the Appendices.

4.1 Creating New Project with Unity Hub

To create a new project for the game, downloading and installing Unity Hub is required. Unity Hub is an application for users to manage their Unity projects and installs easier. First, we need to install a version of Unity. There are many versions, and they are updated fairly frequently. (UnityTechnologies 2021.)

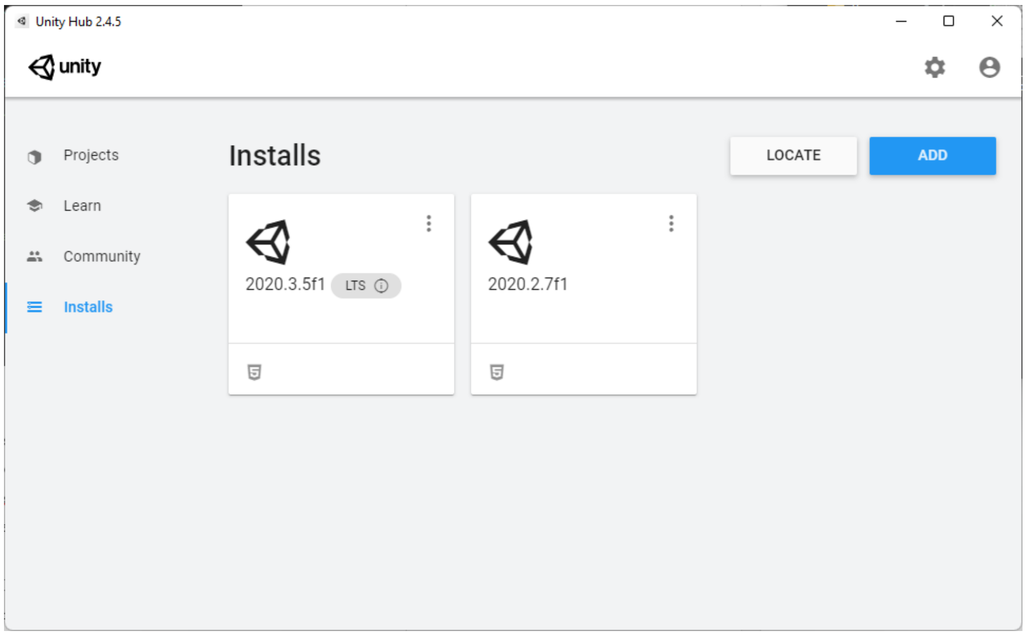
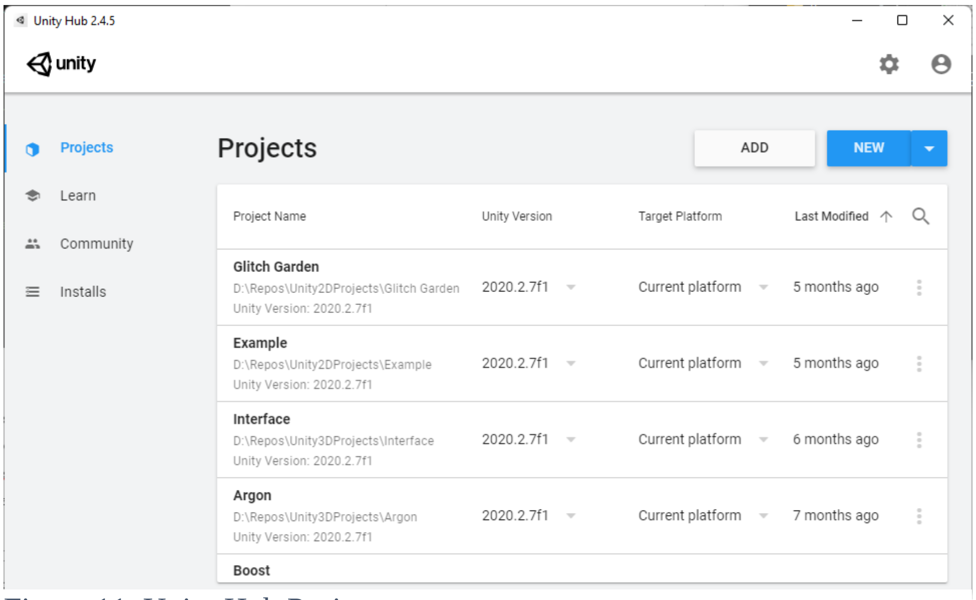


Figure 10. Unity Hub Installs

As shown in figure 10 above, a new version of Unity can be installed by pressing the ADD button in the Installs tab of the Hub. There are options for recommended version, official release versions, and pre-releases versions which are still in alpha and beta testing. As a beginner, the Recommended Version can be chosen for its stability. (UnityTechnologies 2021.)



Unity Hub 2.4.5

unity

Projects

Learn

Community

Installs

Projects

ADD

NEW

Project Name	Unity Version	Target Platform	Last Modified
Glitch Garden D:\Repos\Unity2DProjects\Glitch Garden Unity Version: 2020.2.7f1	2020.2.7f1	Current platform	5 months ago
Example D:\Repos\Unity2DProjects\Example Unity Version: 2020.2.7f1	2020.2.7f1	Current platform	5 months ago
Interface D:\Repos\Unity3DProjects\Interface Unity Version: 2020.2.7f1	2020.2.7f1	Current platform	6 months ago
Argon D:\Repos\Unity3DProjects\Argon Unity Version: 2020.2.7f1	2020.2.7f1	Current platform	7 months ago
Boost			

Figure 11. Unity Hub Projects

In the Projects tab shown in figure 11 above, new projects can be created with the NEW button, and the arrow on the right is used for choosing version if multiple versions of Unity are installed. Users can also add their own existing project with the ADD button. When creating the project, naming and choosing the location of the project is possible. For this project, the game is called Glitch Garden, and at the time of writing this, Unity version 2020.2.7f1 is used. The 2D option on the Template Menu is also needed since the game created is in 2D. (UnityTechnologies 2021.)

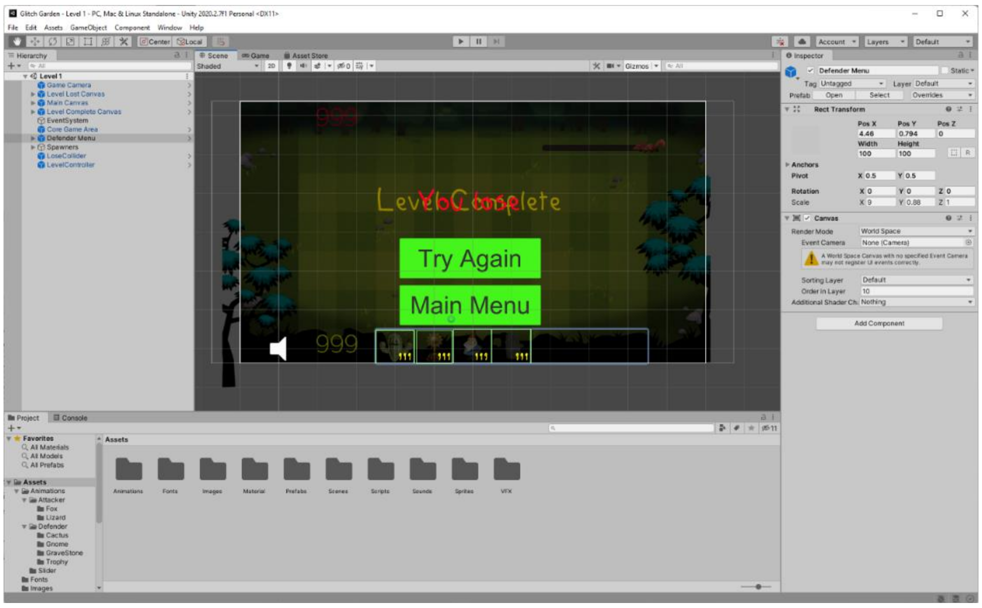


Figure 12. Basic Project Layout

The basic layout of a newly created project is as shown in Figure 12, with the Hierarchy, Inspector, Project Windows, and the Scene/ Game view. Figure 12 is what our project will look like after everything is done. All the files and folder that are required for the game can be seen in the Project View. The files can be placed anywhere in the Assets folder. However, it is good practice to have them in different folders based on their functionalities. (UnityTechnologies 2021.)

4.2 Background Playspace

Before going into this section, knowledge on aspect ratio is essential. The aspect ratio of an image is its ratio of its width to its height. It is commonly expressed in the format of width:height. The most commonly used ratio is currently 16:9. However, the 4:3 ratio is also very popular, so when the game is created, this needs to be taken into consideration. (UnityTechnologies 2021.)

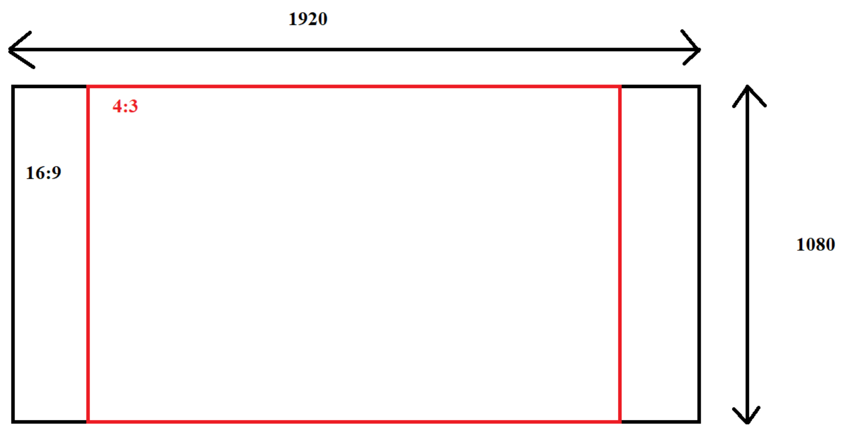


Figure 13. Aspect Ratio Changes

The figure 13 above is used for demonstration purpose, the size might not be 100 percent correct. The 4:3 ratio changes the width but not the height, meaning if the image has a resolution of 1920x1080(16:9 ratio) when we change it to 4:3 the resolution will become 1440x1080. The new width can be calculated by dividing the unchanged height 1080 by 3 and multiplies by 4. Now when we create our background playspace, as can be seen from the figure 13 above, and the calculation, our “Safe Zone” which is the area when it will be visible both in 16:9 and 4:3 ratio is 1440x1080. The excess area in 16:9 can be used purely for cosmetic purposes.

The first thing needed to be done is to create a new Canvas for our background. This can be done by right clicking on the Hierarchy Window then go down to UI then Canvas. Then an Image is created by right clicking on the Canvas that was just created then select UI and Image. The Image Component can be seen from the Inspector Window when the Image is selected. The desired image now needs to be dragged to the Source Image as seen from figure 14 below.

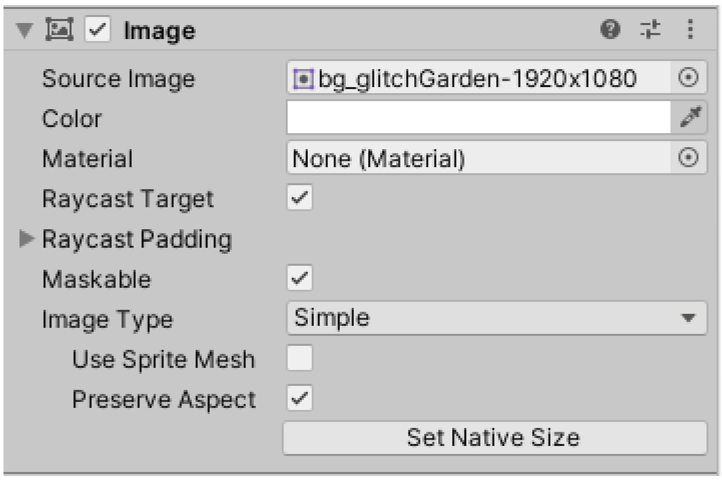


Image
Source Image bg_glitchGarden-1920x1080
Color
Material None (Material)
Raycast Target ✓
Raycast Padding
Maskable ✓
Image Type Simple
Use Sprite Mesh
Preserve Aspect ✓

Figure 14. Image Component

As good practice, all images used for the game should be in its separate folder. This will make the drag and drop process easier. After dragging the image into the Source Image, tick on Preserve Aspect and click on Set Native Size so the image will resize to fit the Canvas entirely.

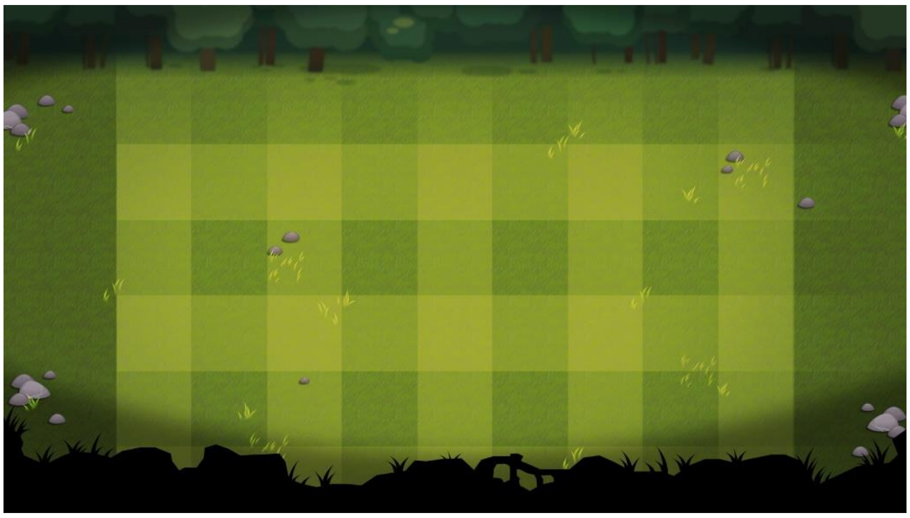
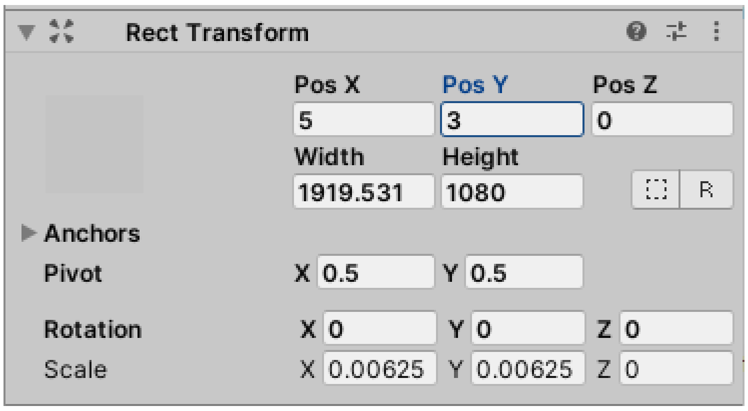


Figure 15. Background Playspace

The figure 15 above is the playspace shown in 16:9 ratio**.** The “Safe Zone” is the area where all the squares are located. During the game design process, decisions on how many lanes and how long they are to get the correct feeling for the game can be made. For this project, design decisions have already been made and chosen to have a total of 5 lanes from top to bottom, and each lane will be 9 squares in length. This will be the area where enemies and defenders can be spawned. The excess areas from the sides can be used for cosmetic purposes, and the excess areas from the top and bottom can be used for placing buttons and status bar.

Finally, the main camera needs to be realigned with the Canvas so the camera would be right on top of our canvas, resulting in a coordinate system. The first step is to change the Render Mode option in the Canvas Component by clicking on Canvas in the Hierarchy. World Space option is chosen. More details on the options can be found on the Unity Manual Website. Deeper discussion is not mentioned here to avoid confusion. For this project, one grass square shown in figure 15 to be one world unit is wanted, This means if the first square from the bottom left is chosen, its coordinate from the middle should be 1,1, the Z coordinate should not be modified as the project is in 2D.

After changing the Render Mode, the Canvas needed to be scaled down so it fits into the camera. Right now, the canvas has the size of 1920 width and 1080 height. This can be seen from the Rect Transform Component of the Canvas. These sizes are currently in world units, but the camera resolution is in pixels. Each of the grass square has a size of 160x160 pixels, the game has 12 squares in total (9 squares of playspace, and 3 squares of background cosmetic). As mentioned above, each square should be the size of 1 world unit, so the 1920 world units from the Canvas need to be scaled down to just 12. This can be done by dividing 12 by 1920, then set the Scale of option of X and Y in the Rect Transform to the result which is 0.00625.


Rect Transform

Pos X 5 | Pos Y 3 | Pos Z 0
Width 1919.531 | Height 1080
Anchors
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 0.00625 Y 0.00625 Z 0

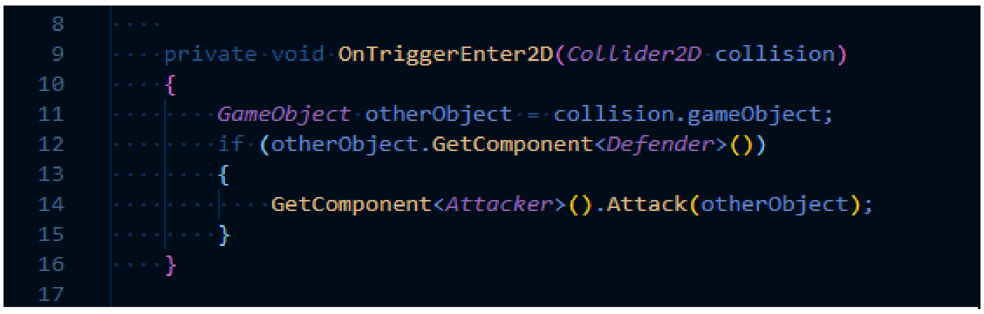
Figure 16. Canvas Scaling and Repositioning

Now that the Canvas is scaled down to fit the camera, the first square on the bottom left needed to be at the exact coordinate of 1,1. The Pos X and Pos Y coordinate in the Rect Transform determine where the middle of our GameObject is. Therefore, for the first square on the bottom left to be at the desired coordinate, the Canvas needs to be moved to the position as shown in Figure 16 above. Those numbers are aquirred from manually dragging the Canvas so that the first square is somewhere around the 1,1 position, then the number is rounded up. To test if this is correct, a new empty GameObject can be created then add in a random Sprite so it can be seen. Afterwards set its Pos X and Pos Y value to 1,1, if the GameObject is in the middle of the first square, the position is correct.

The last thing needed to be done is to move the Camera to match the Canvas. For the Pos X and Pos Y of the Camera would be exactly as the Canvas, changing the Size value of the camera is required. The default value is 5, which is 5 world units from the middle to the top and the middle to the bottom. We need to calculate how many world units do we have as height. This can be done by dividing 1080 to 160 (since our height is 1080 pixels, and each square has 160 pixels of height, and each square is equal to 1 world unit). The result is 6.75, we need to divide this by 2 then set the Size value in our Camera. Dividing by 2 is needed because our Size value is base from the middle to the top or bottom, our 6.75 is the value from top to bottom.

4.3 Attackers

All of the enemies in this project are Non-playable characters (NPCs). NPCs as their name suggested, are the character that are not controlled by the players. These characters instead behave according to the scripts that are attached to them. They have a certain set of actions and pattern that will trigger when the conditions, which are provided in the scripts. For this project, one common script for all the enemies and a different script for each type of enemies is presented. The enemies in this game will be referred to as "Attackers" in the scripts.



```

    private void OnTriggerEnter2D(Collider2D collision)
    {
        GameObject otherObject = collision.gameObject;
        if (otherObject.GetComponent<Defender>())
        {
            GetComponent<Attacker>().Attack(otherObject);
        }
    }

```

Figure 17. Lizard Attacker Script

In figure 17 above is the function that is in one of the many enemy scripts, the Lizard. This is a script specifically for the Lizard enemy type. This is the only function needed, the Start() and Update() function can be removed.

All of the characters in the game will have a collider attached to them. Colliders are just Unity Component so they are easily added through the Inspector window with the Add Component button. Colliders are used for event detection, in our case, the collision of the attacker and the defenders. As seen from the script in figure 17, the OnTriggerEnter2D function will be called whenever a collision happens, and the Lizard attacker will trigger an Attack() function if it collided with a GameObject of type Defender. The "collision" parameter only gets the collider of the GameObject, therefore, line 11 was needed to get the GameObject itself for comparison in line 12.

4.3.1 Attacker Animations

The process of creating an animation for the characters will not be discussed in this thesis, only how the animations trigger, and when they move from one state to another. Only the animations applied on the Lizard attacker is mentioned, the same principle can be applied for all other type of Attackers. Firstly, to be able to animate a character, the Animator Component is added to the character's GameObject. However, repeating this for every GameObject of the same type is extremely time-consuming and ineffective. Therefore, all developers use prefabs for any GameObject that need to be reusable. Unity's Prefabs system aids developers when reusing GameObjects by keeping all its configurations and components, keeping all the instances of that GameObject in sync. Changes made in the prefabs will be applied to all its instances across multiple Scenes in the project. A prefab can be created by dragging the GameObject into the Project Window. All instances of a prefab will blue colored text in the Hierarchy Window. It is good practice to have all the Prefabs in a separate folder in Project Window for easy management. (UnityTechnologies, 2021)

For our Lizard attacker, there are 3 animations, the Lizard Jump, Lizard Walk, and Lizard Attack. The Lizard Jump will be the initial animation, which is when the lizard attackers spawn. Since this action only happens once, looing is not needed for this animation, and the next animation will be triggered when the Lizard Jump ends. Transition duration and exit time and be set if required in the Inspector Window when clicking on the animation transitions. Transitions between animation state can be created by right-clicking on the state and choose "Make transition" as seen from figure 18 below.

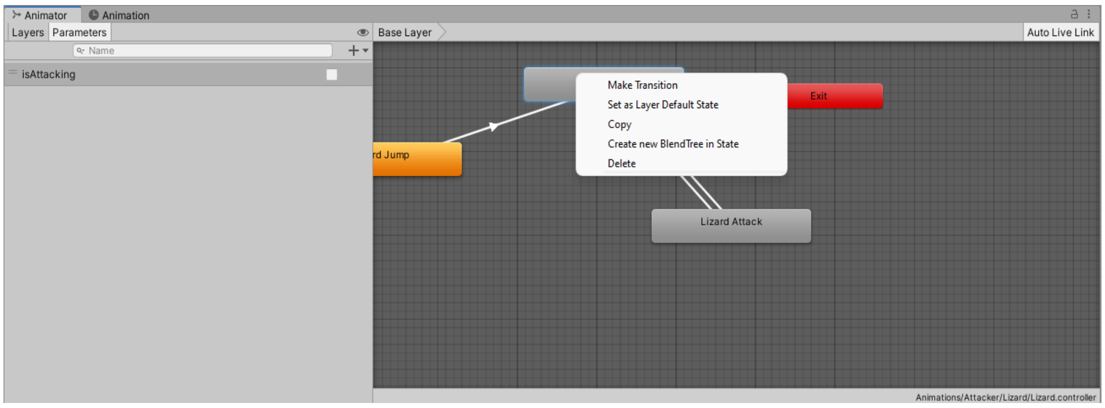
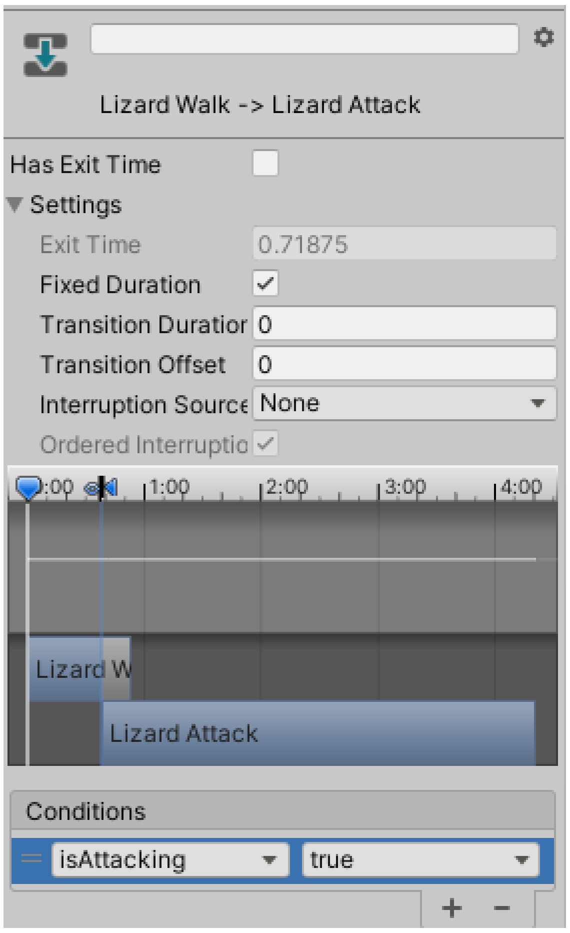


Figure 18. Animator Window

As shown in figure 17, the attacker will call a function whenever it collides with a defender. A function can be added as a condition to trigger the transition from the Lizard Walk to the Lizard Attack animation. Since the Lizard Walk animation will need to be looped as long as the attacker is still alive, waiting for the animation to end and transition to the next one like the Lizard Jump animation is not correct.

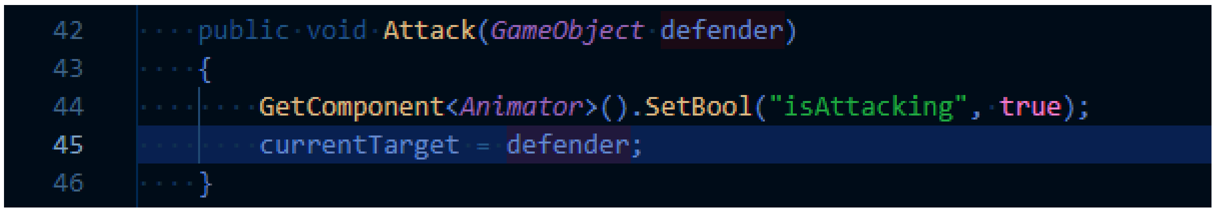

Lizard Walk -> Lizard Attack

Has Exit Time ☐
Settings
Exit Time 0.71875
Fixed Duration ✓
Transition Duratior 0
Transition Offset 0
Interruption Source None
Ordered Interruptio ✓

Conditions
isAttacking true

Figure 19. Transition Conditions

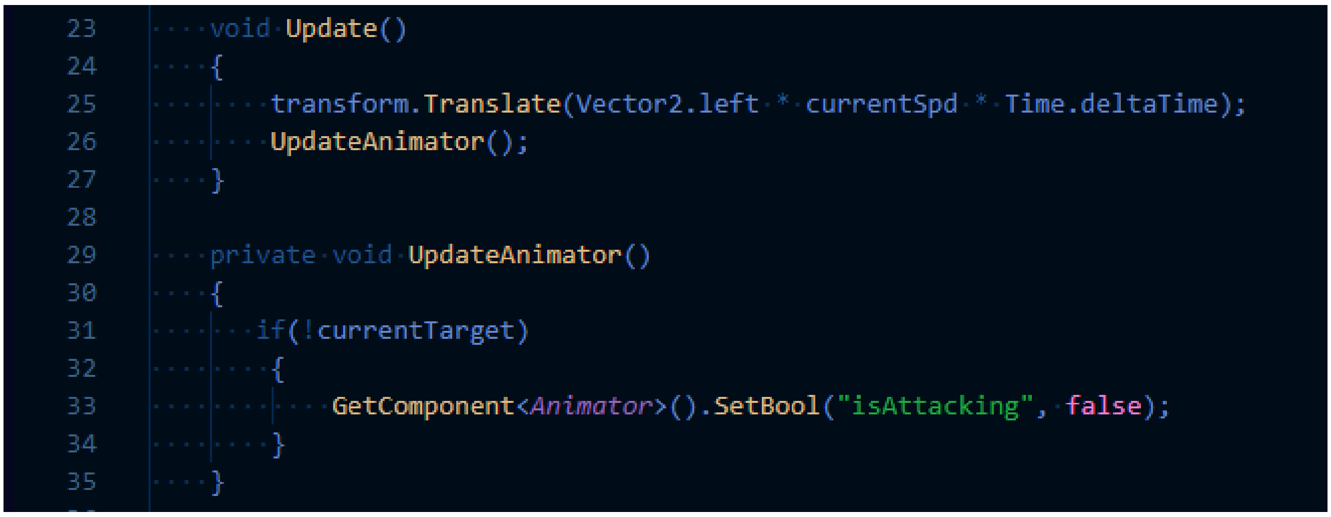
To create a condition for a transition, parameters must be added to the transition in the Animators Window. As seen in figure 18, parameters can be added by clicking on the “+” button in the parameters tab. The “isAttacking” parameter is already created which can be used as a condition in the Conditions tab as seen in figure 19. Parameters can be of type Float, Bool, Int or Trigger. The “isAttacking” parameter will determine which transition will act. In this case, as seen from figure 19, the Lizard Walk will transit to Lizard Attack when “isAttacking” is true. Since the effect of the attacker to start attacking as soon as it collides with the defender is desired, transition duration is set to 0.



```
42      public void Attack(GameObject defender)
43      {
44          GetComponent<Animator>().SetBool("isAttacking", true);
45          currentTarget = defender;
46      }
```

Figure 20. Attack Function

Figure 20 displays the content of the Attack() function mentioned above, this function will manipulate the “isAttacking” Boolean value mentioned above and changes the animation states. Line 45 will determine the current target of the Attacker. This will help with the damage dealing system, which will be discussed later in this thesis.



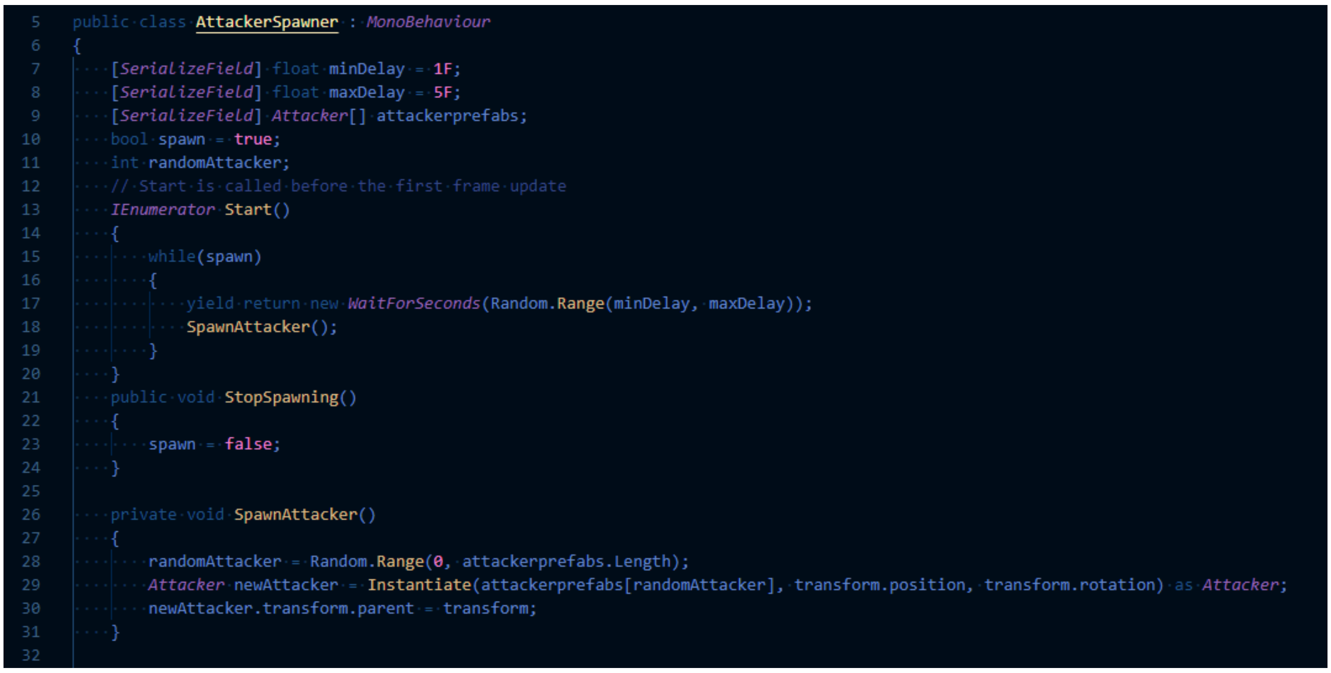
```
23      void Update()
24      {
25          transform.Translate(Vector2.left * currentSpd * Time.deltaTime);
26          UpdateAnimator();
27      }
28
29      private void UpdateAnimator()
30      {
31          if(!currentTarget)
32          {
33              GetComponent<Animator>().SetBool("isAttacking", false);
34          }
35      }
```

Figure 21. Update and Update Animator Function

The functions in figures 20 and 21 are from the common script for attacker, the “Attackers.cs” script. The “currentTarget” variable is set to null by default in the Attacker.cs script, which will set the value of “isAttacking” to false by default, as seen in line 31 from figure 21. When the attacker collides with the defender, “currentTarget” is assigned with the defender GameObject and the value of “isAttacking” is set to true. This value will return to null when the GameObject is destroyed. Since the UpdateAnimator is updated every frame, as soon as the defender GameObject is destroyed, the Attacker will resume to its walking animation state. Line 25 in figure 21 allows the Attacker to advance from right to left.

4.3.2 Attacker Spawner

There are 5 attacker spawners placed at the right corner of the 5 playable lanes in the game. We can determine which attacker to spawn in the AttackerSpawner.cs script attached to each of the spawners. Since we are reusing the Spawner object many times, a prefab of it should be created. Each prefab instance will have all its attributes synchronized. However, the transform component, which determines where the object is, is different. A Spawner is just a GameObject with the AttackerSpawner.cs script attached to it. It is recommended to create a parent object for all the Spawners. Each of the spawners from the lanes will be the children of the Spawners object, and each attacker created by that spawner will be the child of that Spawner object.



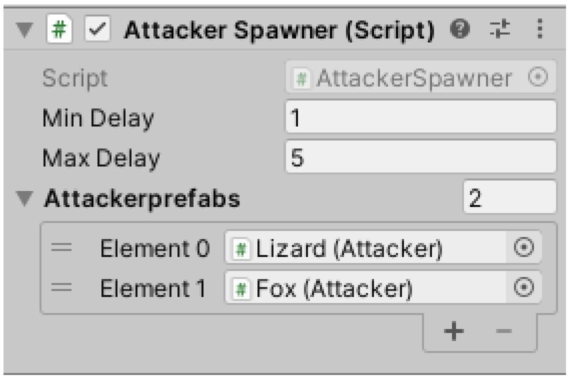
```
public class AttackerSpawner : MonoBehaviour
{
    [SerializeField] float minDelay = 1F;
    [SerializeField] float maxDelay = 5F;
    [SerializeField] Attacker[] attackerprefabs;
    bool spawn = true;
    int randomAttacker;
    // Start is called before the first frame update
    IEnumerator Start()
    {
        while(spawn)
        {
            yield return new WaitForSeconds(Random.Range(minDelay, maxDelay));
            SpawnAttacker();
        }
    }
    public void StopSpawning()
    {
        spawn = false;
    }

    private void SpawnAttacker()
    {
        randomAttacker = Random.Range(0, attackerprefabs.Length);
        Attacker newAttacker = Instantiate(attackerprefabs[randomAttacker], transform.position, transform.rotation) as Attacker;
        newAttacker.transform.parent = transform;
    }

```

Figure 22. AttackerSpawner Script

Figure 22 above display the entirety of the AttackerSpawner.cs script. The SerializedField option in line 7,8 and 9 is for easier debug purpose. A variable that is a serialized field will have its value shown in the Script Component in the Inspector Window as visualized in figure 23 below.


Attacker Spawner (Script)

Script	AttackerSpawner
Min Delay	1
Max Delay	5
Attackerprefabs	2
Element 0	Lizard (Attacker)
Element 1	Fox (Attacker)

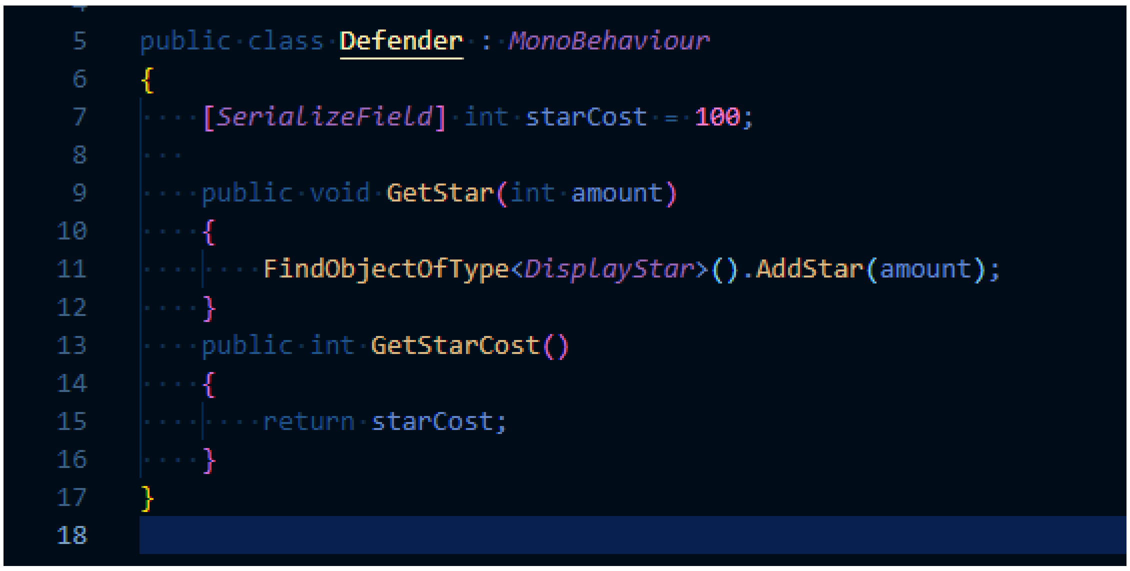
Figure 23. Script Component

The aim is to spawn an attacker with a random delay from one to five seconds, hence the minDelay and and maxDelay variable in line 7 and 8. These values can be modified in the Inspector to fit our liking. Line 13 in figure 22 is the declaration of coroutine, which is needed because we do not want the spawner to just spawn once, or frame dependant. The yield return statement is required somewhere in the body of the function that is declared with the IEnumerator return type. Line 17 in the AttackerSpawner.cs script satisfy this requirement and is also responsible for the random delay time between each attacker spawns. Figure 23 shown 2 possible type of attackers that will be spawned, the lizard, and the fox. More attacker types can be added by clicking on the "+" button and drag the desired attacker prefab to that new element. Since the Start() function is already turned into a coroutine, the SpawnAttacker() function will be called repeatedly as long as the spawn variable declared in line 10 is true. The spawner will stop when the StopSpawning() function is called, which is not in the AttackerSpawner.cs script but in the LevelController.cs script because we want to have overall control on the game level.

As seen from figure 22, in the SpawnAttacker() function, line 28 is responsible for the randomness of the attackers spawned. We can spawn multiple type of attackers in a lane, or just one type since the value of randomAttacker variable is based on the size of our attackerPrefabs array. This value is used as an index of the attackerPrefabs array for the first parameter of the Instantiate() function in line 29. The newAttacker is of type Attacker so the instance that was create should be casted as Attacker to avoid unnecessary errors. The second and third parameter of the Instantiate() function is to determine where the attacker should be created, so transform.position and transform.rotation is used. This is the position and rotation of the object that has this script attached. Finally, line 30 is to instantiate the newAttacker object as a child of the Spawner object by assigning its parent’s transform value to the Spawner transform value. The purpose of this is to assist on detecting if a lane has an Attacker, which is required for the Defenders to start shooting. If there is an attacker spawned in a lane, that lane’s spawner will have a child of type Attacker, which is the condition for the Defenders to start firing.

4.4 Defenders

In this section, I will discuss on how and where to spawn a Defender for the game. A defender’s purpose can be both shooting to destroy the attackers or blocking the attackers’ path. Spawning defenders should cost some sort of resources; therefore, resource system needs to be implemented. The resource used in this game is referred to as “Stars” in the game scripts. Defender animation will not be discussed since it follows the same principles as the Attacker animation. However, animation event will be mentioned as it is required in our resources system.



```
5   public class Defender : MonoBehaviour
6   {
7       [SerializeField] int starCost = 100;
8       
9       public void GetStar(int amount)
10      {
11          FindObjectOfType<DisplayStar>().AddStar(amount);
12      }
13      public int GetStarCost()
14      {
15          return starCost;
16      }
17  }
18
```

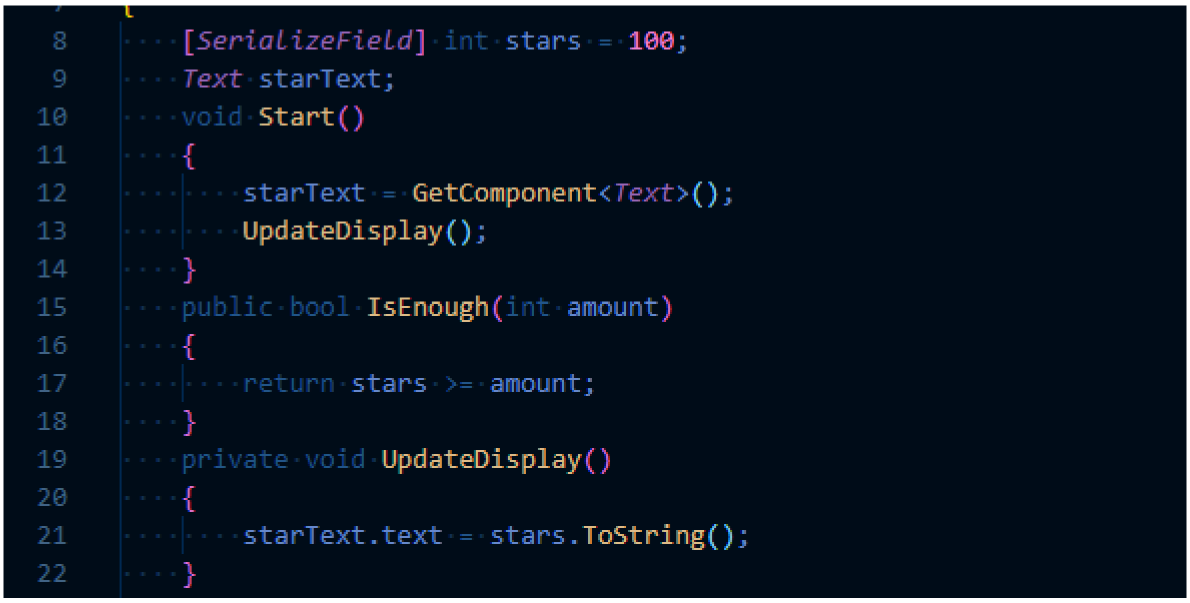
Figure 24. Defender Script

Figure 24 above is the entirety of the defender.cs script. The starCost variable is serialized so it can be changed with ease when the script is placed in a different defender. The function GetStar from line 9 to 12 is to increase the resources that the player has which will be discussed later in the Resources section. The GetStarCost function is simply to get the cost to spawn a defender, which is the starCost variable. Using a getter function like this will avoid undesirable changes to the variable it is getting.

Similar to Attackers, all defenders have a Defender.cs script shared, while each different type of defender can have their own exclusive scripts. For the game mentioned in this thesis, defenders are categorized into Offensive, Defensive, and Utility. The Offensive defenders share their Shooter.cs and Projectile.cs scripts since they are all ranged defenders. Defensive and Utility defenders do not have their exclusive scripts yet since this is only a simple project. More features can be added if the need ever rises.

4.4.1 Resources

The main resource of the game is Star. Each defender will require a different number of stars to spawn. One of our Utility defenders, the Trophy defender, is responsible for generating stars after a period of time. More interesting and unique ways of spending and acquiring stars can be added into the game as improvements; however, for the simplicity of this thesis, the Trophy defender will be the only one generating stars.

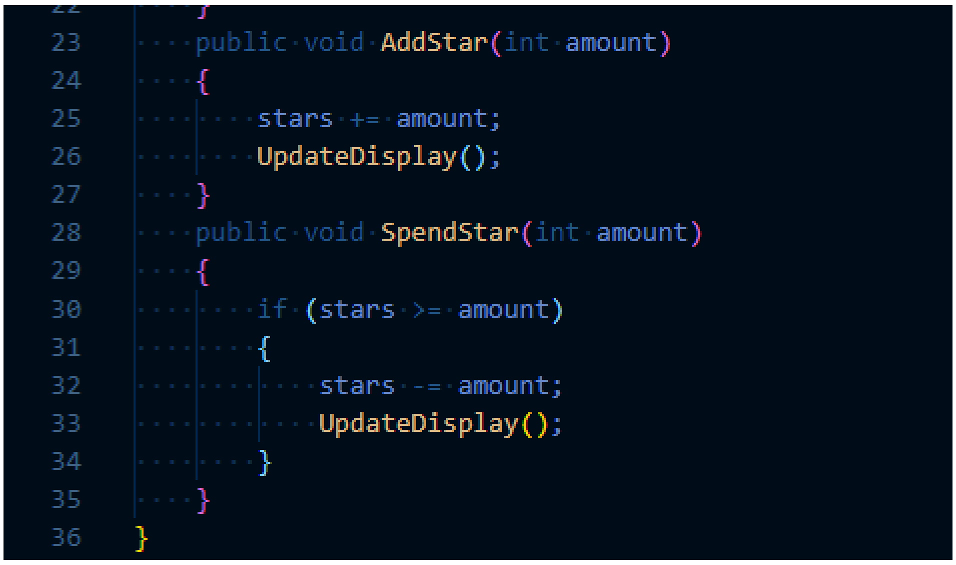


```
 8      [SerializeField] int stars = 100;
 9      Text starText;
10      void Start()
11      {
12          starText = GetComponent<Text>();
13          UpdateDisplay();
14      }
15      public bool IsEnough(int amount)
16      {
17          return stars >= amount;
18      }
19      private void UpdateDisplay()
20      {
21          starText.text = stars.ToString();
22      }
```

Figure 25. DisplayStar Script

Figure 25 contain 2 additional functions of the DisplayStar.cs script besides the Start function. The purpose of the starText variable and the UpdateDisplay function is to visualize the amount of stars the player currently possesses. A Text component is added the in game screen, the Canvas, to display the current stars. Line 21 is to convert the stars variable into string so it can be shown in the Text component on the Canvas. The UpdateDisplay function is called whenever we make changes to the value of the star variable. The IsEnough function from line 15 to 18 is to determine if the player have enough resources to spawn the selected defender. This function takes the parameter “amount” of type int, this is the cost to spawn the selected defender. Line 17 will return a True value if the player currently has enough stars for that defender, else it will return False.

In the previous section, I have mentioned the function from the DisplayStar.cs to generate stars. The AddStar function from figure 26 below is also from the DisplayStar.cs script. This is simply a function that adds more value to the stars variable every time it is called. For this function to be called after a period of time, we need to use Animation Event.



```
    }
    public void AddStar(int amount)
    {
        stars += amount;
        UpdateDisplay();
    }
    public void SpendStar(int amount)
    {
        if (stars >= amount)
        {
            stars -= amount;
            UpdateDisplay();
        }
    }
}
```

Figure 26. DisplayStar Script 2

Each defender has an Idle animation. Offensive and Defensive can have additional animations when they encounter an Attacker. However, the Utility defenders which is the Trophy defender, only required the Idle animation. To use Animation Event as a trigger to call the function, going through the frames of the Trophy’s Idle Animation is required.

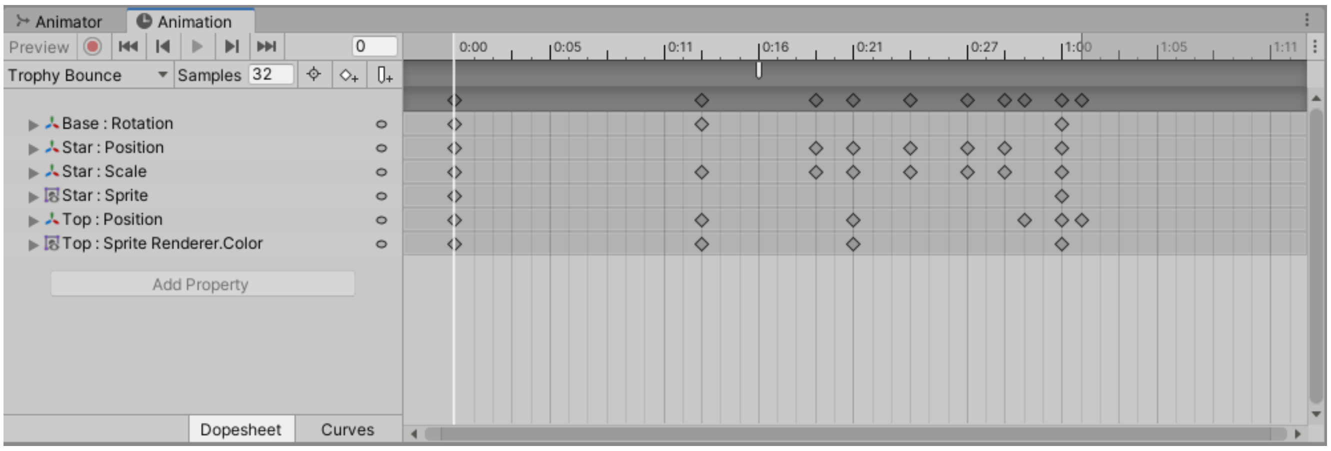
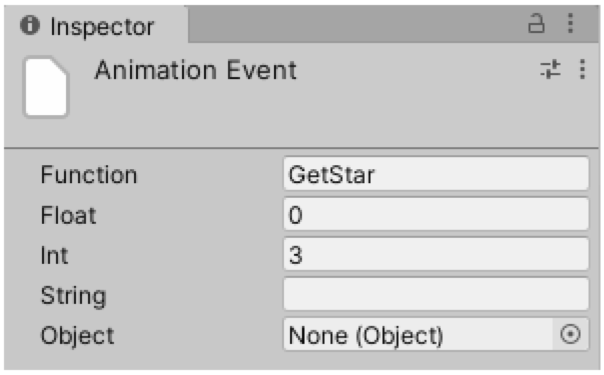


Figure 27. Trophy Animation

Figure 27 displays the idle animation of the Trophy defender, which is called Trophy Bounce. The white mark under the 0:16 point is where I have placed the animation event. To add an animation event, click on the last icon to the right on the same line of the animation's name. The event can be placed anywhere within the frames of the animation, I find it the area around the middle suitable for this animation.


Inspector

Animation Event

Function	GetStar
Float	0
Int	3
String	
Object	None (Object)

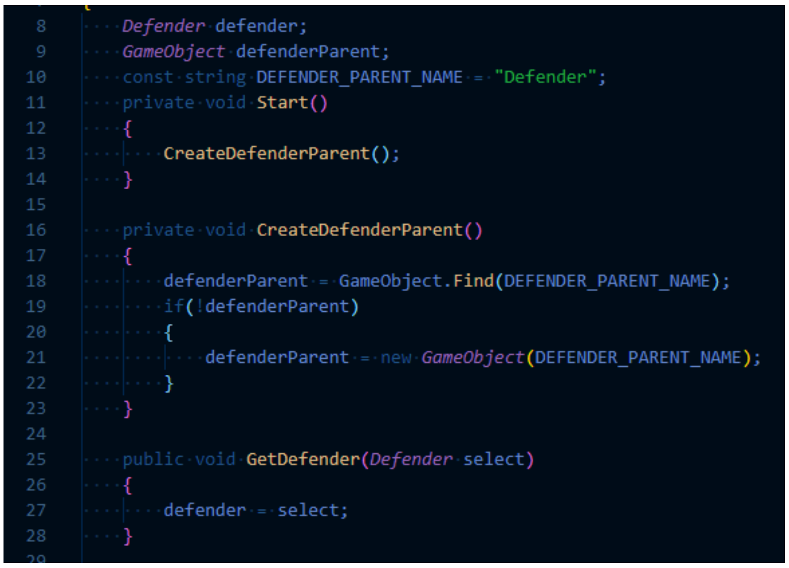
Figure 28. Animation Event

Figure 28 is the Animation Event window viewed from the Inspector when we click on the event from the frames. In older version of Unity, only the Function field is visible. The other fields will only appear once the function from the script is selected. The functions available are the functions from the Defender.cs script because the Defender.cs script is a Script Component in the parent of the Trophy defender object. In newest version, all fields are visible, the name of the function is typed in manually. The function suitable for calling is the GetStar function from figure 24, line 9. This function will find the DisplayStar object and call for the AddStar function, which will generate an amount of stars to the player.

This event is called every Trophy Bounce animation, so we have a mechanic of generating stars over time. The more Trophy defender a player spawns, the more stars they get.

4.4.2 Spawning Defenders

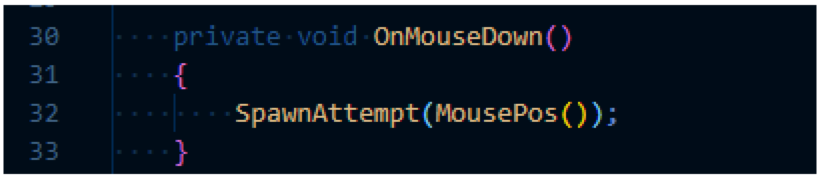
With the resources system in the previous section functioning properly, we have the prerequisites for spawning a defender. The defenders will be spawned on click; therefore, decision on which part of the game playspace is available for this purpose is needed. A box collider is required to limit the game area where you can place a defender. The first step is to create a simple box collider, resize it properly to fit the game playspace, and attach the DefenderSpawner.cs script to it.



```
 8      Defender defender;
 9      GameObject defenderParent;
10      const string DEFENDER_PARENT_NAME = "Defender";
11      private void Start()
12      {
13          CreateDefenderParent();
14      }
15
16      private void CreateDefenderParent()
17      {
18          defenderParent = GameObject.Find(DEFENDER_PARENT_NAME);
19          if(!defenderParent)
20          {
21              defenderParent = new GameObject(DEFENDER_PARENT_NAME);
22          }
23      }
24
25      public void GetDefender(Defender select)
26      {
27          defender = select;
28      }
```

Figure 29. DefenderSpawner Script

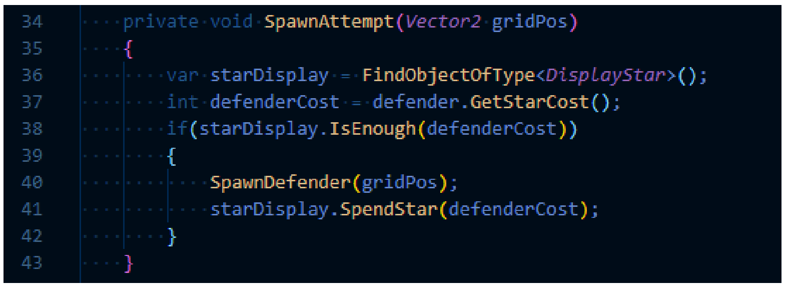
The same parent and child relationship from Attacker is also applied to the Defenders to avoid any unnecessary errors. Since the defenders do not have a fixed spawner like the attackers, the parent object of the defenders needs to be created manually. The CreateDefenderParent function from line 16 to 23 in figure 29 is responsible for this purpose. All defenders will be grouped into 1 parent object, the "Defender" object. The function simply checks if that object existed yet and create 1 if it is not. The GetDefender function from line 25 is to set the current defender variable to the one selected by the players from the defender menu. I will discuss more about the menu in the Selection UI section.



```
    private void OnMouseDown()
    {
        SpawnAttempt(MousePos());
    }
```

Figure 30. Spawn On Click Funtion

To spawn a defender on a mouse click, the OnMouseDown function is needed. This is a built-in function from Unity libraries, the name of the function must be these exact words. The function for spawning defenders will be identical to the one for attackers; however, a SpawnAttempt function as seen from figure 30 above is needed. Checking if the player has enough resources before spawning the selected defender is required. The MousePos function as parameter of the SpawnAttempt function is to determine where the player is clicking and spawning the defender into the corresponding grid.



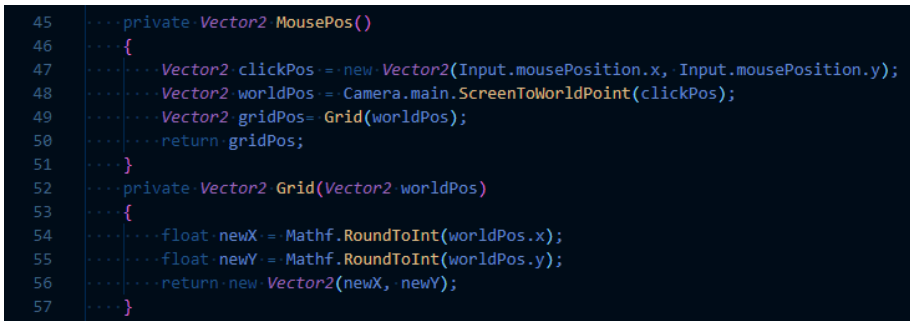
```
34      private void SpawnAttempt(Vector2 gridPos)
35      {
36          var starDisplay = FindObjectOfType<DisplayStar>();
37          int defenderCost = defender.GetStarCost();
38          if(starDisplay.IsEnough(defenderCost))
39          {
40              SpawnDefender(gridPos);
41              starDisplay.SpendStar(defenderCost);
42          }
43      }
```

Figure 31. SpawnAttempt Function

Figure 31 is the content of the SpawnAttempt function. The GetStarCost function is required in the Defender.cs script from figure 24 to assign the cost of that selected defender. The variable starDisplay is also require since we need to update the star value once the defender is spawned. Line 38 checks if the player currently have enough stars to spawn the selected defender. If the requirement if met, SpawnDefender function is called, and the star value is reduced by the value of defenderCost. The gridPos parameter in line 40 is the fixed position in each individual square of the game playspace, which will be discussed in the next section. This parameter is passed into the SpawnDefender function from the MousePos function in figure 30. The SpawnAttempt function takes the MousePos function as its parameter, and the MousePos function returns a value, which is the gridPos parameter in line 34 and 40 in figure 31.

4.4.3 Placing the Defenders

The content of the OnMouseDown function from the previous section is sufficient to spawn a defender at where the player clicks. However, we want the defender to be spawned at a fixed position inside each of the game playspace square in figure 15. If the player clicks anywhere inside one of the squares, a defender should spawn only at 1 position in the square.



```
45      private Vector2 MousePos()
46      {
47          Vector2 clickPos = new Vector2(Input.mousePosition.x, Input.mousePosition.y);
48          Vector2 worldPos = Camera.main.ScreenToWorldPoint(clickPos);
49          Vector2 gridPos= Grid(worldPos);
50          return gridPos;
51      }
52      private Vector2 Grid(Vector2 worldPos)
53      {
54          float newX = Mathf.RoundToInt(worldPos.x);
55          float newY = Mathf.RoundToInt(worldPos.y);
56          return new Vector2(newX, newY);
57      }
```

Figure 32. MousePos and Grid Function

The functions from figure 32 above are also members of the DefenderSpawner.cs script. The clickPos variable on line 47 is the exact position of the mouse click in screen units. All the position variables in this script are of type Vector2 because we are working on a 2D game, the third axis of the vector is not required. After acquiring the position of the mouse click in screen units, we need to convert it into world units because our playspace is aligned in world units. Line 48 will convert the position into world units as float value. The ScreenToWorldPoint is a built-in function, so we do not have to do this manually. The world position is then snapped into the correct position in the square using the Grid function in line 49. The Grid function is declared in line 52 and takes the worldPos from line 48 as its parameter. Since the ScreenToWorldPoint function converts the position into a float value, the values are rounded on line 54 and 55. This ensure that the same rounded value is achieved by clicking anywhere on that specific square. The Grid function then returns a new position value, which is also the return value of the MousePos function as seen on line 50 in figure 32.

4.4.4 Defender Selection UI

With the defenders spawning and the resources system both working properly, we can now create a menu for the list of the defenders available for the game. Since we want to click to select a defender to spawn, a collider is required. A hierarchy is recommended for easier management. For this project, I have created a parent menu object which has each of the defender type as its children. Each child has its own child component of type Text to display their star cost value.

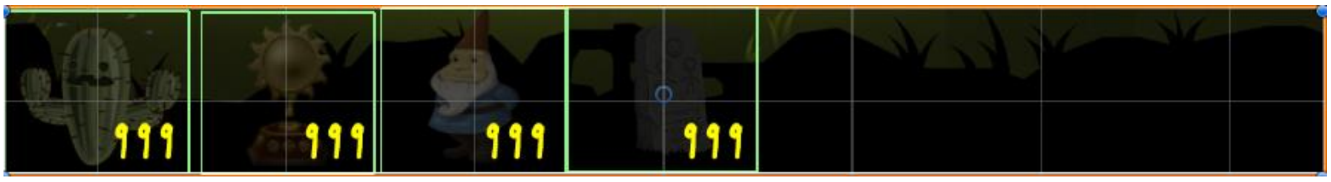
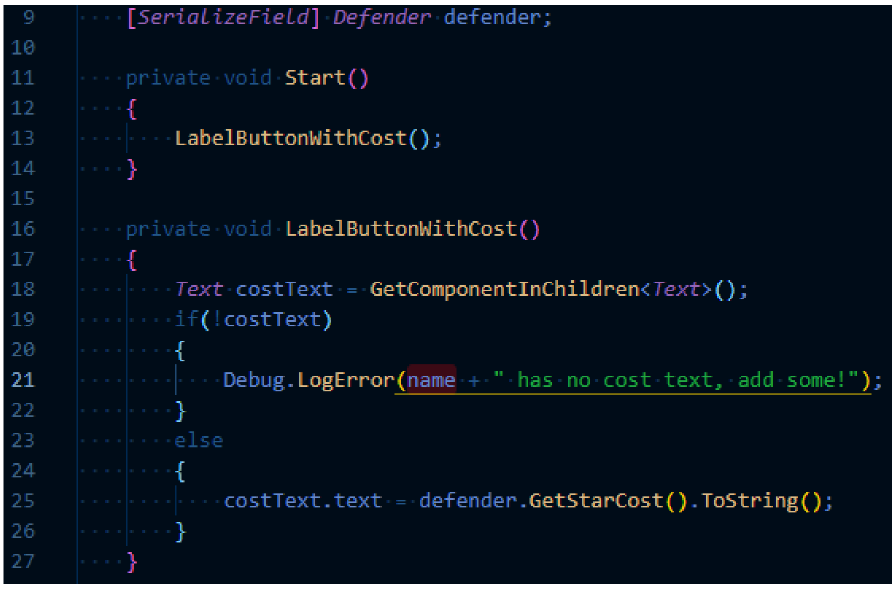


Figure 33. Defender Menu

Figure 33 above is how the menu looked like in development mode. Each defender in the menu has their own collider to detect the click action. A sprite renderer component is also required to display the visual representation of that defender. The 999 values are just for clearer visual in development, the values will be changed by the script attached to each of the defender.



```
 9      [SerializeField] Defender defender;
10
11      private void Start()
12      {
13          LabelButtonWithCost();
14      }
15
16      private void LabelButtonWithCost()
17      {
18          Text costText = GetComponentInChildren<Text>();
19          if(!costText)
20          {
21              Debug.LogError(name + " has no cost text, add some!");
22          }
23          else
24          {
25              costText.text = defender.GetStarCost().ToString();
26          }
27      }
```

Figure 34. Menu Buttons Script

Figure 34 above contains the functions of the MenuButtons.cs script. This script is attached to each of the child component of the Menu object, which is the defender object. Since each different defender contain this same script, a serialized field of type Defender is created for easy drag and drop in the Inspector Window. Developers can simply drag the prefab of the desired defender and drop it into the defender field in the Inspector window of the selected object that has this script attached, the defender variable on line 9 will take that dragged prefab as its value. The LabelButtonWithCost function is to get the cost of the defender that was dragged into this object. Line 19 to 22 in the script is only for debugging purpose, to check if there might be unexpected errors.

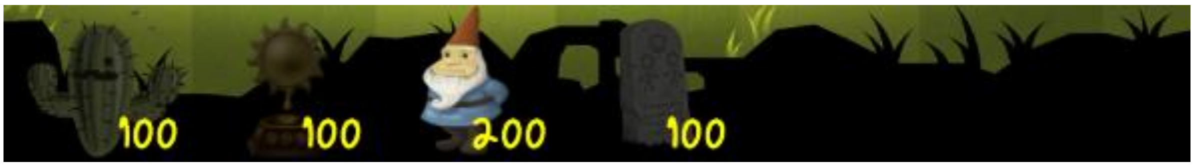
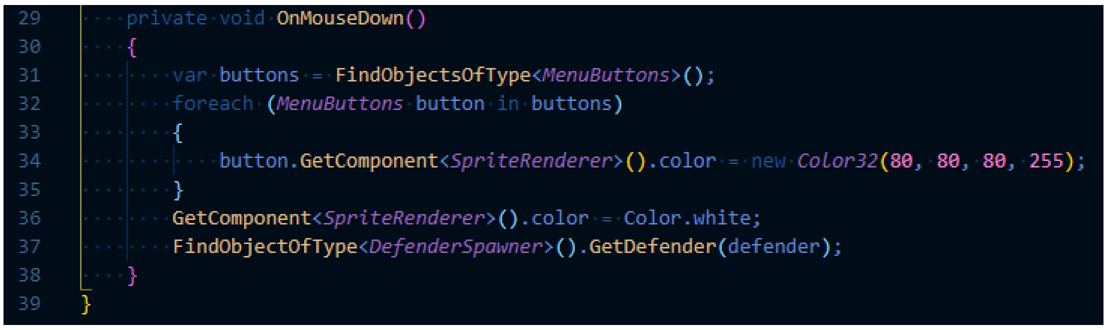


Figure 35. Defender Menu (Game Mode)

Figure 35 above is how the defender menu would look like while playing the game. The star cost value is set correctly thanks to the script in figure 34, and only the selected defender has it colour value set to white. The others is in a darker shade indicating that they are not selected.



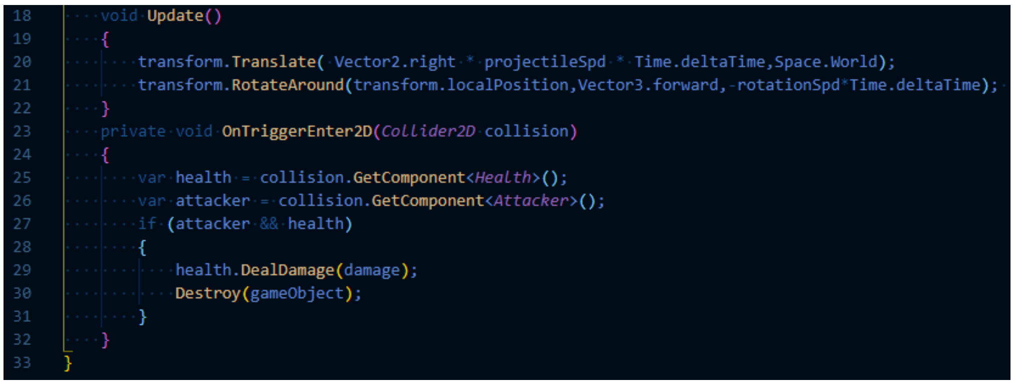
```
29      private void OnMouseDown()
30      {
31          var buttons = FindObjectsOfType<MenuButtons>();
32          foreach (MenuButtons button in buttons)
33          {
34              button.GetComponent<SpriteRenderer>().color = new Color32(80, 80, 80, 255);
35          }
36          GetComponent<SpriteRenderer>().color = Color.white;
37          FindObjectOfType<DefenderSpawner>().GetDefender(defender);
38      }
39  }
```

Figure 36. Menu Buttons Script 2

Figure 36 above is the remaining function of the MenuButtons.cs script. This is the same function mentioned in the defender spawning section since we are doing the same action, on click. As default, we have all set the buttons colour to a dark shade, the RBGA colour of 80, 80, 80, 255. Line 36 will change the colour of the selected defender to its original white colour. However, when the player clicks on one of the defenders, the remaining defenders' icons should turn back into its darker shade. Line 31 to 35 fulfilled this task. Whenever this function is called, the buttons variable on line 31 will look for all objects of type MenuButtons, then loops through each one of those buttons and turn them into their darker shade before changing the selected button into white on line 36. Finally, line 37 will determine which defender the player is selecting so when they click on the playspace area that corresponding defender will spawn.

4.4.5 Defender Projectiles

The final section of the defenders will be on their projectiles, how it detects the existent of an attacker and start firing. The projectiles objects will have their own Projectiles.cs script attached, and any defenders that has the ability to shoot will have their Shooter.cs script to initiate the projectiles.



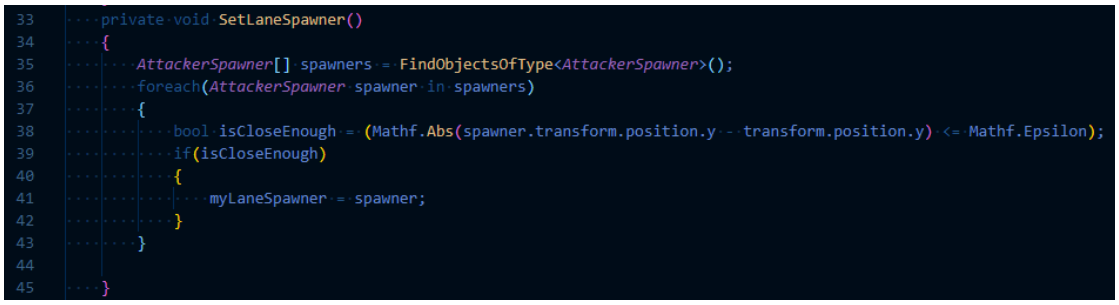
```
18      void Update()
19      {
20          transform.Translate( Vector2.right * projectileSpd * Time.deltaTime,Space.World);
21          transform.RotateAround(transform.localPosition,Vector3.forward,-rotationSpd*Time.deltaTime);
22      }
23      private void OnTriggerEnter2D(Collider2D collision)
24      {
25          var health = collision.GetComponent<Health>();
26          var attacker = collision.GetComponent<Attacker>();
27          if (attacker && health)
28          {
29              health.DealDamage(damage);
30              Destroy(gameObject);
31          }
32      }
33  }
```

Figure 37. Projectile Script

Only the Update function in the Projectile.cs script is used in our game, the Start function can be removed. Our projectile has 3 serialized variables, the projectileSpd, rotationSpd, and damage. These variables contain value as they are named. As seen from figure 37, the projectile position and rotation is updated every frame, creating the motion of it flying towards the attackers. Line 20 determine the left to right movement of the shot, the Time.deltaTime is to make sure the movement is not frame-dependent. On line 21, the parameters of the RotateAround function are self-explainatory. The rotation is happening on the object, so the transform needs to take the localPosition value. The third parameter, -rotationSpd*Time.deltaTime, the value is negative because of the direction of the spin. These values can be modified to fit the developer intentions.

Secondly, since each projectile has their own collider component attached, the OnTriggerEnter2D function can detect if the object is in contact with an attacker. Upon collision, the projectile will assign the health and attacker component of the attacker as seen on line 25 and 26 of figure 37. If both of these values exist, meaning the attacker is still alive with some health point, the projectile will deal damage by calling the DealDamage function from the Health and Damage system. The projectile is then destroyed.

In addition to the Projectile.cs script, we also need the Shooter.cs script on the Offensive defenders so they can trigger the firing event and start launching projectiles. The Start function of the Shooter.cs script is similar to the DefenderSpawner.cs script. A parent object is created for the launched projectiles. An additional animator variable is needed since we want to trigger the attacking animation of the defender. Very similar to the attackers in section 4.3, the shooters will also update the isAttacking Boolean value every frame to check if there is an attack on their lane. If this value is true, the attack animation of the defender is triggered.



```
33      private void SetLaneSpawner()
34      {
35          AttackerSpawner[] spawners = FindObjectsOfType<AttackerSpawner>();
36          foreach(AttackerSpawner spawner in spawners)
37          {
38              bool isCloseEnough = (Mathf.Abs(spawner.transform.position.y - transform.position.y) <= Mathf.Epsilon);
39              if(isCloseEnough)
40              {
41                  myLaneSpawner = spawner;
42              }
43          }
44
45      }
```

Figure 38. SetLaneSpawner Function

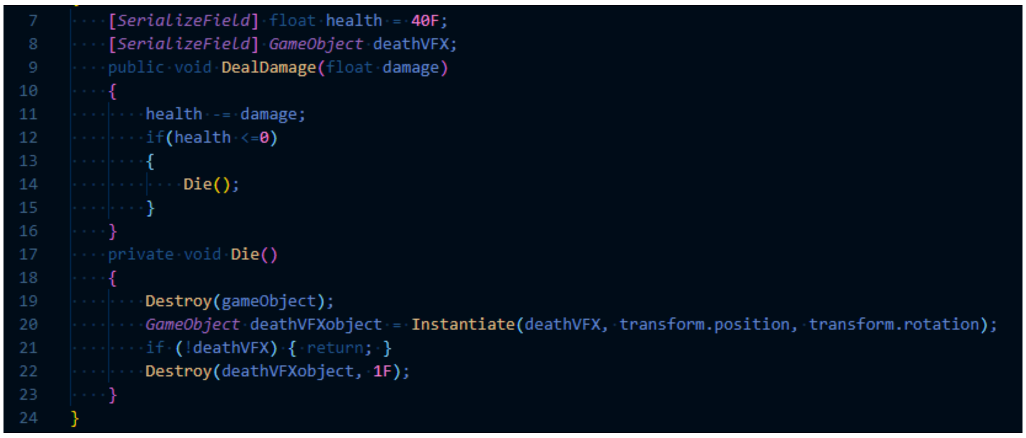
The Shooter.cs script has a variable myLaneSpawner, which will determine if there is an attacker on their lane or not. Figure 38 above is the SetLaneSpawner function in the Shooter.cs script. On line 35, this function will look for all the existing attacker spawners and put them into the “spawners” array. From line 36 to 42, the function will loop through all the spawners that it found and check if that spawner is on the defender’s lane with the isCloseEnough variable. The Abs function on line 38 is to make sure the value is always equals or higher than 0. The code from line 38 checks if the Y position of the spawner is the same as the Y position of the defenders by comparing the subtractions of both value to Epsilon. The Mathf.Epsilon is the closet value to 0 possible. Using Mathf.Epsilon instead of 0 is recommended to avoid unnecessary errors. If the subtraction of the two Y value is smaller than Epsilon, this means the value are basically the same, meaning the attacker spawner and the defender is on the same lane. The myLaneSpawner variable is now assign with the value of that spawner.

As mentioned at the end of section 4.3.2, each attacker that is created from a spawner will become that spawner’s child. To check if there is an attacker on the defender that has the Shooter.cs script attached lane, we just need to call the isAttackerInlane function from the Shooter.cs script and check the myLaneSpawner variable if it has any children object. If the children count is more than 0, the Update function will return true value for the isAttacking Boolean value.

The final function of the Shooter.cs script is the Fire function; this function instantiates an object of type Projectile. The code logic is identical to the SpawnAttacker and SpawnDefender function. Each instance of a projectile is also set to become a child of the Projectile parent component.

4.5 Health and Damage System

The heal and damage system is simple and straight-forward, each attacker and defender has a certain health value, attackers and offensive defenders also have damage value. Each time the deal damage to each other by collision, health is reduced by the value of damage.



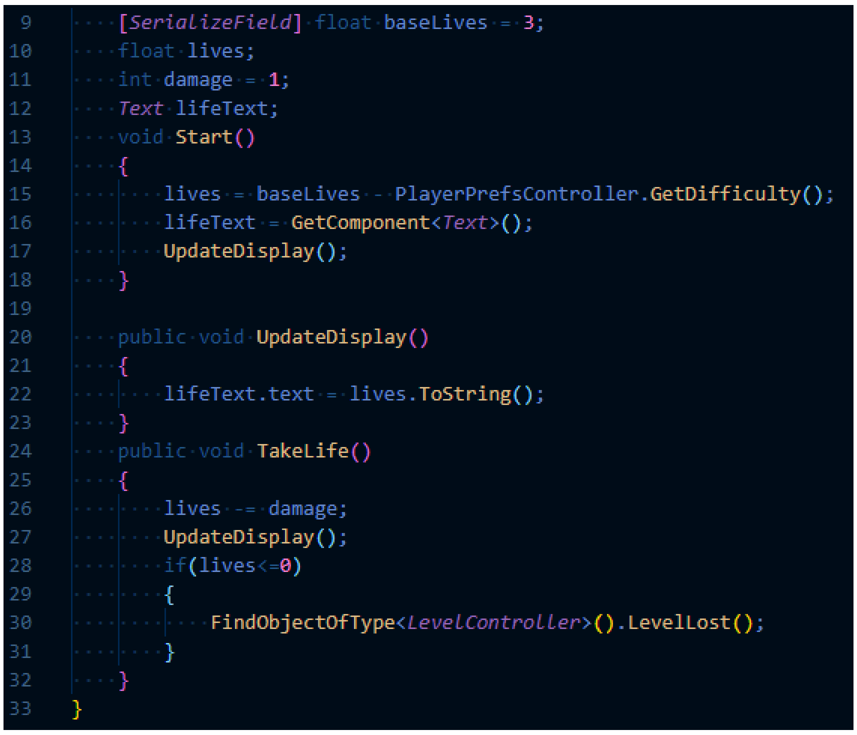
```
 7      [SerializeField] float health = 40F;
 8      [SerializeField] GameObject deathVFX;
 9      public void DealDamage(float damage)
10      {
11          health -= damage;
12          if(health <=0)
13          {
14              Die();
15          }
16      }
17      private void Die()
18      {
19          Destroy(gameObject);
20          GameObject deathVFXobject = Instantiate(deathVFX, transform.position, transform.rotation);
21          if (!deathVFX) { return; }
22          Destroy(deathVFXobject, 1F);
23      }
24  }
```

Figure 39. Health Script

Figure 39 above is the content of the Health.cs script, which is attached to all the defenders and attackers. The health variable is serialized for easier modification and testing purposes. The deathVFX variable is for the visual effects when the object is destroyed, attacker or defender. This is completely optional. The DealDamage function, which has been used in the attacker and the projectile scripts many times, is simple. The function reduces the health value by the value of damage, if health is less than or equal to 0, the Die function is called. This function does as its name suggested, the gameObject “dies”, so the built-in function Destroy is called. Line 20 to 22 is to activate the death visual effects.

4.6 Player Health

The previous sections have covered the fundamentals for the NPCs, a player health mechanic is needed for a better losing experience. For the game mentioned in this thesis, if the attackers manage to get through all the defender and reach the left side of the area, the player loses 1 health point for each attacker that break through. Firstly, a collider is required at the far left of the game area to detect collision when the attacker reached it. The collider needs to cover all 5 lanes of the game, and it has a LoseCollider.cs script attached to it.



```
 9      [SerializeField] float baseLives = 3;
10      float lives;
11      int damage = 1;
12      Text lifeText;
13      void Start()
14      {
15          lives = baseLives - PlayerPrefsController.GetDifficulty();
16          lifeText = GetComponent<Text>();
17          UpdateDisplay();
18      }
19
20      public void UpdateDisplay()
21      {
22          lifeText.text = lives.ToString();
23      }
24      public void TakeLife()
25      {
26          lives -= damage;
27          UpdateDisplay();
28          if(lives<=0)
29          {
30              FindObjectOfType<LevelController>().LevelLost();
31          }
32      }
33  }
```

Figure 40. LivesDisplay Script

The LoseCollider.cs script only has 1 function, the OnTriggerEnter2D, which will call the TakeLife function from the LivesDisplay.cs script in figure 40 above and destroy the gameObject of the attacker that contacted the lose collider. The LivesDisplay.cs script is almost identical to the DisplayStar.cs in term of functionality. We can duplicate the Text component from the stars in the Canvas and rename it to lives and replace the script. The TakeLife function would reduce the player health base value every time it is called, if that value reached 0, a LevelLost function from the level controller is called and the game is over. Line 15 from figure 40 is to reduce the amount of base lives the player has and create a game difficulty system. This will not be discussed in this thesis.

4.7 Level Design

Level design as its name suggested, is more based on how to design an interesting level rather than the technical aspect, so it will not be discussed in detail. However, the base line of a level is required. This section will be on how to create a level timer for each round, and a level controller to manipulate the game when the player wins or loses.

4.7.1 Level Timer

For level timer, we need to use the Slider component from the UI. Slider can be created by right clicking on the Hierarchy Window, choose UI and choose Slider. When a Slider object is created, it already has 3 children, the Background, Fill Area, and Handle Slide Area. As their name suggested, we can modify the background colour and the fill area colour, so it is more visible to the player. For this project, I will replace the handle slide with an animated attacker of type Fox so the timer has a more interesting visual to it.

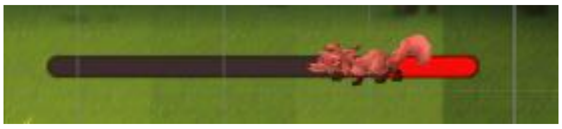
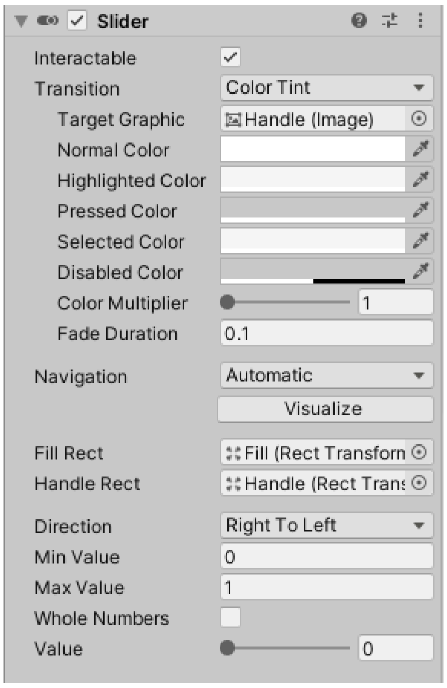


Figure 41. Level Timer

Figure 41 is how the timer would look like when the configurations is correct. To animate the handle, we need to add an Animator component to its parent component, the Slider. Then add a sprite to represent the fox attacker in the Image component in the Handle child object of the Handle Slide Area object. Once we have the Animator component added to the Slider, we can animate the Fox Sprite like we did with the other Attackers and Defenders.


Slider

Interactable	✓
Transition	Color Tint
Target Graphic	Handle (Image)
Normal Color	
Highlighted Color	
Pressed Color	
Selected Color	
Disabled Color	
Color Multiplier	1
Fade Duration	0.1
Navigation	Automatic
	Visualize
Fill Rect	Fill (Rect Transforn
Handle Rect	Handle (Rect Trans
Direction	Right To Left
Min Value	0
Max Value	1
Whole Numbers	
Value	0

Figure 42. Slider Component

Each Slider object has its Slider component. There are many options and values for the developer to modify the behaviour of the slider as seen from figure 42. For this game, we only need to focus on the Direction, Min Value, Max Value, and the Value fields. The direction is how the handle moves as the value goes from min value to max value. Since our game the Attackers is moving from right to left, the direction should also be right to left so it will not confuse the player visually.

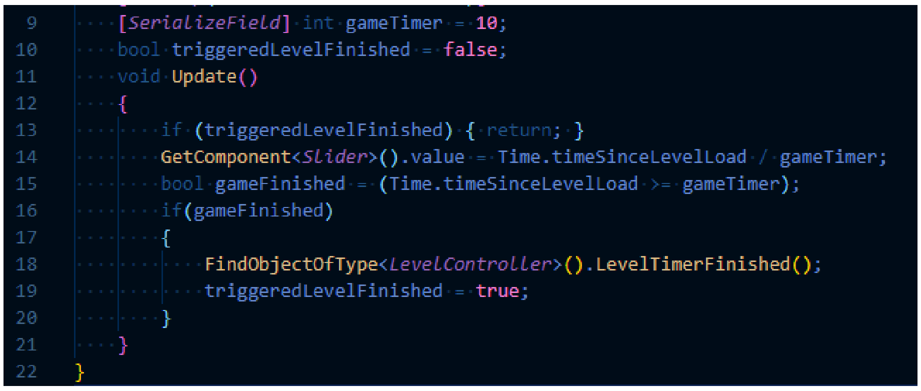
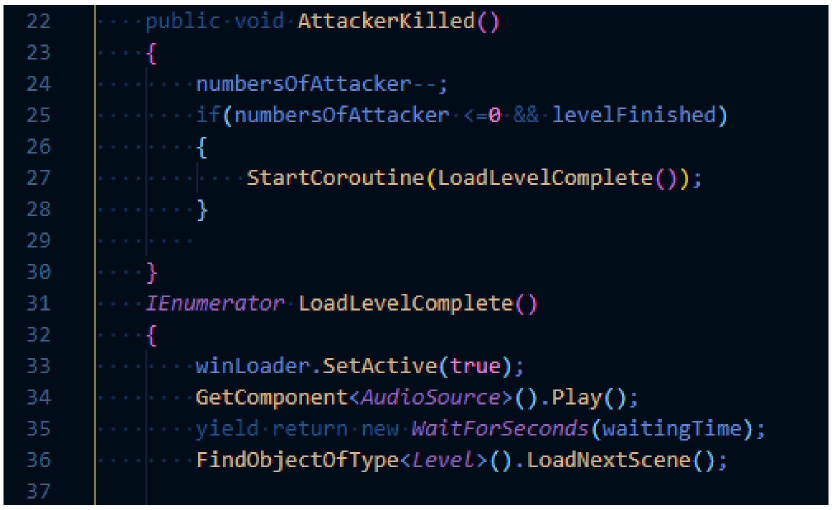

```
9      [SerializeField] int gameTimer = 10;
10     bool triggeredLevelFinished = false;
11     void Update()
12     {
13         if (triggeredLevelFinished) { return; }
14         GetComponent<Slider>().value = Time.timeSinceLevelLoad / gameTimer;
15         bool gameFinished = (Time.timeSinceLevelLoad >= gameTimer);
16         if(gameFinished)
17         {
18             FindObjectOfType<LevelController>().LevelTimerFinished();
19             triggeredLevelFinished = true;
20         }
21     }
22 }
```

Figure 43. Level Timer Script

Finally, a LevelTimer.cs script is required for the Slider Object. The gameTimer variable is how long that game level last in seconds. The triggeredLevelFinished variable is a trigger to stop the code from line 14 to 20 when the timer is up. If the level timer is still running, line 14 is responsible for the value changes of the Slider mentioned above. We can get this value to range from 0 to 1 by dividing the time that has passed with the gameTimer variable. The timeSinceLevelLoad variable is built-in from Unity so we can easily get the value of time that has passed in seconds. Once the play time is equal to or higher than the gameTimer value, the function LevelTimerFinished is loaded from the level controller script and stop the spawner from creating new attackers.

4.7.2 Level Controller

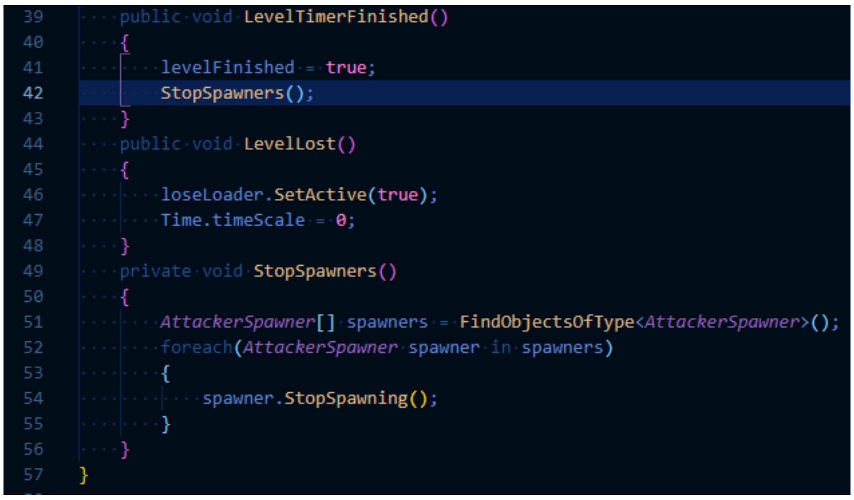
The final object of this project, the Level Controller, determines when the level ends. We also need a LevelController.cs script for this object. Firstly, the script has an AttackerSpawned and AttackerKilled function. These functions are called in the Attacker script. We have a numbersOfAttacker variable in the LevelController.cs script to decide if there is any attacker left on the screen. The AttackerSpawned is called before the Start function in the Attacker script. It is called in the Awake function, and increment the numbersOfAttacker variable by 1 every time it is called. The AttackerKilled is the exact opposite, it is called in the OnDestroy function of the Attacker.cs script. Once it is called, the numbersOfAttacker is decreased by 1 and check if this variable is currently 0.



```
22      public void AttackerKilled()
23      {
24          numbersOfAttacker--;
25          if(numbersOfAttacker <=0 && levelFinished)
26          {
27              StartCoroutine(LoadLevelComplete());
28          }
29          
30      }
31      IEnumerator LoadLevelComplete()
32      {
33          winLoader.SetActive(true);
34          GetComponent<AudioSource>().Play();
35          yield return new WaitForSeconds(waitingTime);
36          FindObjectOfType<Level>().LoadNextScene();
37
```

Figure 44. Level Controller Script

As seen from figure 44 above, on line 25, the function checks if the number of attackers is now 0 and if the level has finished. If these requirements are met, it will start a coroutine to load the winning screen. Coroutine is not compulsory here; however, it is used here because the desired outcome is for the function to wait for a period of time before going to the next level. The LoadNextScene on line 36 will load the game to the next level.



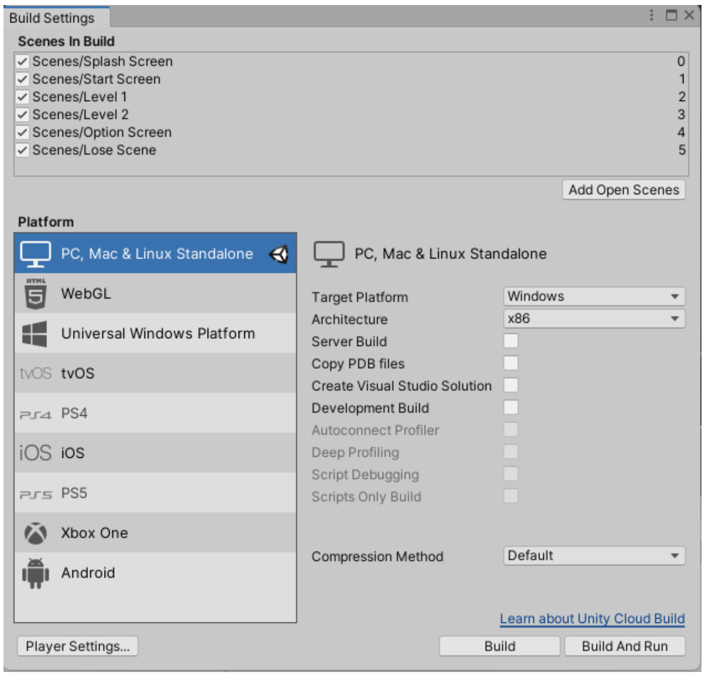
```
39        public void LevelTimerFinished()
40        {
41            levelFinished = true;
42            StopSpawners();
43        }
44        public void LevelLost()
45        {
46            loseLoader.SetActive(true);
47            Time.timeScale = 0;
48        }
49        private void StopSpawners()
50        {
51            AttackerSpawner[] spawners = FindObjectsOfType<AttackerSpawner>();
52            foreach(AttackerSpawner spawner in spawners)
53            {
54                spawner.StopSpawning();
55            }
56        }
57    }
```

Figure 45. LevelController Script 2

Refer to the previous section, figure 43 displayed the call on LevelTimerFinished on line 18. When this function is called, the levelFinished value is set to true to meet the condition on line 25 in figure 44. Additionally, we want to stop all the attacker spawners from creating anymore instances. We already have the StopSpawning function in the Attacker.cs script, now we just need to call it from the level controller. Since we have multiplie spawners, a loop is required. The StopSpawners function from line 49 to 56 in figure 45 loops through all the spawners available in the game and stop them. Finally, the LevelLost function sets the time scale back to 0 so when the player wants to replay the level, the time-SinceLevelLoad variable seen in figure 43 works correctly. The loseLoader on line 46 in figure 45 and the winLoader on line 33 in figure 44 are optional. These are for a better winning or losing experience. They are created as new Canvas that are on top of the main Canvas. These canvases are disabled by default, they are set back to active state when the functions are called.

4.8 Build The Game

The final section of this thesis will be on how to build the game once it is completed. When the game is completed with all its core features and mechanics, the developer can build a standalone version of the game for further testing, and bug report purposes. Building a game in Unity is simple, choose the Build Settings from the File tab to open the menu for modifications of the build.


Build Settings

Scenes In Build

✓ Scenes/Splash Screen	0
✓ Scenes/Start Screen	1
✓ Scenes/Level 1	2
✓ Scenes/Level 2	3
✓ Scenes/Option Screen	4
✓ Scenes/Lose Scene	5

Platform: PC, Mac & Linux Standalone

Target Platform	Windows
Architecture	x86
Server Build	
Copy PDB files	
Create Visual Studio Solution	
Development Build	
Autoconnect Profiler	
Deep Profiling	
Script Debugging	
Scripts Only Build	
Compression Method	Default

Figure 46. Build Settings Window

As shown in figure 46 above, the build setting window contains all the scenes that are in the project. These scenes can be a start screen, levels, menu screen, and more. We can select which scene to include in the standalone build. Unity offers builds for many platforms such as PC, MacOS, iOS, WebGL or Android. The player settings option from the left bottom corner allows us to change the name and version of the project. After all scenes and the desired platform is selected, click on “build and run”. After the game is built it will run automatically. We now have all the files needed to play the game, sharing these files to others will act as a copy of the game. Other software can be used to convert this into a standalone installation of the game and share it with other people. If the game is built in WebGL platform, it can be uploaded to websites for others to play.

5 CONCLUSION

Through this thesis, how to create simple core features of a 2D tower defend game is demonstrated. Some parts such as animations and sound designs are skipped to prevent confusion on the topic. As seen from the various sections of the thesis, Unity engine assist developers from C# built-in functions to its development interface. With the help of Unity, 2D game development has become significantly easier, developers no longer need to create many functions from scratch, and a simple understanding of C# is more than enough to make simple project like this. More complex problems will be introduced in larger project; however, the fundamentals of game development are learnt quickly through projects like this one. Working on the game and this thesis, I have gained a deeper understanding of both programming and game development. The project in this thesis is created for learning purposes, it can be further developed in the future.

Game development is a fun and interesting way to learn programming and logical thinking. Not only that it is a way to learn, but it is also a medium to express the developers' imagination. There are many game engines current on the market, choosing which engine to start with is completely down to personal reference. Unity has been gaining more popularity over the recent years as it is constantly updating its engine and provides a simpler interface for beginners. More features are added in every new version.

To summarize, this thesis has presented a way to approach game development through the process of creating a game. This thesis can act as a guideline to development but not as step-by-step tutorial.

6 REFERENCE

Adam, S. 2021. What is Unity? Everything you need to know. Available at: https://www.androidauthority.com/what-is-unity-1131558/. Accessed 1 November 2021.

Basu, S. 2017. 2D PLATFORM GAME. Available at: https://www.theseus.fi/bitstream/handle/10024/139982/Basu_Sourav.pdf?sequence=1&isAllowed=y. Accessed 1 November 2021.

Brackeys, 2018. *Basic Principles of Game Design.* Available at: https://www.youtube.com/watch?v=G8AT01tuyrk&t=203s. Accessed 2 November 2021

Burgun, K., 2012. *Game Design Theory A New Philosophy for Understanding Games.* 1st ed. Massachusetts: A K Peters/CRC Press.

Corazza, S., 2013. *History of the Unity Engine [Freerunner 3D Animation Project]*. Available at: https://seraphinacorazza.wordpress.com/2013/02/14/history-of-the-unity-engine-freerunner-3d-animation-project/. Accessed 23 April 2021.

Downie, C., 2016. *Evolution of our products and pricing.* Available at: https://blog.unity.com/community/evolution-of-our-products-and-pricing. Accessed 23 April 2021.

TechTerms, 2012. *Sprite.* Available at: https://techterms.com/definition/sprite. Accessed 1 December 2021.

UnityTechnologies, a., 2021. *Learning the Interface.* Available at: https://docs.unity3d.com/2021.2/Documentation/Manual/LearningtheInterface.html. Accessed 1 November 2021.

UnityTechnologies, b., 2021. *Project view.* Available at: https://docs.unity3d.com/2021.2/Documentation/Manual/ProjectView.html. Accessed 1 November 2021.

UnityTechnologies, c., 2021. *Scene View.* Available at: https://docs.unity3d.com/2021.2/Documentation/Manual/UsingTheSceneView.html. Accessed 1 November 2021.

UnityTechnologies, d., 2021. *Game View.* Available at: https://docs.unity3d.com/2021.2/Documentation/Manual/GameView.html. Accessed 1 November 2021.

UnityTechnologies, e., 2021. *Hierarchy.* Available at: https://docs.unity3d.com/2021.2/Documentation/Manual/Hierarchy.html. Accessed 1 November 2021.

UnityTechnologies, f., 2021. *Inspector Window*. Available at: https://docs.unity3d.com/2021.2/Documentation/Manual/UsingTheInspector.html. Accessed 1 November 2021.

UnityTechnologies, g., 2021. *Toolbar*. Available at: https://docs.unity3d.com/2021.2/Documentation/Manual/Toolbar.html. Accessed 1 November 2021.

UnityTechnologies, h., 2021. *The Status Bar*. Available at: https://docs.unity3d.com/2021.2/Documentation/Manual/StatusBar.html. Accessed 1 November 2021.

UnityTechnologies, i., 2021. *Prefabs.* Available at: https://docs.unity3d.com/2021.2/Documentation/Manual/Prefabs.html. Accessed 1 December 2021.

Game Scripts

Attacker.cs

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

public class Attacker : MonoBehaviour
{
    [Range(0F, 5F)]
    float currentSpd;
    GameObject currentTarget;
    private void Awake()
    {
        FindObjectOfType<LevelController>().AttackerSpawned();
    }
    private void OnDestroy()
    {
        LevelController level = FindObjectOfType<LevelController>();
        if(level != null)
        {
            level.AttackerKilled();
        }
    }
    void Update()
    {
        transform.Translate(Vector2.left * currentSpd * Time.deltaTime);
        UpdateAnimator();
    }

    private void UpdateAnimator()
    {
       if(!currentTarget)
        {
            GetComponent<Animator>().SetBool("isAttacking", false);
        }
    }

    public void SetMovementSpd(float speed)
    {
        currentSpd = speed;
    }

    public void Attack(GameObject defender)
    {
        GetComponent<Animator>().SetBool("isAttacking", true);
        currentTarget = defender;
    }

    public void StrikeCurrentTarget(float damage)
    {
        if (!currentTarget) { return; }
        Health health = currentTarget.GetComponent<Health>();
        if(health)
        {
            health.DealDamage(damage);
        }
    }
}
```

APPENDIX 1/3

AttackerSpawner.cs

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

public class AttackerSpawner : MonoBehaviour
{
    [SerializeField] float minDelay = 1F;
    [SerializeField] float maxDelay = 5F;
    [SerializeField] Attacker[] attackerprefabs;
    bool spawn = true;
    int randomAttacker;
    // Start is called before the first frame update
    IEnumerator Start()
    {
        while(spawn)
        {
            yield return new WaitForSeconds(Random.Range(minDelay, maxDelay));
            SpawnAttacker();
        }
    }
    public void StopSpawning()
    {
        spawn = false;
    }

    private void SpawnAttacker()
    {
        randomAttacker = Random.Range(0, attackerprefabs.Length);
        Attacker newAttacker = Instantiate(attackerprefabs[randomAttacker], trans-
form.position, transform.rotation) as Attacker;
        newAttacker.transform.parent = transform;
    }

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

    }
}
```

Defender.cs

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

public class Defender : MonoBehaviour
{
    [SerializeField] int starCost = 100;

    public void GetStar(int amount)
    {
        FindObjectOfType<DisplayStar>().AddStar(amount);
    }
    public int GetStarCost()
    {
        return starCost;
    }
}
```

APPENDIX 1/5

DefenderSpawner.cs

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

public class DefenderSpawner : MonoBehaviour
{
    Defender defender;
    GameObject defenderParent;
    const string DEFENDER_PARENT_NAME = "Defender";
    private void Start()
    {
        CreateDefenderParent();
    }

    private void CreateDefenderParent()
    {
        defenderParent = GameObject.Find(DEFENDER_PARENT_NAME);
        if(!defenderParent)
        {
            defenderParent = new GameObject(DEFENDER_PARENT_NAME);
        }
    }

    public void GetDefender(Defender select)
    {
        defender = select;
    }

    private void OnMouseDown()
    {
        SpawnAttempt(MousePos());
    }
    private void SpawnAttempt(Vector2 gridPos)
    {
        var starDisplay = FindObjectOfType<DisplayStar>();
        int defenderCost = defender.GetStarCost();
        if(starDisplay.IsEnough(defenderCost))
        {
            SpawnDefender(gridPos);
            starDisplay.SpendStar(defenderCost);
        }
    }

    private Vector2 MousePos()
    {
```

APPENDIX 1/6

```
        Vector2 clickPos = new Vector2(Input.mousePosition.x, Input.mousePosition.y);
        Vector2 worldPos = Camera.main.ScreenToWorldPoint(clickPos);
        Vector2 gridPos= Grid(worldPos);
        return gridPos;
    }
    private Vector2 Grid(Vector2 worldPos)
    {
        float newX = Mathf.RoundToInt(worldPos.x);
        float newY = Mathf.RoundToInt(worldPos.y);
        return new Vector2(newX, newY);
    }
    private void SpawnDefender(Vector2 worldPos)
    {
        Defender newDefender = Instantiate(defender, worldPos, Quaternion.identity)
as Defender;
        newDefender.transform.parent = defenderParent.transform;
    }

}
```

DisplayStar.cs

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

public class DisplayStar : MonoBehaviour
{
    [SerializeField] int stars = 100;
    Text starText;
    void Start()
    {
        starText = GetComponent<Text>();
        UpdateDisplay();
    }
    public bool IsEnough(int amount)
    {
        return stars >= amount;
    }
    private void UpdateDisplay()
    {
        starText.text = stars.ToString();
    }
    public void AddStar(int amount)
    {
        stars += amount;
        UpdateDisplay();
    }
    public void SpendStar(int amount)
    {
        if (stars >= amount)
        {
            stars -= amount;
            UpdateDisplay();
        }
    }
}
```

APPENDIX 1/8

Fox.cs

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

public class Fox : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        GameObject otherObject = collision.gameObject;
        if(otherObject.GetComponent<GraveStone>())
        {
            GetComponent<Animator>().SetTrigger("isTrigger");
        }
        else
        if (otherObject.GetComponent<Defender>())
        {
            GetComponent<Attacker>().Attack(otherObject);
        }
    }
}
```

Health.cs

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

public class Health : MonoBehaviour
{
    [SerializeField] float health = 40F;
    [SerializeField] GameObject deathVFX;
    public void DealDamage(float damage)
    {
        health -= damage;
        if(health <=0)
        {
            Die();
        }
    }
    private void Die()
    {
        Destroy(gameObject);
        GameObject deathVFXobject = Instantiate(deathVFX, transform.position, trans-
form.rotation);
        if (!deathVFX) { return; }
        Destroy(deathVFXobject, 1F);
    }
}
```

Level.cs

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

public class Level : MonoBehaviour
{
    [SerializeField] int LoadingTime = 3;
    int currentSceneIndex;
    // Start is called before the first frame update
    void Start()
    {
        currentSceneIndex = SceneManager.GetActiveScene().buildIndex;
        if (currentSceneIndex == 0)
        {
        StartCoroutine(LoadMainScreen());
        }
    }
    IEnumerator LoadMainScreen()
    {
        yield return new WaitForSeconds(LoadingTime);
        LoadNextScene();
    }
    public void LoadNextScene()
    {
        SceneManager.LoadScene(currentSceneIndex+1);
    }
    public void LoadLoseScreen()
    {
        SceneManager.LoadScene("Lose Scene");
    }
    public void LoadMainMenu()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene("Start Screen");
    }
    public void RestartScene()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(currentSceneIndex);
    }
    public void QuitGame()
    {
        Application.Quit();
    }
    public void LoadOption()
    {
        SceneManager.LoadScene("Option Screen");
    }

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

    }
}
```

LevelController.cs

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

public class LevelController : MonoBehaviour
{
    [SerializeField] int waitingTime = 2;
    [SerializeField] GameObject winLoader;
    [SerializeField] GameObject loseLoader;
    [SerializeField] int numbersOfAttacker = 0;
    [SerializeField] bool levelFinished = false;

    private void Start()
    {
        winLoader.SetActive(false);
        loseLoader.SetActive(false);
    }
    public void AttackerSpawned()
    {
        numbersOfAttacker++;
    }
    public void AttackerKilled()
    {
        numbersOfAttacker--;
        if(numbersOfAttacker <=0 && levelFinished)
        {
            StartCoroutine(LoadLevelComplete());
        }

    }
    IEnumerator LoadLevelComplete()
    {
        winLoader.SetActive(true);
        GetComponent<AudioSource>().Play();
        yield return new WaitForSeconds(waitingTime);
        FindObjectOfType<Level>().LoadNextScene();

    }
    public void LevelTimerFinished()
    {
        levelFinished = true;
        StopSpawners();
    }
    public void LevelLost()
    {
        loseLoader.SetActive(true);
```

APPENDIX 1/13

```
        Time.timeScale = 0;
    }
    private void StopSpawners()
    {
        AttackerSpawner[] spawners = FindObjectsOfType<AttackerSpawner>();
        foreach(AttackerSpawner spawner in spawners)
        {
            spawner.StopSpawning();
        }
    }
}
```

LevelTimer.cs

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

public class LevelTimer : MonoBehaviour
{
    [Tooltip("Game timer in seconds")]
    [SerializeField] int gameTimer = 10;
    bool triggeredLevelFinished = false;
    void Update()
    {
        if (triggeredLevelFinished) { return; }
        GetComponent<Slider>().value = Time.timeSinceLevelLoad / gameTimer;
        bool gameFinished = (Time.timeSinceLevelLoad >= gameTimer);
        if(gameFinished)
        {
            FindObjectOfType<LevelController>().LevelTimerFinished();
            triggeredLevelFinished = true;
        }
    }
}
```

LiveDisplay.cs

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

public class LivesDisplay : MonoBehaviour
{
    [SerializeField] float baseLives = 3;
    float lives;
    int damage = 1;
    Text lifeText;
    void Start()
    {
        lives = baseLives - PlayerPrefsController.GetDifficulty();
        lifeText = GetComponent<Text>();
        UpdateDisplay();
    }

    public void UpdateDisplay()
    {
        lifeText.text = lives.ToString();
    }
    public void TakeLife()
    {
        lives -= damage;
        UpdateDisplay();
        if(lives<=0)
        {
            FindObjectOfType<LevelController>().LevelLost();
        }
    }
}
```

APPENDIX 1/16

Lizard.cs

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

public class Lizard : MonoBehaviour
{


    private void OnTriggerEnter2D(Collider2D collision)
    {
        GameObject otherObject = collision.gameObject;
        if (otherObject.GetComponent<Defender>())
        {
            GetComponent<Attacker>().Attack(otherObject);
        }
    }


}
```

LoseCollider.cs

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

public class LoseCollider : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D otherCollider)
    {
        FindObjectOfType<LivesDisplay>().TakeLife();
        Destroy(otherCollider.gameObject);
    }
}
```

APPENDIX 1/18

MenuButtons.cs

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

public class MenuButtons : MonoBehaviour
{
    [SerializeField] Defender defender;

    private void Start()
    {
        LabelButtonWithCost();
    }

    private void LabelButtonWithCost()
    {
        Text costText = GetComponentInChildren<Text>();
        if(!costText)
        {
            Debug.LogError(name + " has no cost text, add some!");
        }
        else
        {
            costText.text = defender.GetStarCost().ToString();
        }
    }

    private void OnMouseDown()
    {
        var buttons = FindObjectsOfType<MenuButtons>();
        foreach (MenuButtons button in buttons)
        {
            button.GetComponent<SpriteRenderer>().color = new Color32(80, 80, 80,
255);
        }
        GetComponent<SpriteRenderer>().color = Color.white;
        FindObjectOfType<DefenderSpawner>().GetDefender(defender);
    }
}
```

APPENDIX 1/19

Projectile.cs

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

public class Projectile : MonoBehaviour
{
    [SerializeField] float projectileSpd = 5F;
    [SerializeField] float rotationSpd =500F;
    [SerializeField] float damage = 20F;

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

    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate( Vector2.right * projectileSpd * Time.del-
taTime,Space.World);
        transform.RotateAround(transform.localPosition,Vector3.forward,-rota-
tionSpd*Time.deltaTime);
    }
    private void OnTriggerEnter2D(Collider2D collision)
    {
        var health = collision.GetComponent<Health>();
        var attacker = collision.GetComponent<Attacker>();
        if (attacker && health)
        {
            health.DealDamage(damage);
            Destroy(gameObject);
        }
    }
}
```

APPENDIX 1/20

Shooter.cs

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

public class Shooter : MonoBehaviour
{
    [SerializeField] GameObject projectile,gun;
    AttackerSpawner myLaneSpawner;
    Animator animator;
    GameObject projectileParent;
    const string PROJECTILE_PARENT_NAME = "Projectile";
    private void Start()
    {
        SetLaneSpawner();
        animator = GetComponent<Animator>();
        projectileParent = GameObject.Find(PROJECTILE_PARENT_NAME);
        if(!projectileParent)
        {
            projectileParent = new GameObject(PROJECTILE_PARENT_NAME);
        }
    }
    private void Update()
    {
        if(isAttackerInLane())
        {
            animator.SetBool("isAttacking", true);
        }
        else
        {
            animator.SetBool("isAttacking", false);
        }
    }
    private void SetLaneSpawner()
    {
        AttackerSpawner[] spawners = FindObjectsOfType<AttackerSpawner>();
        foreach(AttackerSpawner spawner in spawners)
        {
            bool isCloseEnough = (Mathf.Abs(spawner.transform.position.y - trans-
form.position.y) <= Mathf.Epsilon);
            if(isCloseEnough)
            {
                myLaneSpawner = spawner;
            }
        }

    }
    private bool isAttackerInLane()
    {
        if (myLaneSpawner.transform.childCount <= 0)
        {
            return false;
        }

        else return true;
    }
    public void Fire()
    {
        GameObject newProjectile = Instantiate(projectile, gun.transform.position, 
Quaternion.identity) as GameObject;
        newProjectile.transform.parent = projectileParent.transform;
    }
}
```