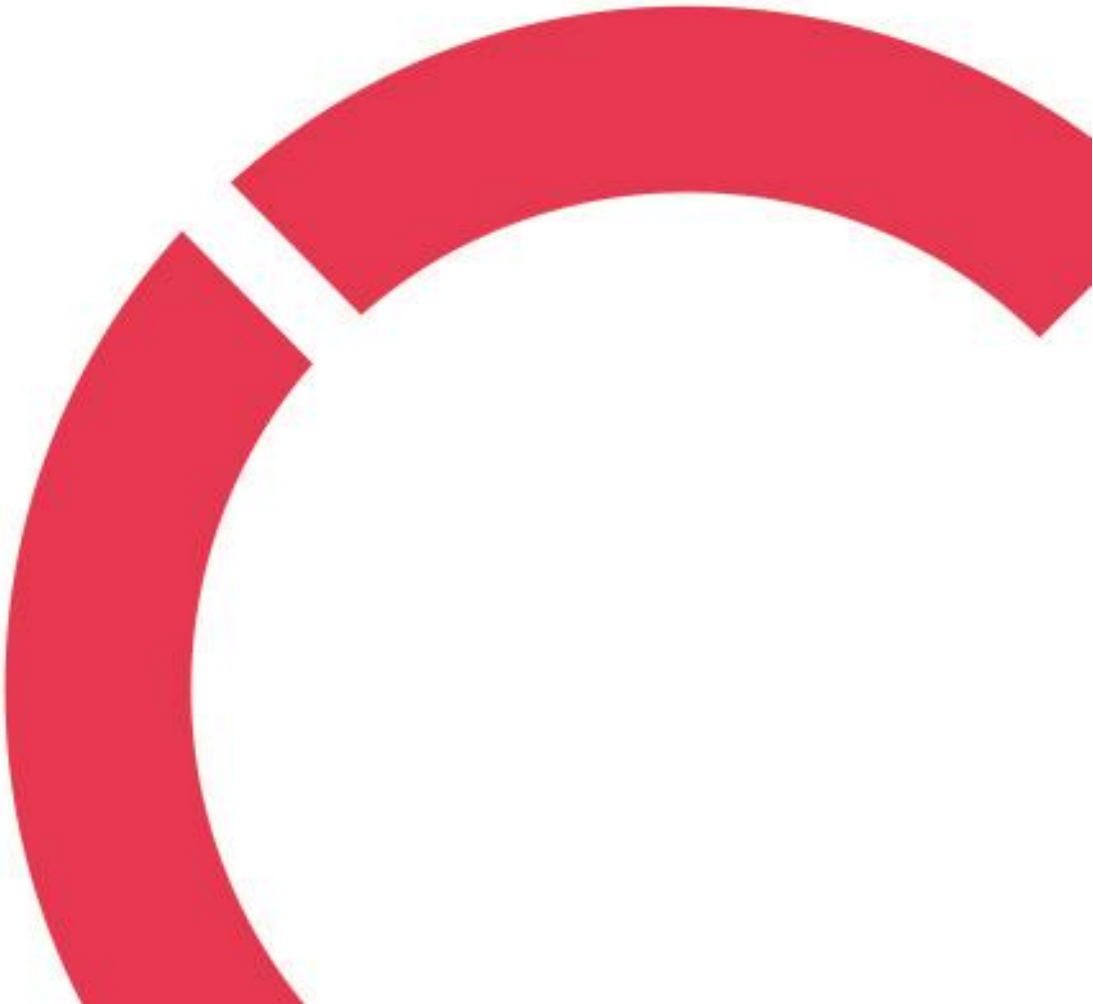
Nguyen Tuan

DEVELOPING A 3D ZOMBIE SURVIVAL RUN GAME WITH UNITY.

Thesis
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Information Technology
June 2023



ABSTRACT

Centria University of Applied Sciences	**Date** June 2023	**Author** Nguyen Tuan
Degree programme Information Technology		
Name of thesis DEVELOPING A 3D ZOMBIE SURVIVAL GAME WITH UNITY		
Centria supervisor Jari Isohanni		**Pages** 50 + 10

The aim of this project was to learn how to create a 3D game with Unity, see the workflow, and learn the different steps and mechanics of the Unity environment.

The goal was to create a zombie survival game, in which the player would have to try to kill and survive among a lot of zombies and a final boss in the final level within a specified time. The player must find and pick up some beneficial equipment such as ammunition, light, and weapon to survive through the game. To win the game, the player must pass all the levels of the game and kill the final boss. In case the player died at any level, the player must start the game from the beginning.

The game can be played on IOS, and Windows platforms. This is the first step in making a Unity game and some useful experience for the future of the game industry.

Key words
Game, Unity, zombie, 3D

ABSTRACT

CONTENTS

FIGURES

CODE

1 INTRODUCTION

The video game industry is growing fast nowadays. It meets everyone's needs to relax and be entertained after a working day, a school day for those who have a passion for games. A video game can be played everywhere, every time on an electronic device such as a mobile phone, tablet, computer, and gaming console. The millions of online players who can connect with each other has become more than just a childhood hobby, it is for people of all ages.

The game industry has risen strongly in many years, it became a business worth billions of dollars. The revenue of the worldwide PC gaming market was calculated at approximately 37 billion US dollars and the mobile gaming market has an estimated income of around 77 billion US dollars. (Clement, 2022.) Especially in Finland, the game industry has developed in recent years after two successful companies which are Rovio Entertainment (the father of Angry Birds) and Supercell (the creator of Hay Day and Clash of Clans), both based in Finland. (Helsinki Times, 2022) This is one of the reasons that the thesis topic was chosen , besides, it is because of the author's interest in the field of game development.

The thesis focuses on the development of a 3D game utilizing the Unity game engine. It outlines the techniques and principles involved in creating a 3D game using game development tools. Although it may present challenges, developing a 3D game for the thesis can be a fulfilling experience, offering opportunities for skill, creativity, and knowledge expansion in the field of game development. The game created for this thesis is intended for mature audiences aged 17 and above due to its violent and graphic content.

The zombie games can highlight everyone's real-world issues, for example, the danger of the pandemic, societal collapse, or the consequences of human greed and exploitation. They are also popular due to the excitement of fighting zombies and surviving in a post-apocalyptic world. The massive fan base of zombie games makes them highly profitable through both sales and in-game purchases.

The main goal of this game is to try to survive among a lot of zombies, and the player must find the tools needed for the game such as weapons, flashlights, and ammo. The importance is trying to kill the final boss in the final level to win the game. The game will be over when the player dies because of receiving a lot of damage from monsters.

Ultimately, the main purpose of this thesis is to analyze the development process of a 3D Zombie survival game in detail and to show the challenges encountered as well as the way to overcome them during the development. The thesis also tries to contribute knowledge on game development and to serve as a valuable resource for game researchers, and enthusiasts, who have interested in the potential of video games for education, social impact, and entertainment.

2 THEORETICAL PARADIGMS

Unity is software that game developers utilize to create games compatible with various platforms that accommodate both 2D and 3D game engines. The engine is powerful and easy to use, it provides developers with various tools including physics, collision detection, and 3D rendering. The “Asset Store” is an essential component of Unity that allows developers to upload their creations and share them with other members of the development community. By using the Asset Store, programmers can focus on the game’s design, coding, or creating enjoyable experiences without having to start from scratch. (Sinicki 2021.)

2.1 History of Unity engine

Unity Technologies developed Unity, a game engine and IDE that works across various platforms. It is utilized by over one million developers who use it to create games for desktop, mobile devices, consoles, and web plugins. Unity was first developed in C/C++ and can support code written in C# or JavaScript. Initially, the purpose of its creation was to develop games for the OS X operating system. However, with the passage of time, it has expanded to become a game engine that supports multiple platforms. (Corazza. S, 2013) Unity Technologies was established in Copenhagen, Denmark in 2004 by David Helgason (CEO), Nicholas Francis (CCO), and Joachim Ante (CTO) after their first game, GooBall, failed to succeed. The individuals acknowledged the importance of creating and improving engines and tools and decided to develop an engine that was accessible and cost-effective for everyone. No details were left out in their efforts. Funding for Unity Technologies has been provided by Sequoia Capital, WestSummit Capital, and iGlobe Partners. (Corazza 2013.)

At the Apple Worldwide Developers Conference in 2005, Unity was introduced for the first time to the world. Initially, it was designed to operate and construct projects solely on Mac platform computers, and it received enough acceptance to continue its engine and tools development. Unity 3 was launched in September 2010 with a focus on incorporating more of the tools that are usually available to high-end studios. As a result, the company was able to attract larger developers while also offering a game engine in a single, affordable package to independent and smaller teams. The next version of Unity, Unity 4.0, was released at the end of 2012 and includes Mecanim animation and DirectX11 support as new features. (Corazza 2013.) The upcoming versions of Unity which are Unity 5 and Unity 2017.1,

were published in 2015 and 2017. Unity aims to introduce new features and improvements. Especially, Unity 2017.1 had improved overall performance and released new tools for game developers such as Cinemachine, Timeline, and Unity Collaborate (Unity 2015; Unity 2017). The latest version of Unity, Unity 2022.2.13, was released in March 2023 and focused on improving the performance, fixing, and changing to give developers the best experiences. (Unity 2023)

2.2 Unity's feature

Unity became one of the most popular gaming engines in recent years. The engine has numerous features and a visually impressive interface for game creation and development. The Unity editor is a crucial aspect of the engine, providing a smoother and more productive coding experience. Additionally, this interface enables developers to easily visualize the results of their coding efforts. (French 2022.)

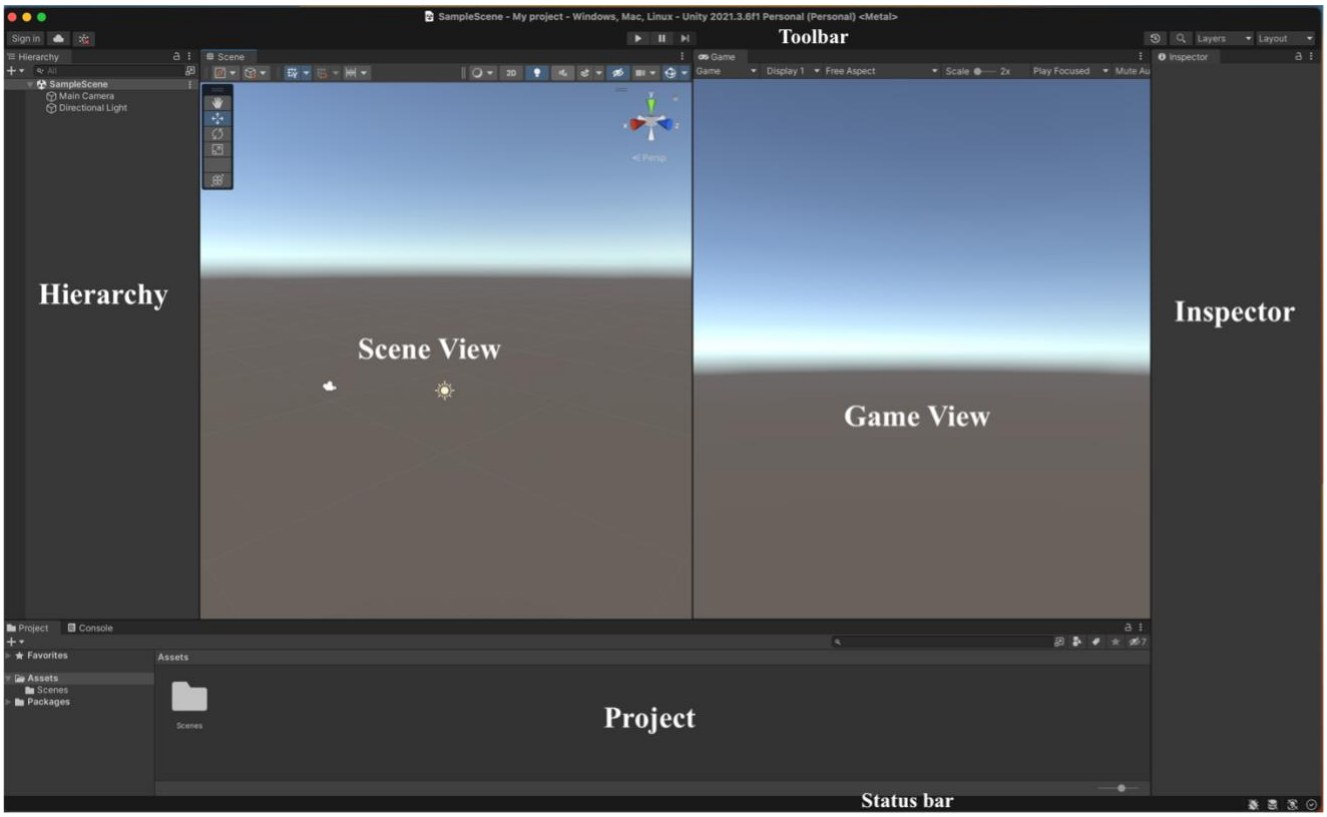


Figure 1. Unity's user interface

The group of buttons located at the top of the Unity editor is referred to as the Toolbar. This group of buttons grants quick access to various tools and features, such as move, rotate, scale, rect transform,

and transform. Positioned in the center of the Toolbar are three buttons, namely Play, Pause, and Step buttons, which aid in navigation with the Game View. The toolbar enables developers to view, search, undo, or redo multiple actions performed in the Unity editor. Additionally, there are drop-down buttons at the end of the Toolbar that allow modifications of Layers and Layout. (Unity Documentation 2023.)

Developers can access and modify all game objects through the Hierarchy window in the Unity editor. This window is crucial as it offers a hierarchical view that enables users to organize and manage game objects within a scene. Additionally, it exhibits a tree-like structure that exemplifies a parent-child relationship. It implies that a game object can contain other objects and inherit the attributes of its parent game object, such as rotations, scales, and transforms. This feature helps developers save time when modifying multiple similar game objects. The Hierarchy window also provides various tools for creating, duplicating, and deleting game objects, as well as adjusting their position, rotation, and scale. As an essential, powerful, and versatile interface, the Hierarchy window enables developers to control game objects in a scene effectively and efficiently. (Unity Documentation 2023.)

The Game View is a crucial resource that offers game developers a real-time preview of their game from the perspective of the camera when they hit the Play button. While the developer tests the prototype, values can be adjusted which can be seen in real-time. Any changes made during Play mode are temporary and will be undone when the mode is exited. This feature provides developers with an opportunity to test and operate their game as a player in a live preview environment. Developers are able to use specific tools such as camera position, zoom level, and aspect ratio, to help their games use on different platforms and devices, ensuring optimal performance. Additionally, it shows debugging and performance information such as FPS, memory usage, and more, allowing users to identify and resolve any errors or glitches in their game (Unity Documentation 2023.).

Upon opening the Unity editor, developers are greeted with the Scene View window as their initial interface. This window displays the game world in a 3D view and includes the light, camera, and skyline elements. Developers are able to construct and develop their game in this window with the ability to make real-time edits to the objects during the game. The Scene View also allows for object manipulation, such as movement, rotation, and scaling, as well as adjustments to properties and

settings. With this window, developers have the ability to simulate the camera and light and place objects in desired positions. (Unity Documentation 2023.2)

To access and modify information about various aspects of an object such as GameObject, Unity component, Asset, Materials, developers use the Inspector window. This window displays all the components of an object, including script, physics, colliders, and sound, and allows developers to add new components as needed. The Inspector window provides an easy-to-use interface for creating and editing the game world and is an essential tool for working with Unity. Becoming familiar with this tool in Scene View is an important initial step in using Unity. (Unity Documentation 2023.)

When working with Unity, it is a straightforward process to bring in different types of files and resources, such as models, audio, scripts, and more. The Project window is the central hub where all of these assets are organized and displayed, making it an essential part of the software. Any of the assets found in this window can be easily dragged and dropped directly into the Scene View. Any changes made to these assets can be immediately seen in the game, and they can be modified at any time. Additionally, the Project window offers several built-in tools to help users search for and sort through assets, making it easier to work with specific files. Furthermore, this window provides options for previewing and checking assets. (Unity Documentation 2023.)

The Unity Editor has a fixed Status Bar located at its bottom, just like the Toolbar, which cannot be moved or changed. The Status Bar is an essential window for developers as it displays information related to Unity processes, memory usage, tools, and settings. It also shows the latest message from the Console window, which can be accessed by clicking on it. Additionally, the Status Bar allows adjusting the Editor's behavior through various controls and settings such as customizing the Console window's size, Editor window's layout, and color scheme. This bar is a crucial tool for managing project status and providing real-time feedback to developers. (Unity Documentation 2023.)

When a game lacks scary sounds, audio, or music in the background, it is not complete. With audio in Unity, developer can choose various options such as importing them, play sounds in 3D space, or applying effects like echo and filtering by adding Audio Filters to objects; moreover, the audio can be recorded on the machine from any microphone. The microphone’s API has missions to find available

microphones, query their capabilities, and start and stop the record. Another option of Unity is to access the microphones from a script and make audio clips by recording directly. Unity allows user to import audio files in AIFF, WAV, MP3, and OGG formats by dragging them to the Project window and importing them to create an Audio Clip that user can drag to an Audio Source or use from a script. For music, Unity accepts to import tracker modules from .xm, .mod, .it, and .s3m. (Unity Documentation 2023.)

3 3D GAME DEVELOPMENT

Before beginning any game or software development, it is crucial to plan for the whole project. Proper game design, engaging gameplay, quality game art, and sound are essential for success in game development. Clear objectives and schedules facilities the development process and allow for necessary modifications during game development. A game engine is a necessary component when combined with effective planning to create a captivating game.

3.1 Gameplay overview

The game is a first-person zombie run game where the player must survive through a post-apocalyptic which filled with hordes of zombies. The game's theme is set in a dark world environment where the player can discover different areas of the forest, moreover, they must scavenge supplied tools such as ammo, weapons, light, and battery. This game is fast-paced, intense, moody, and scary when the player is constantly on the move to avoid zombies or find the weapons to kill the zombies. During the game, the player will go through different levels, and they also encounter different types of zombies with their own abilities and weaknesses.

In the beginning, the game features only night mode and one-player mode. After finishing the first version of this game, the game will be updated to multiple modes such as night and day circle which will affect zombie's action, multiple players mode, and more promising features. The game now places weapons and necessary tools for the player in different places, the player needs to go around and find them to survive the game. Overall, the 3D game will recreate the setting of a post-apocalyptic with a haunting soundtrack and realistic sound effects that immerse the player in the game world.

3.2 Design and requirements

When making a game project, that project needs to be clearly planned and implemented step by step. This will make the game-making process easier and more efficient. The first step of game design is to define the core of the game. One of the most important of the game which all developers need to consider first is player experience. Because this game is intense, scary, and fast-paced, the player's agility and vision are of paramount importance. The idea of this game is to keep the player focus on

the game and complete the stages of the game. Next, the core mechanic of this game is shooting zombies to survive, and the core game loops are included collecting suppliers, shooting the enemy, and reaching the end of the level.

The second step in the project's plan is choosing art assets and sound effects for the zombie game which are the most important components to making the interesting game. In Unity's asset store, there are available numerous assets that are free and commercial including textures, and animations. There are some free assets for developers to use, but while developing the game, several moments when developers require high-quality assets which are not free to use, this will affect the design phase where the plan needs to be changed due to unavailable assets. Moreover, this weakness can be challenging for new developers when their budget is limited. However, in case they have some knowledge of graphic design and Photoshop, this problem will be solved easily.

Scripting is an integral part of any game development and in this game. The game cannot operate without scripting, and it is written in C# language. After the scripts are attached to the Game Objects, the game will start to function in the game accordingly. In Unity, the default built-in Integrated Development is MonoDevelop but in this project, Visual Studio Code will be used for scripting.

3.3 Scripting

In Unity, to modify or add velocity, motion, and other functions to the game's character, a script folder needs to be added to the Game Object which developers want to write some code on it. Almost applications need scripts to respond to the gameplay, moreover, they can be used to create effects, and easily controlled objects' behaviors or implement Artificial Intelligence for characters in the game. In Unity, the Script folder can be created simply by right-clicking on the Project window, choosing Create, and then clicking on C# Script to create the new folder as in Figure 2 below.

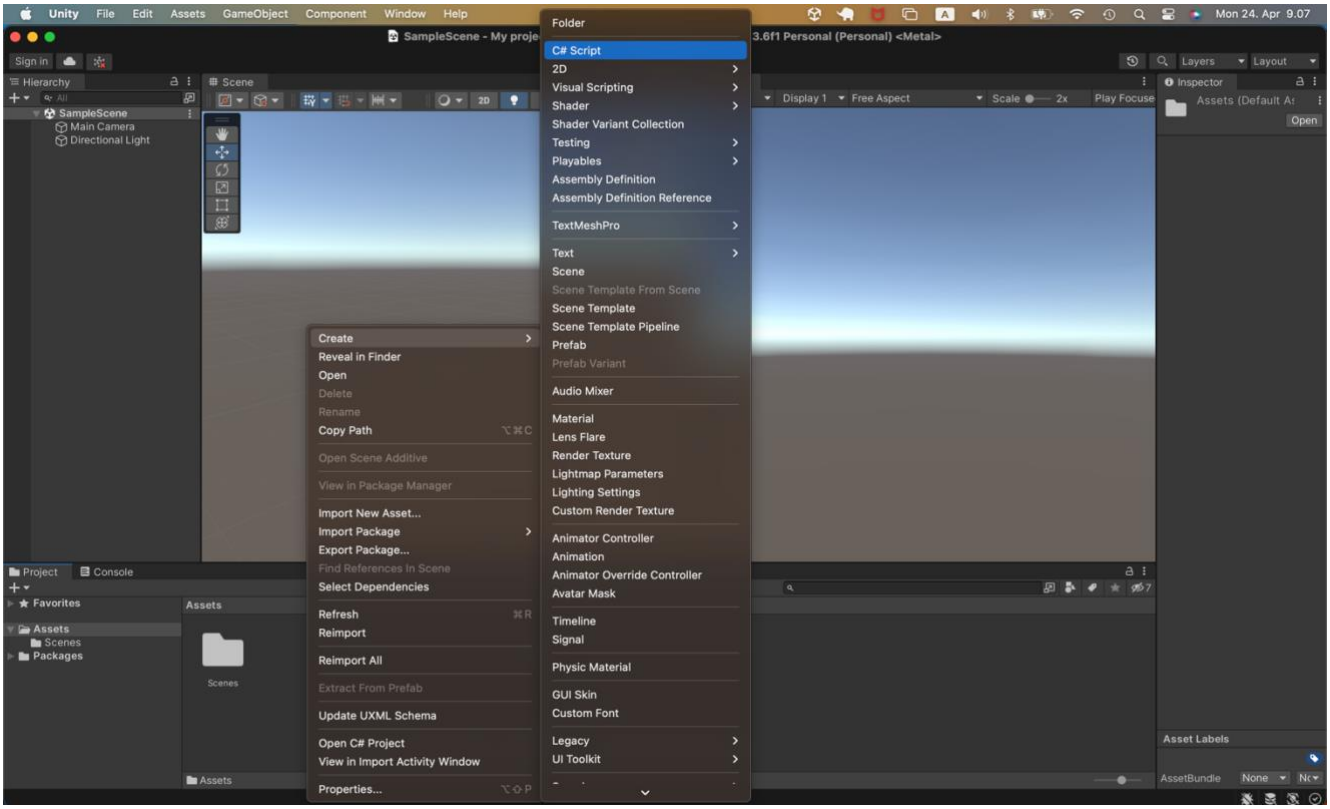


Figure 2. Create C# Script folder.

After creating a new script, developers can rename the script as seen in Figure 3 below, the script named “NewBehaviourScript” and then double-click on it to open. On opening a script, it is shown that the class created derives from the base class MonoBehaviour which provides developers with a template script, and useful functions such as Start and Update for their game. These two functions are called immediately such as the Start() function is called before the first frame update and the Update() function is called once per frame.

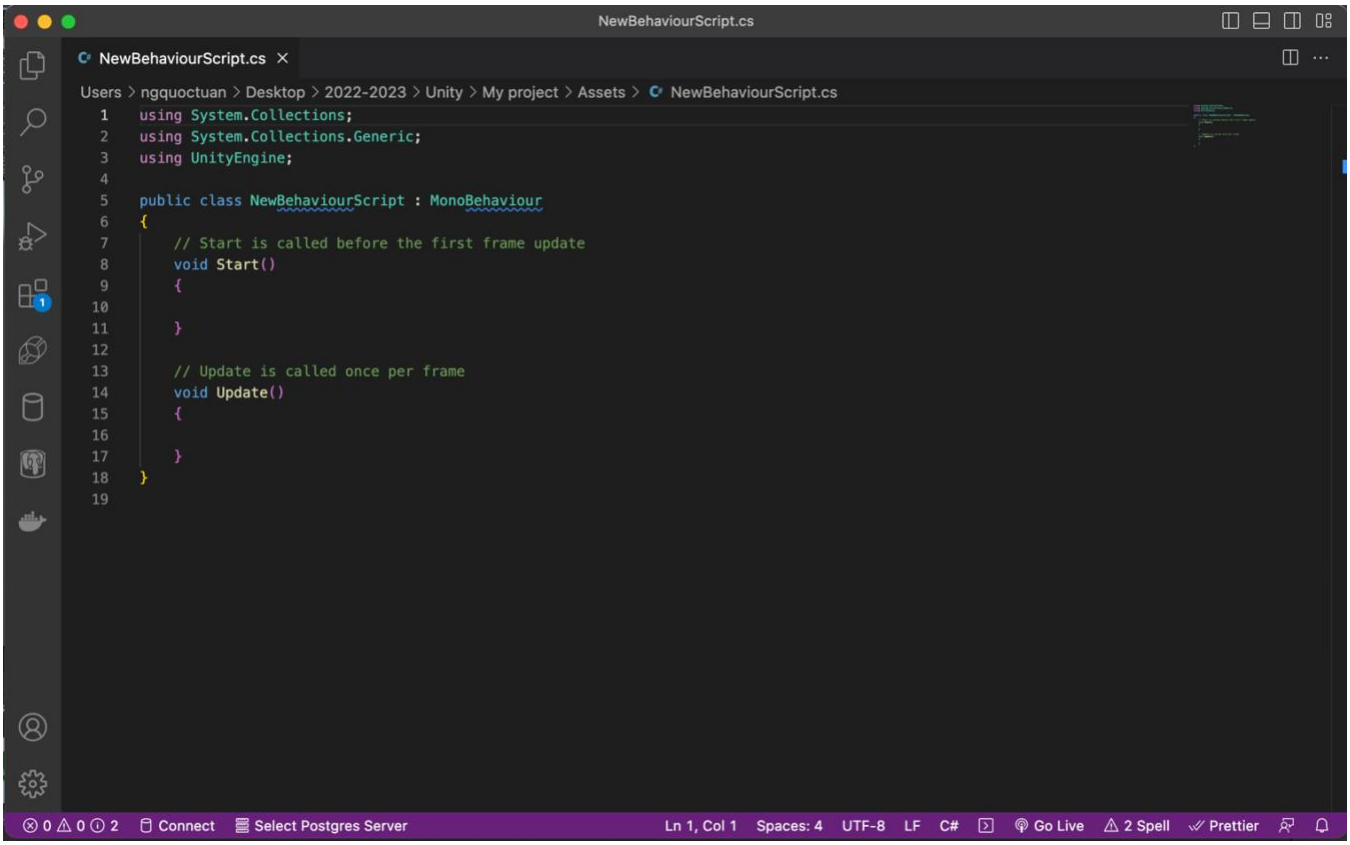


Figure 3. Default of Unity C# script.

Developers can fix properties such as measuring time elapsing, adding velocity, and modifying transformation in this script. After that, those changes need to be saved so that Unity can know which properties are changed in this script. However, those changes will not affect to the game if the script is not attached to any GameObject. There are two ways to attach it to a GameObject.

The first way, as known as the easiest way by dragging a script directly to the GameObject from the Project window. The second way to add a script to the GameObject is by clicking on the GameObject. It will display a list of components. As seen in the Figure 4 below, developers can see the “Add Component” button at the end of the window they can add a script by clicking on it and it will appear with the list of possible components that can be added including the script component.

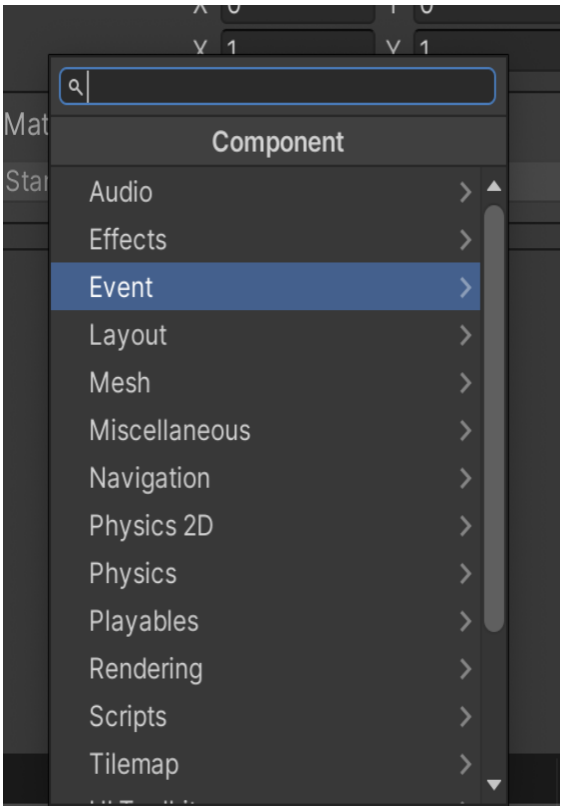


Figure 4. “Add Component” window.

3.4 Navmesh

In Unity, the navigation mesh as known as Navmesh is an important component that allows non-player controllers (NPCs) or AI-controller characters to move intelligently and autonomously in the game. To enable the NPCs or AI-controlled characters to navigate, they need to be added to the chosen game object the new component in the Inspector window, named “NavMeshAgent”. After adding, those characters can be able to move in the navigation area. The “NavMeshAgent” component gives the characters the ability to avoid obstacles while following their target.

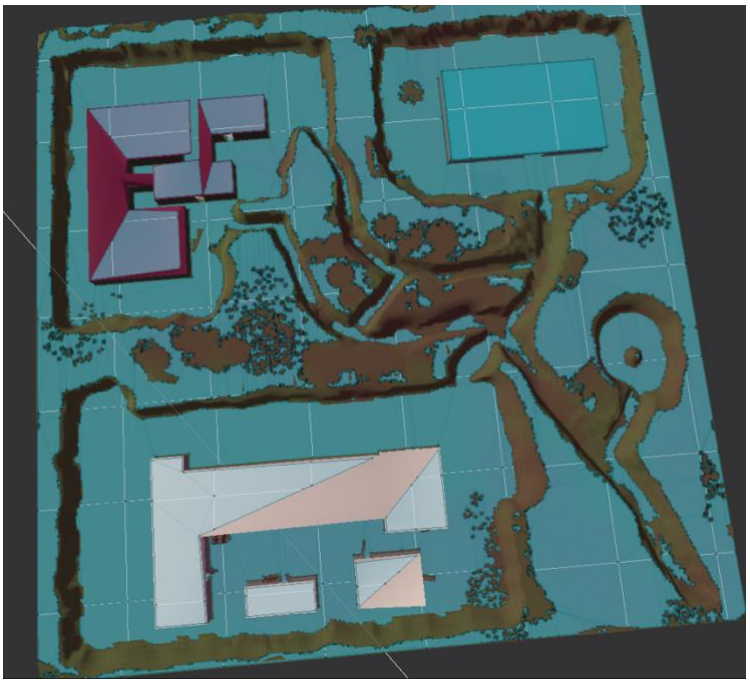


Figure 5. Game map with the navmesh.

As seen in Figure 5, everything on the game map is colored blue. It means the navigation mesh agent can move around the map. If the developers want to build the navigation mesh, those objects need to be marked static. The navmesh agent can only move on static objects. To change the mechanism, the developers choose the game objects that they want the agent to move and should affect the navigation, then mark the static in the top-right in the Inspector window of selected game objects.

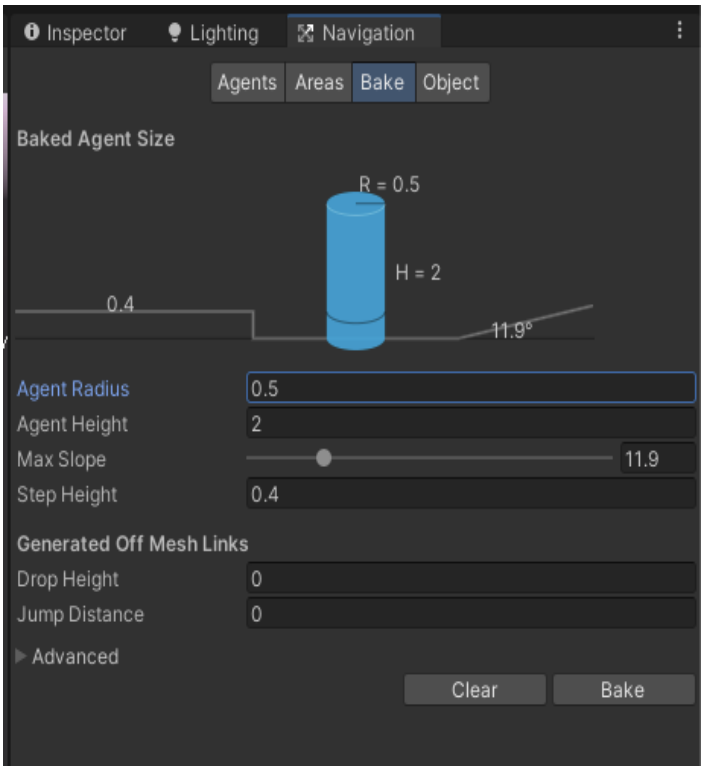


Figure 6. Navigation window.

In the navigation window (see Figure 6), the developers can adjust the bake setting. In the settings, they are allowed to alter the agent size such as agent radius, agent height, max slope, and step height. The agent radius describes the distance that the navigation mesh agent can reach to a wall or a ledge, agent height is the value that describes how low the agent can reach. Max slope and step height explain how steep and how high obstructions the developers allow the agent to walk up. After adjusting the settings, the developer clicks the bake button to build the navigation mesh.

4 IMPLEMENTATION

This section will explain in detail the game implementation and it can be used as an example on creating and developing 3D game. This section showed the step by step of a game development. The basics knowledge and important scripts will be explained clearly as a guidance for creating a zombie game. All the scripts used in this project can be seen in the Appendices.

4.1 Create a scene

In Unity, to create a game scene, the first requirement that install Unity Hub. It is an application for users to manage their recent projects. After completing installing Unity Hub and opening it, users will see the Unity Hub window as in Figure 7 below. To create a project by clicking the “New project” button, it will display all necessary templates for developers such as 2D, 3D, VR, First Person, and Third Person, and sample templates for those who are new to Unity. This project is aimed at a 3D game so a 3D template will be chosen. When creating the project, naming, and choosing the location of the project are needed. For this project, the game is decided to be called The Last Run.

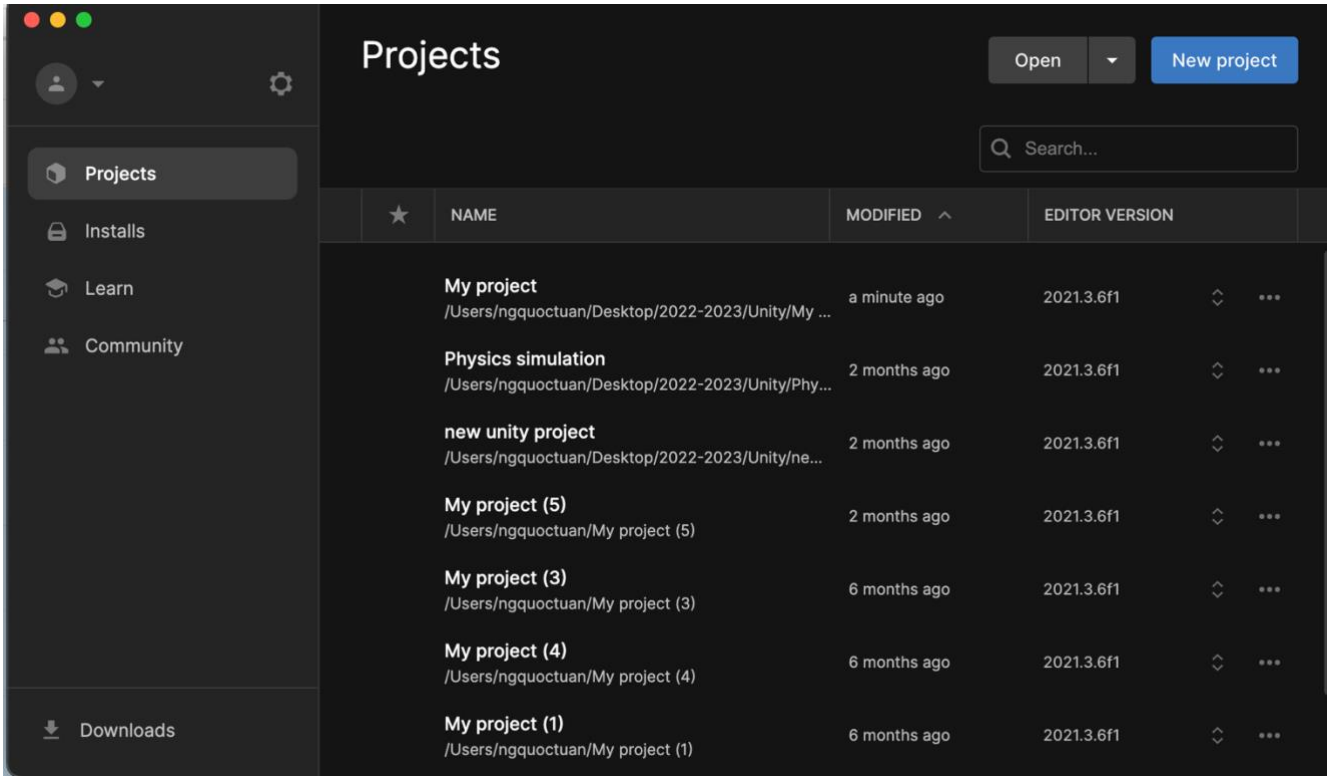


Figure 7. Unity Hub window.

4.2 Player

A first-person controller is a script that helps the player control the movement and behavior of their character from the first-person perspective. There is numerous first-person shooters and other games which require precise movement and aiming, for example, Call of Duty 4, Half-life 2, Counterstrike, Left 4 Dead 2. (Matthew Byrd 2021.) Unity provides a built-in first-person controller that can be added to the scene. It helps creators make immersive and engaging gameplay experiences, which is the reason the first-person controller is an important part of many Unity games.

As mentioned above, the Asset store is powerful and helpful for developers in process of developing their projects. To implement the first-person controller, this project needs access to the Asset store to install the crucial asset as can be seen Figure 8 below to the package manager and then import it to the project.

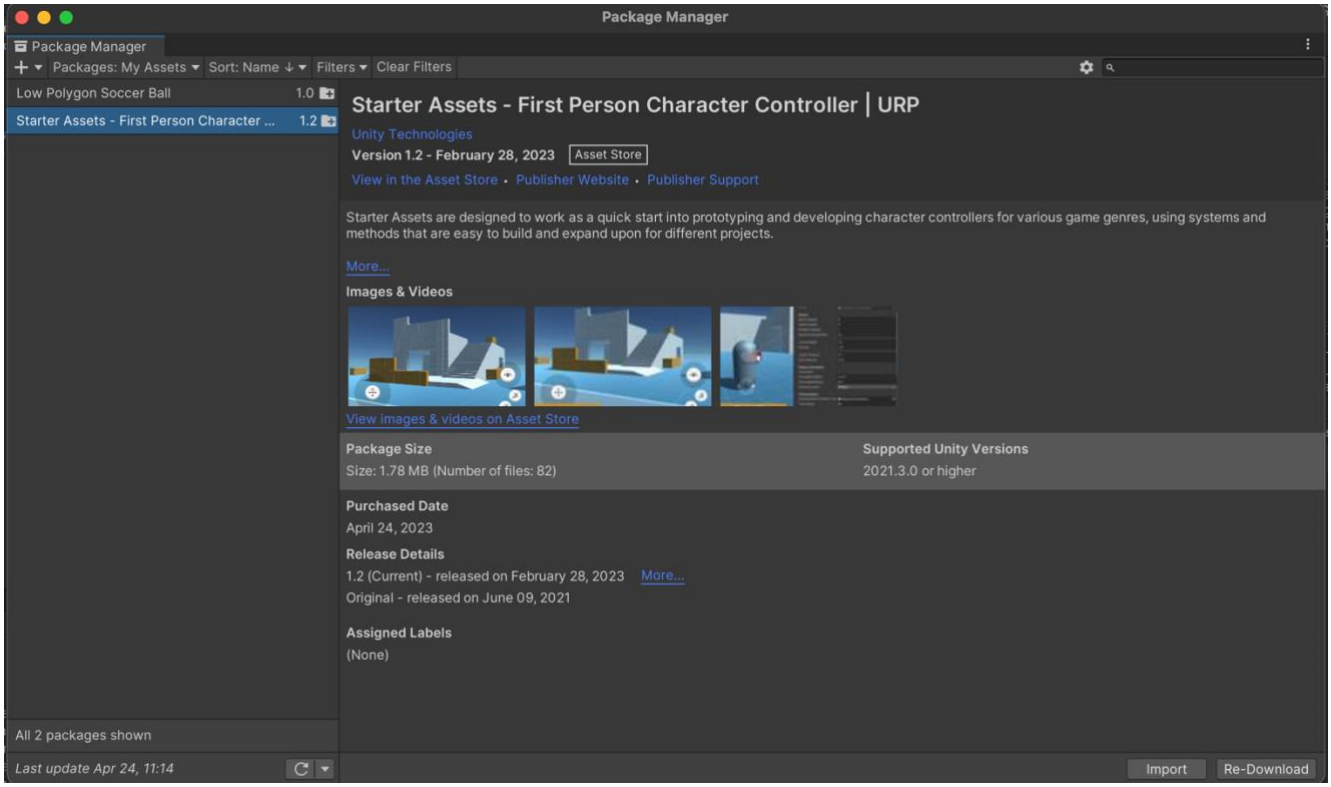


Figure 8. Package Manager window.

After successfully importing this asset to the project, it can be seen in the Project window as the StarterAssets folder as in Figure 9 below. This folder will display a group of folders, choose the folder named FirstPersonController, and then click the prefabs folder to see a list of different options to add the first-person perspective to the scene. The default main camera of the project can be deleted because

there is another main camera in the FirstPersonController folder, the default camera is not needed. In the prefabs folder, a list of options can be dragged directly as below figure to the Hierarchy window to activate the first-person controller. With the addition of the asset from Asset store, the character is possible to do basic functionally such as movement, jumping, and looking around from a first-person perspective in the scene after pressing Play button. (DocFX 2023.)

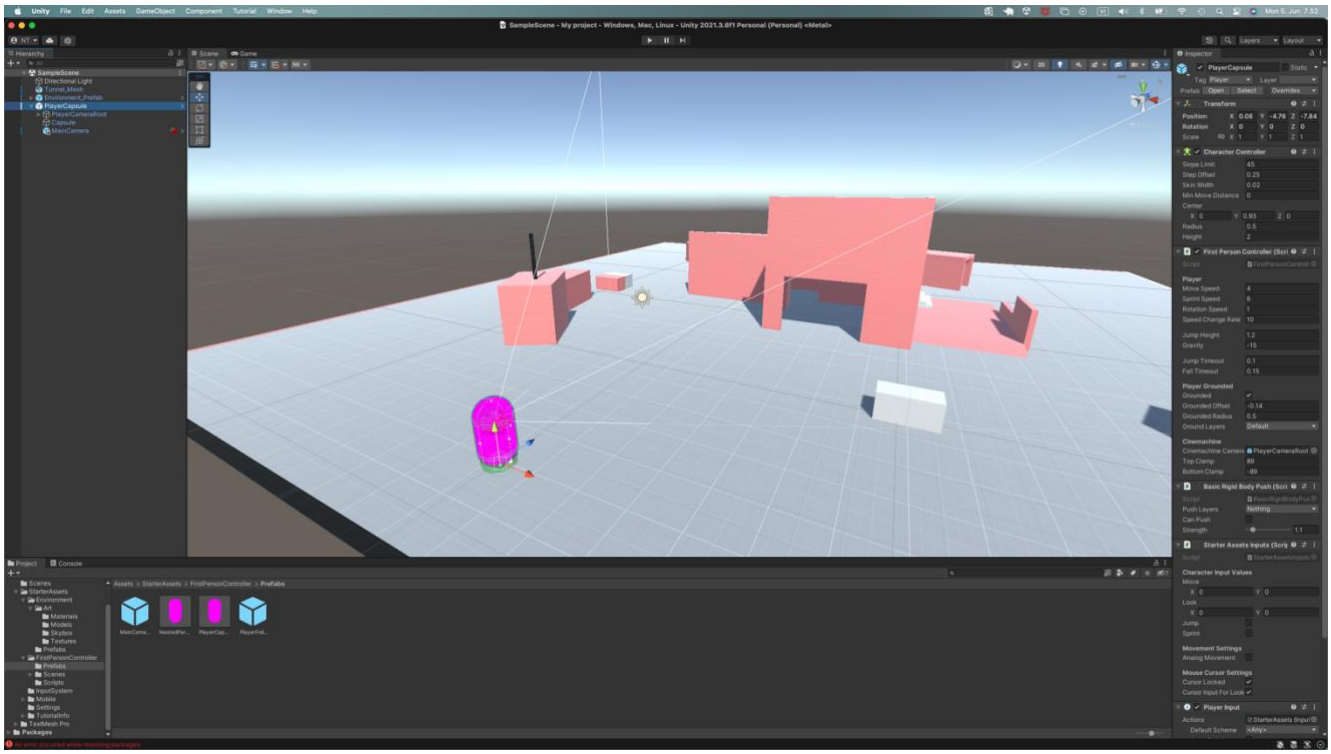


Figure 9. First-person controller added to the scene.

4.2.1 Player's health

This section pertains to the player's health, which is a crucial aspect of both the player and the zombies. It indicates the number of health points the player has remaining. The player's health value is predetermined by the code and is displayed as a bar at the top of the game scene. It decreases when the player is attacked by enemies. Once the health value reaches zero, the game ends, and the game over user interface is displayed. To survive until the end of the game, the player must be cautious and alert. In the upcoming update, players will have the opportunity to increase their health by discovering and acquiring support items.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement;
5  using UnityEngine.UI;
6
7  public class PlayerHealth : MonoBehaviour
8  {
9      public float health = 100f;
10     public float maxHealth;
11     public Image healthBar;
12
13     void Start()
14     {
15         maxHealth = health;
16     }
17
18     void Update()
19     {
20         healthBar.fillAmount = Mathf.Clamp(health / maxHealth, 0, 1);
21     }
22
23     public void TakeDamage(float damage)
24     {
25         health -= damage;
26         if (health <= 0)
27         {
28             SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
29             Cursor.lockState = CursorLockMode.None;
30             Cursor.visible = true;
31         }
32
33     }
34 }
```

Code 1. Player's health script.

In Code 1 above line 10, the player's "health" is assigned a value of 100. But to make the health bar which will be explained in detail in the latter part of this thesis, the second float for "maxHealth" need to be created. In line 14, two properties need to be equal because the maximum of the player's health needs to be 100 from the beginning of the game and it cannot be changed during the game, the "health" value will be changed while the player is playing. It means the "health" value is set to clamp between zero and "maxHealth" value. To use to restrict the fill amount to the minimum and maximum value, the "Mathf.Clamp" is used to clamp between 0 and 1 in line 20. For example, if the health is set as 100, the minimum and maximum are 0 and 100, and the player's health cannot go higher or lower than those values.

4.3 Enemy

In this section, non-player characters the zombies, will be explained in detail about their functionalities. These characters work according to the script that is attached to them. The attached script plays as the enemy's artificial intelligence and their health, the enemy can engage and attack who provoked them, or when the player comes to a limited zone, the zombie will be awaked and chase the target until they get.

Figure 10. Zombie's character.

Figure 10 shows the non-player character, which is the enemy of the player in this game. Zombie's character is used in this game which is a scary character with a health bar attached to them. The animations are used to make the game more interesting, and lively when they are standing, moving, attacking, and when they are destroyed.

4.3.1 Enemy AI

Zombies are a significant part of the game, serving as opponents to the player. It is essential for the player to eliminate all the zombies as they can suddenly appear and attack the player. In this game, the

zombie is a non-player controller so they need a script that will be attached to them as their brain, named “EnemyAI”. That script will let them know how strong they are, how fast they can run, or who will be their target, they know how to protect themselves whenever their area is infringed. There are several types of zombies that were created to hinder the player, it does not do much damage but with their amount, they will take the player's health bit by bit.

```
4   using UnityEngine.AI;
5   using UnityEngine.UI;
6
7   public class EnemyAI : MonoBehaviour
8   {
9       [SerializeField] Transform target;
10      [SerializeField] float chaseRange = 5f;
11      [SerializeField] float turnSpeed = 5f;
12      [SerializeField] float damage = 40f;
13
14      PlayerHealth objective;
15      NavMeshAgent navMeshAgent;
16      float distanceToTarget = Mathf.Infinity;
17      bool isProvoked = false;
```

Code 2. References in EnemyAI’s script.

In Code 2 line 9, the reference of the target had been created. Because the enemies need to know that the player is their target in this game, the “SerializeField” attribute is used to display and allow developers to modify public variables in Unity’s Inspector of the Enemy which are written in the script. The “Transform” type is called because the enemy will be looking for a transformation of the game object which is dragged to the “Target” box in the Inspector window.

Next, the distance from the enemy to the target is created in line 16, the purpose is that if the target comes to a certain distance, the enemy will be provoked and then follow the player to attack. This variable will be initialized by “Mathf. Infinity” which means it is a gigantic number, so it was set as a starting point. If “Math.Infinity” is not used, the enemy will chase the target immediately after the Play button is hit.

```
void Start()
{
    navMeshAgent = GetComponent<NavMeshAgent>();
    objective = FindObjectOfType<PlayerHealth>();
    health = GetComponent<EnemyHealth>();
}
```

Code 3. Start function in EnemyAI's script.

The reference for the navigation mesh agent was also made which was called "navMeshAgent", and it is a component. To access to the navigation mesh, the script needed to include a sentence "using UnityEngine.AI" in Code 2 line 4. The navigation mesh agent was set up in the Start function (see Code 3) instead of the Update function because it only needs to be called only once. All the enemies in this game will have a script attached to them. Some of them will have different scripts to make the game more challenging. For example, the final boss will have more damage, a different range of chase, or speed when it is chasing the player.

```
28      void Update()
29      {
30          distanceToTarget = Vector3.Distance(target.position, transform.position);
31
32          if (health.IsDead())
33          {
34              enabled = false;
35              navMeshAgent.enabled = false;
36          }
37
38          if(isProvoked)
39          {
40              EngageTarget();
41          }
42          else if(distanceToTarget <= chaseRange)
43          {
44              isProvoked = true;
45          }
46      }
```

Code 4. Update function in EnemyAI's script.

Code 4 shows the simple code which measures the distance from the enemy to the player. To measure the distance, it is necessary to use Vector 3 which is a part of the UnityEngine namespace. Vector3 is a data structure that represents three-dimensional axes respectively including horizontal, vertical, and depth. The method is equal to "distanceToTarget" which means this method will return the distance between the position of the target and the enemy's transform position will be measured in every frame.

This function also presents that if the enemy is provoked by the player, it will engage the target immediately. The “IsProvoked” Boolean is created in Code 2 line 17, which can be turned into true or false. The value is false in the beginning, it means the enemy will stand or do their animations. Then if the distance from the enemy to the target is smaller than the value of the range of chase, the “IsProvoked” will change the value to true, the “EngageTarget()” function will be active, and the enemy will start to chase the target. For example, the chase of the range is assigned 5 initially, if the value of “distanceToTarget” is 4, the enemy will be started to hunt the player.

```
void OnDrawGizmosSelected()
{
    // Display the explosion radius when selected
    Gizmos.color = new Color(1,0,0,1);
    Gizmos.DrawWireSphere(transform.position, chaseRange);
}
```

Code 5. “OnDrawGizmosSelected” function from the EnemyAI’s script.

This script is created to make a visual representation so that the range of the enemy is visible to modify. If the player moves into the range, the enemy will be noticed and start to chase the player. After implementation, the gizmo will be drawn as a wireframe with the center and radius of the game object selected. This function allows the user to adjust the color of gizmos, the developer can choose a specific color or create new color as above figure. The second sentence displays that the gizmos will be drawn as a wire sphere, then the sphere has the center point which is the enemy’s position, and the radius, in this case, is the “chaseRange”.

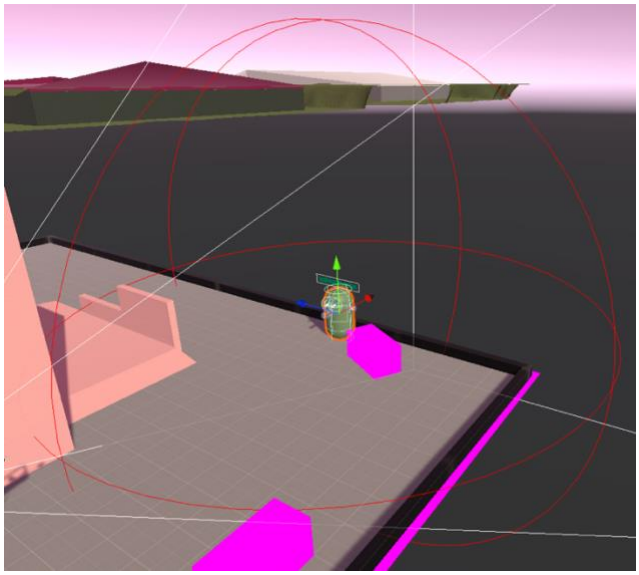


Figure 11. the “chaseRange” of the enemy.

```
private void EngageTarget()
{
    FaceTarget();
    if(distanceToTarget >= navMeshAgent.stoppingDistance)
    {
        ChaseTarget();
    }
    else if(distanceToTarget <= navMeshAgent.stoppingDistance)
    {
        AttackTarget();
    }
}
```

Code 6. the “EngageTarget” function from the EnemyAI’s script.

In Figure 17, the “EngageTarget()” function is called, whenever the enemy is provoked. The “stop-pingDistance” of the navigation mesh agent sets an acceptable stopping radius for the agent to stop within a certain distance from the target position and gives an agent more space for attacking. The function showed that if the value of the distance from the enemy to the target is greater than or equal to the value of the stop-ping radius, the “ChaseTarget()” function will be enabled for the enemy to hunt the target. On the contrary, if the distance is close enough to the target, the “AttackTarget()” will be operated for the enemy to hit the player. In Code 7 below, the function shows a simple code. The “GetComponent” lines will be talked about latter in this enemy’s section. The third line shows that the navigation agent called the “SetDestination()” to locate the target’s position.

```
private void ChaseTarget()
{
    GetComponent<Animator>().SetBool("attack", false);
    GetComponent<Animator>().SetTrigger("move");
    navMeshAgent.SetDestination(target.position);
}
```

Code 7. “ChaseTarget” function from the EnemyAI’s script.

```
private void FaceTarget()
{
    Vector3 direction = (target.position - transform.position).normalized;
    Quaternion lookRotation = Quaternion.LookRotation(new Vector3(direction.x, 0, direction.z));
    transform.rotation = Quaternion.Slerp(transform.rotation, lookRotation, Time.deltaTime * turnSpeed);
}
```

Code 8. “FaceTarget” function from the EnemyAI’s script.

In Code 8, the “FaceTarget” function is created to help the enemy rotate and locate exactly and move naturally to their target’s position. The “direction” variable was created with the “Vector3” type, which is used to take the direction of the “normalized” magnitude from the vector taken from the subtraction of the target’s position and the enemy’s position. The “normalized” helps the enemy’s direction to keep the same direction within a magnitude of 1. The next variable is “lookRotation” with the type of “Quaternion”, which is used to represent all rotations of the enemy, the creation of the “new Vector3” that is de-rived from the “direction” variable. The “lookRotation” allows the enemy to look left and right, that’s based upon the new vector 3 within a magnitude of 1. The last one is the transformation of the rotation, the “Slerp” method of “Quaternion” stands for the spherical interpolation between two quaternions by the ratio of time and speed. The method allows the enemy to rotate smoothly with a suitable speed in an independent time. The “FaceTarget” function is called in the “EngageTarget” function as Code 6, which means whenever the enemy is engaging the target, the enemy always focuses on their target.

4.3.2 Enemy’s health

With the enemy health, the enemy was attached with another script, called “EnemyHealth” script. The health’s script of the enemy is simple same as the player’s health. Whenever the enemy was shot by the player, their health is decreased by the amount of damage inflicted. In Code 9 below, each enemy has the same health point except the boss which will have different health and damage. “hitPoints” means the enemy’s health, which is assigned as 100 at the beginning. The “hitPoints” variable is public for the developer to modify and test the game comfortably. The Start and Update functions were created exactly the same as the “PlayerHealth” script to access and control the UI. If the enemy is losing their health, the health bar of the enemy will be losing at the same time.

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

public class EnemyHealth : MonoBehaviour
{
    public float hitPoints = 100f;
    public float maxPoint;
    public Image enemyHealthBar;

    bool isDead = false;

    public bool IsDead()
    {
        return isDead;
    }

    void Start()
    {
        maxPoint = hitPoints;
    }

    void Update()
    {
        enemyHealthBar.fillAmount = Mathf.Clamp(hitPoints / maxPoint, 0, 1);
    }

    public void TakeDamage(float damage)
    {
        BroadcastMessage("OnDamageTaken");
        hitPoints -= damage;
        if (hitPoints <= 0)
        {
            Dead();
        }
    }
```

Code 9. “EnemyHealth” script.

The “TakeDamage()” function was created and made as a public function. The method is called whenever the health reduces heath point by the damage of the player’s weapon. The damage of the weapon was called in parentheses of this function with a type of float. It means when the enemy receives damage, the “TakeDamage” function will be called. In case, the enemy’s health is out of point, the enemy will be dead and fall to the ground immediately. The “BroadcastMessage” is used in “EnemyHealth” as a method to announce to the enemy that they are shot by the player and that they

should start to chase the player immediately. The “BroadcastMessage” component uses a string reference, named “OnDamageTaken”, which was created in “EnemyAI”.

```
public void OnDamageTaken()
{
    isProvoked = true;
}
```

Code 10. “OnDamageTaken” function from the EnemyAI’s script.

4.3.3 Enemy’s damage

```
public void AttackHitEvent()
{
    if (objective == null) return;
    objective.TakeDamage(damage);
    Debug.Log("bang bang");
}
```

Code 11. “AttackHitEvent” function from the EnemyAI’s script.

In Code 2 line 12, the variable of damage was initialized at 40, that is the amount of damage which the enemy will make the player’s health lower. In the same code line 14, the objective was created as a type of player health. Therefore, the health component can be accessed, the objective will be a type of the player’s health. Then in the Start function, it means in the beginning, the objective will be found by “FindObjectOfType” method, and the type is “PlayerHealth”. Next, the “AttackHitEvent” function in Code 11, is created to call from the animation event. This function will attach to the attack animation to be able to activate the animation and hurt the player at the same time. The “Debug.Log” will print the message in the parentheses to the console, the purpose of this line is to ensure that the enemy is hurting the player.

4.3.4 Enemy Animations

In this game, most of the enemies had the same animations and the same principles. To enable the animation of the Game Object, the Animator component needs to be added to the character. At the early stage of this game, the zombie character is created as an example to run through the code, and then use the zombie asset from the Asset store. After the code runs smoothly, the enemy is ready to chase the target. The zombie asset will be used and dragged to the enemy, after that it will be used as a prefab to recreate the enemy because it is extremely time-saving and effective for all the developers to reuse the Game Object.

The zombie has 4 animations in this game, “Idle”, “Move”, “Attack, and “Dead”. The “Idle” animation is the initial animation of the zombie at the beginning. The next animation will be enabled when the “Idle” ends. The setting of transition and transition duration of the animations will be set in the Inspector window by clicking on the animation transitions. To make the transition between two animations, simply right-click on the state and choose “Make Transition” as the Figure 18.

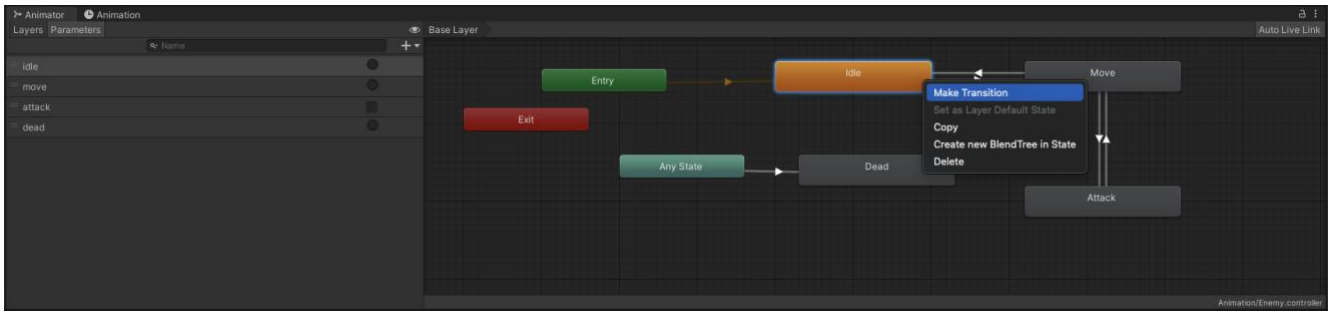


Figure 12. Animator window.

As seen in Figure 12, the idle, move, and attack animations were connected by transitions. The “Dead” animation is linked to any state, which means whenever the parameter “dead’ is set to trigger, the animation will animate. If the enemy’s health is equal to 0, it will be felled to the ground. Since the enemy character had more than one animation, it is required to create parameters to run those animations smoothly when the functions are called. To trigger the transition from one stage to another stage, the condition is needed. To make a condition, parameters must be added to the transition to use as the condition in the Animator window as shown in Figure 12 and Figure 13. The parameters have 4 types including Float, Bool, Int, and Trigger. In this game, the Bool and Trigger types were mostly used.

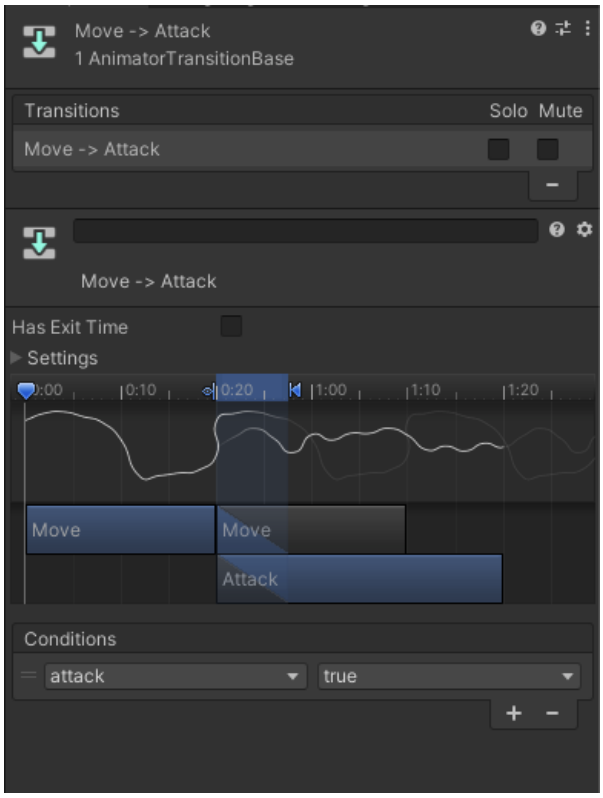


Figure 13. Conditions of Transition.

Figure 13 shows the conditions of the “Move -> Attack” transition. It did not have an exit time so that it can be run until the condition is enabled. The “attack” parameter is created already which can be used as a condition for the transition. In Figure 13, the “Move” animation will turn to the “Attack” animation when the “attack” parameter is true. The “idle”, “move”, and “dead” parameters are “Trigger” type, and only the “attack” is the “Bool” type. To set a trigger or change the state of the animation, those values need to be called as seen in Code 7 and Code 12.

```
35      bool isDead = false;
36
37      public bool IsDead()
38      {
39          return isDead;
40      }
41
42      private void Dead()
43      {
44          if (isDead) return;
45          isDead = true;
46          GetComponent<Animator>().SetTrigger("dead");
47      }
48  }
```

Code 12. Codes from EnemyHealth's script.

Code 12 shows a function that is called whenever the "hitPoints" is equal to 0 as in Code 9. To call the "Dead" animation, which is attached to the enemy, it is required to get the animator component and set the "dead" parameter to trigger. The "isDead" Boolean is created to check if the enemy is dead or alive. The "IsDead" function is created in Code 12, to be accessed in the Update function of the EnemyAI's script. It means if the enemy is out of health, the "isDead" will turn to true, then the "dead" animation will be set to trigger, and the enemy component and the navigation mesh of that enemy will be disabled as can be seen in Code 13 below.

```
if (health.IsDead())
{
    enabled = false;
    navMeshAgent.enabled = false;
}
```

Code 13. if statement from the EnemyAI's script.

4.4 Weapon

This section will discuss the player's weapon and suppliers such as ammo, and light. Support items are important components to support and help the player fight with their opponents. It will help players to survive in the dark forest. In the beginning, the player will have 3 weapons already, which are a pistol, shotgun, and AK-47, they can be chosen by pressing the "1,2,3" buttons on the keyboard. The weapon was created with its effect, which can be seen whenever the player shot.

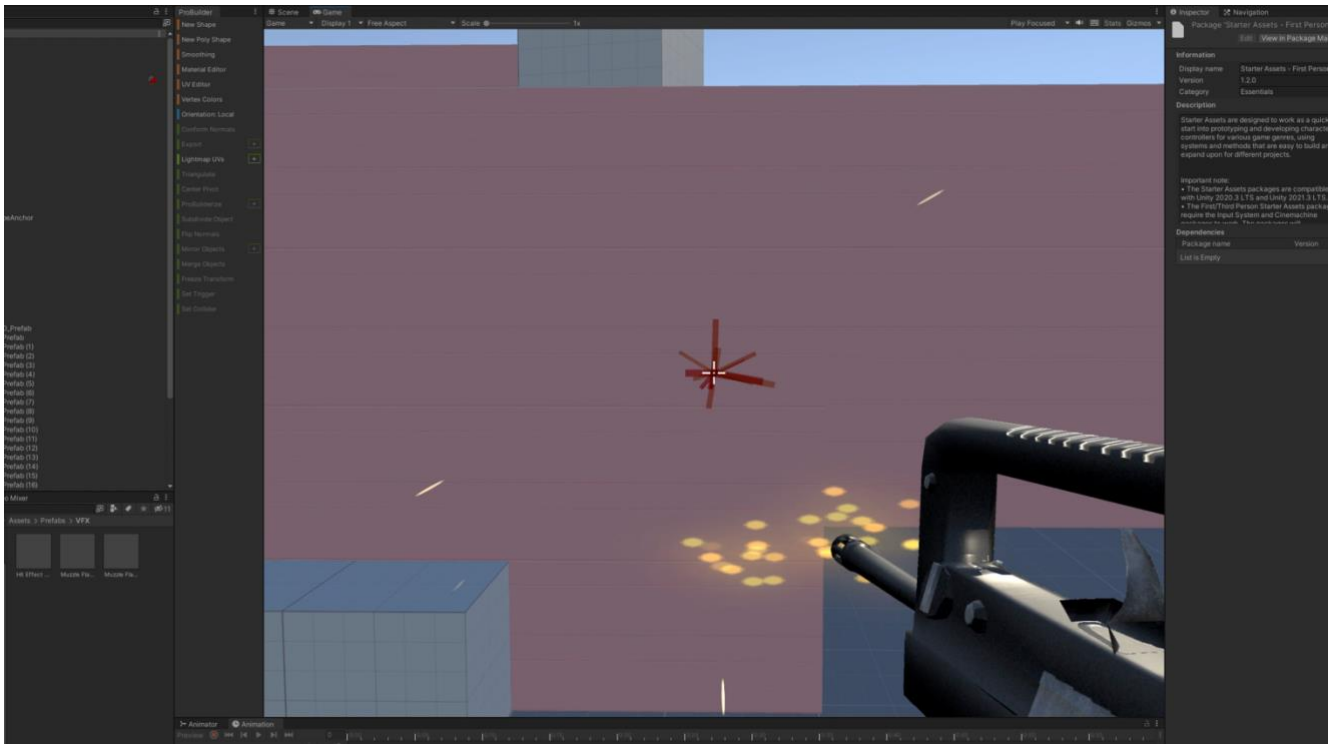


Figure 14. A player is shooting in the Game Scene.

Figure 14 shows the Raycasting from the gun to the wall in a straight direction. Raycasting is a process of shooting an invisible ray from an original point, in a specified direction, against all colliders in the Game Scene (Unity Documentation 2021). It also displays the effects when the weapon is shot. The effect is shown at the top of the gun, and in the wall that is shot by the weapon. To activate the effect on the weapon, the script attached to the weapon is needed, it is the “Weapon” script. To create the effect for the gun while it is shooting, it is necessary to use the effect from the Particle System. To create the particle effect, simply right-click on the Hierarchy window, choose “Effect” and select “Particle System”.

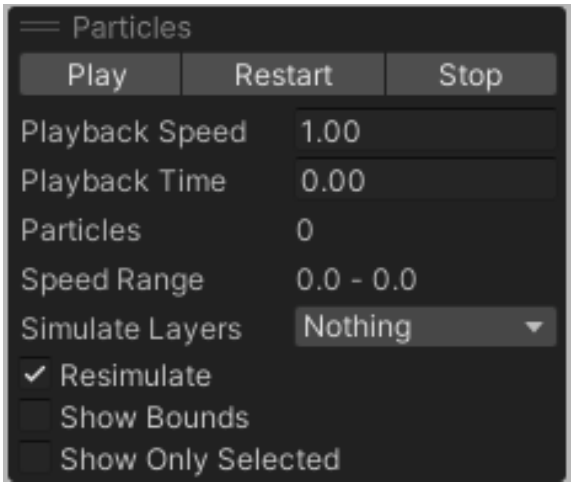


Figure 15. Particle’s window.

After successfully created, the developer can adjust the Particle System on the Inspector window. The developer can adjust the speed and the range of the system. The developer can be able to check if the particles work as expected in the small window on the right-bottom corner of the scene view as shown in Figure 15. The particle effects are called “Muzzle Flash” and “Hit Effect” in this game. The “Muzzle Flash” is shown when the gun is shot, and the “Hit Effect” is shown on the Game Object shot by the weapon.

```
4   using UnityEngine;
5   using UnityEngine.UI;
6
7   public class Weapon : MonoBehaviour
8   {
9       [SerializeField] Camera FPCamera;
10      [SerializeField] float range = 100f;
11      [SerializeField] float damage = 30f;
12      [SerializeField] ParticleSystem muzzleFlash;
13      [SerializeField] GameObject hitEffect;
14      [SerializeField] Ammo ammoSlot;
15      [SerializeField] AmmoType ammoType;
16      [SerializeField] float timeBetweenShots = 0.5f;
17      [SerializeField] Text ammoText;
18
19      bool canShoot = true;
```

Code 14. Codes from Weapon’s script.

This game is a first-person controller, so the weapon needs to have the same camera as the player’s camera, it is called “FPCamera” with a type of Camera in Code 14 line 9. The range and damage of the weapon are required because weapons will have different range and damage that is suitable for each weapon. There are various variables of the weapons that will be explained later in this section.

```
59  private void ProcessRaycast()
60  {
61      RaycastHit hit;
62      if (Physics.Raycast(FPCamera.transform.position, FPCamera.transform.forward, out hit, range))
63      {
64          CreateHitImpact(hit);
65          EnemyHealth target = hit.transform.GetComponent<EnemyHealth>();
66          if (target == null) return;
67          target.TakeDamage(damage);
68      }
69      else
70      {
71          return;
72      }
73  }
```

Code 15. “ProcessRaycast” function from Weapon’s script.

Code 15 shows how the process of ray cast works and hurts the enemy. The function executes the weapon's ray cast by casting a ray from the player's camera position in the direction they are looking and checking if it hits anything within a certain range. The "RaycastHit" variable is used to store the information on the type of ray cast. As seen in Figure 31 line 65, the "ProcessRaycast" function accessed to the "EnemyHealth" to give the weapon's damage to the enemy. The function shows that if the player hit the "target", the weapon is enabled to take the damage on the enemy, on the contrary, if the player does not hit any Game Object, the function will be returned.

```
private void PlayMuzzleFlash()
{
    muzzleFlash.Play();
}
```

Code 16. "PlayMuzzleFlash" function from the Weapon's script.

In Code 16, the function plays the "muzzleFlash" which was created with a type of "ParticleSystem" to activate the particle effect which named "Muzzle Flash" in the game in Code 14 line 12. As seen in Code 15, the "CreateHitImpact" was called in the "ProcessRaycast" function because the flash from the weapon will happen at the same time as its impact. The function instantiates a visual effect at the point of impact using a prefab called "hitEffect" which was created with a type of GameObject in Code 14 line 13.

```
private void CreateHitImpact(RaycastHit hit)
{
    GameObject impact = Instantiate(hitEffect, hit.point, Quaternion.LookRotation(hit.normal));
    Destroy(impact, 1);
}
```

Code 17. "CreateHitImpact" function from the Weapon's script.

```
21      private void OnEnable()
22      {
23          canShoot = true;
24      }
25
26      void Update()
27      {
28          DisplayAmmo();
29          if (Input.GetMouseButtonDown(0) && canShoot == true)
30          {
31              StartCoroutine(Shoot());
32          }
33      }
```

Code 18. Codes from the Weapon's script.

In Code 18, the Update function allows the player can click the left mouse button to shoot. The "OnEnable" function was created with the bugging purpose. Whenever the mouse button is pressed, "canShoot" turns to true and it starts the "Shoot" coroutine. In Code 19 below, the "Shoot" function is a coroutine that is used to delay certain actions. The "canShoot" variable is set to false to prevent the player from shooting again. Next, the function will check if the current ammo is greater than 0, it will call "PlayMuzzleFlash" and "ProcessRaycast" functions and reduces the current ammo amount. After that, it needs to wait a time between shots which variable was created in Code 14 line 16 before the "canShoot" back to true.

```
IEnumerator Shoot()
{
    canShoot = false;
    if (ammoSlot.GetCurrentAmmo(ammoType) > 0)
    {
        PlayMuzzleFlash();
        ProcessRaycast();
        ammoSlot.ReduceCurrentAmmo(ammoType);
    }
    yield return new WaitForSeconds(timeBetweenShots);
    canShoot = true;
}
```

Code 19. "Shoot" function from the Weapon's script.

4.4.1 Weapon's type

In this part, the type of weapon in this game will be explained. The weapons have three types such as pistol, shotgun, and AK-47. It can be switched between these weapons by pressing the key number. The type of weapon dictates its potential damage output. There are different weapons in this game, it is required to differentiate each weapon by the number. The first weapon in the list has an index of zero, the next two weapons are 1 and 2. The player can select the weapon by inputting the key number from 1 to 3 based on the order of the above list.

```
8     [SerializeField] int currentWeapon = 0;
9
10    void Start()
11    {
12        SetWeaponActive();
13    }
14
15    void Update()
16    {
17        int previousWeapon = currentWeapon;
18
19        ProcessKeyInput();
20
21        if (previousWeapon != currentWeapon)
22        {
23            SetWeaponActive();
24        }
25    }
```

Code 20. Codes from the WeaponSwitcher's script.

In Code 9 line 8, the "currentWeapon" variable is created and initialized at zero. It means the first weapon the player will have from the start of the game is the pistol, and it is activated to control in the beginning. The Update function called the new variable "previousWeapon", it helps the player to change and activate the weapon by inputting the key code. Therefore, the player can change between the three types of weapons comfortably.

```
45      private void SetWeaponActive()
46      {
47          int weaponIndex = 0;
48
49          foreach (Transform weapon in transform)
50          {
51              if (weaponIndex == currentWeapon)
52              {
53                  weapon.gameObject.SetActive(true);
54              }
55              else
56              {
57                  weapon.gameObject.SetActive(false);
58              }
59              weaponIndex++;
60          }
61      }
```

Code 21. “SetWeaponActive” function from the WeaponSwitcher’s script.

The function goes through each weapon and set the selected weapon active. The function created a new variable “weaponIndex” to control the list of weapons. The “foreach” method is used to go through each weapon, called the weapon in transform. It means the current weapon is active, then the other weapons will be inactive. Code 22 shows that each weapon corresponds to a number on the keyboard. The pistol is selected when the player presses the key 1, the shotgun is going with the key 2, and the AK-47 is chosen with the key 3 on the keyboard. This function is called in the Update function to set the weapon active.

```
private void ProcessKeyInput()
{
    if (Input.GetKeyDown(KeyCode.Alpha1))
    {
        currentWeapon = 0;
    }
    if (Input.GetKeyDown(KeyCode.Alpha2))
    {
        currentWeapon = 1;
    }
    if (Input.GetKeyDown(KeyCode.Alpha3))
    {
        currentWeapon = 2;
    }
}
```

Code 22. “ProcessKeyInput” function from the WeaponSwitcher’s script.

4.4.2 Ammo

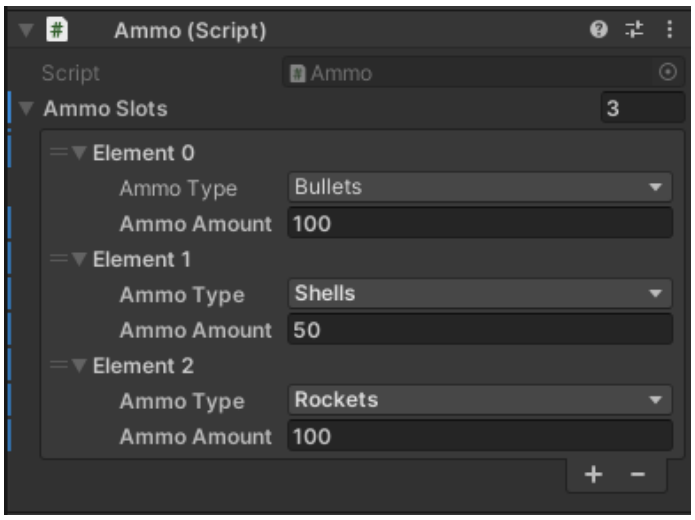


Figure 16. Ammo's script window.

Those weapons have their ammo, that divides into three types such as "Bullets" for pistol, "Shells" for shotgun, and "Rockets" for AK-47. It is required to create the structure for the ammo to divide clearly into three types and their amount in the ammo slot. Figure 16 displays the ammo's script which shows each type of ammo and its amount. The Ammo's script will attach directly to the Player.

```
1  public enum AmmoType
2  {
3      Bullets,
4      Shells,
5      Rockets
6  }
```

Code 23. AmmoType's script.

Code 23 shows an enumeration type called “AmmoType” with three values including Bullets, Shells, and Rockets. This enumeration has the “public” keyword, it means this function can be accessed from anywhere in the code. Code 24 line 11 shows a private nested class called “AmmoSlot”, which is used to store information about the type and amount of ammunition. The “System.Serializable” and “AmmoSlot [] ammoSlots” allowss the type and amount of the ammunition to be displayed as an array which can be edited in the Inspector window.

```
8       [SerializeField] AmmoSlot[] ammoSlots;
9
10      [System.Serializable]
11      private class AmmoSlot
12      {
13          public AmmoType ammoType;
14          public int ammoAmount;
15      }
16
17      public int GetCurrentAmmo(AmmoType ammoType)
18      {
19          return GetAmmoSlot(ammoType).ammoAmount;
20      }
21
22      public void ReduceCurrentAmmo(AmmoType ammoType)
23      {
24          GetAmmoSlot(ammoType).ammoAmount--;
25      }
26
27      public void IncreaseCurrentAmmo(AmmoType ammoType, int ammoAmount)
28      {
29          GetAmmoSlot(ammoType).ammoAmount += ammoAmount;
30      }
31
32      private AmmoSlot GetAmmoSlot(AmmoType ammoType)
33      {
34          foreach (AmmoSlot slot in ammoSlots)
35          {
36              if (slot.ammoType == ammoType)
37              {
38                  return slot;
39              }
40          }
41          return null;
42      }
43  }
```

Code 24. Codes from the Ammo’s script.

Code 24 line 17 and line 22 present two public functions that are “GetCurrentAmmo” and “ReduceCurrentAmmo”. The “GetCurrentAmmo” takes an “AmmoType” parameter and returns the amount of ammunition of that type. The other function also takes a parameter “AmmoType” and reduces the amount of ammunition of that type. The function as can be seen in Code 25 below has a private method, which takes an “AmmoType” parameter and returns the corresponding “AmmoSlot” object from “ammoSlots” array. If the function does not find “AmmoSlot” needed, it will return to null.

```
public int GetCurrentAmmo(AmmoType ammoType)
{
    return GetAmmoSlot(ammoType).ammoAmount;
}

public void ReduceCurrentAmmo(AmmoType ammoType)
{
    GetAmmoSlot(ammoType).ammoAmount--;
}
```

Code 25. “GetCurrentAmmo” and “ReduceCurrentAmmo” functions from the Weapon’s script.

4.4.3 Ammo pickup.

After completing to divide the ammo type with each weapon, the player still needs more ammunition to fight with the number of enemies. In this game, the ammo will be located around the map, the player needs to go around the map to find it. They are made into three prefabs for three types of ammo. The ammo suppliers will have different amounts, it has the amount from 1 to 10. To be able to pick the ammo up and use them, the script is needed to attach with them, named “AmmoPickup”.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class AmmoPickup : MonoBehaviour
6  {
7      [SerializeField] int ammoAmount = 5;
8      [SerializeField] AmmoType ammoType;
9
10     private void OnTriggerEnter(Collider other)
11     {
12         if (other.gameObject.tag == "Player")
13         {
14             FindObjectOfType<Ammo>().IncreaseCurrentAmmo(ammoType, ammoAmount);
15             Destroy(gameObject);
16         }
17     }
18 }
19
```

Code 26. AmmoPickup’s script.

Code 26 explains that when the player's collider enters the ammo object, the player's ammo will increase by a certain amount and type of ammo. The "ammoAmount" variable initializes at 5, it can be edited in the Inspector window. The "OnTriggerEnter" function is called whenever the player's collider enters the trigger zone of the ammo object. The function is called, which will enable the if statement, it means if the Game Object passed the ammo collider which has a tag of "Player", the function will access the Ammo's script and call the "IncreaseCurrentAmmo" function to add the ammo amount. After that, the ammo object will be destroyed.

4.5 UI Canvas

In Unity, the Canvas is a fundamental component of the unity user interface. It displays UI elements such as buttons, text, images, sliders, and other components. The UI Canvas plays as a parent of UI elements as well as a Game Object in the Hierarchy window. As any other game object, the Canvas is shown as a rectangle in Scene View to easily render, scale, or position UI elements. Every time, the Canvas is created, the Event System is created at the same time because Canvas use it for messaging the system. (Unity UI) The Canvas makes the game more interactive, therefore, this game used it for displaying the character and the enemy's health.

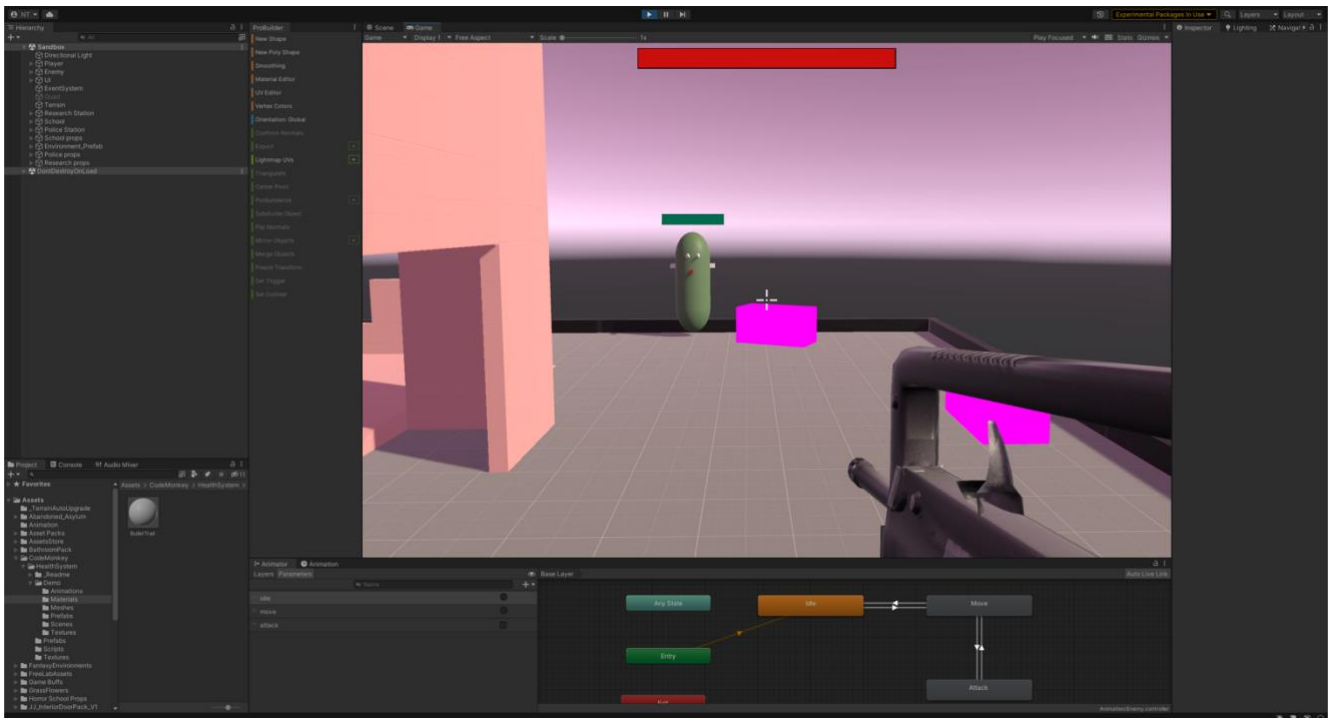


Figure 17. Player and Enemy's health.

In Figure 17, the red bar on the top of the screen displays for the player's health in the game. When the red part is disappearing, the background gradually appears, it means the that the player is losing their health because of zombie's attacking. I have used two images in the UI Canvas to make a health bar for the player. The first image is the background for the bar, the second image is a red part which is shown as a health, and I need to alter the source image to adjust the image type into the "Filled" type. As the below figure, the "Fill Amount" value will display for the second image. It means when the value is 1, the image will be shown as Figure 17, on the contrary, when the value is 0.5, the image just shows half of the bar. Then the value is 0, the image will be disappeared, and the first image will be shown, which means the player is out of health. The "Filled" type of image allows the UI image to move to the "Left" in the "Horizontal" direction at the same time as the "Fill Amount" value that drops from 1 to 0.

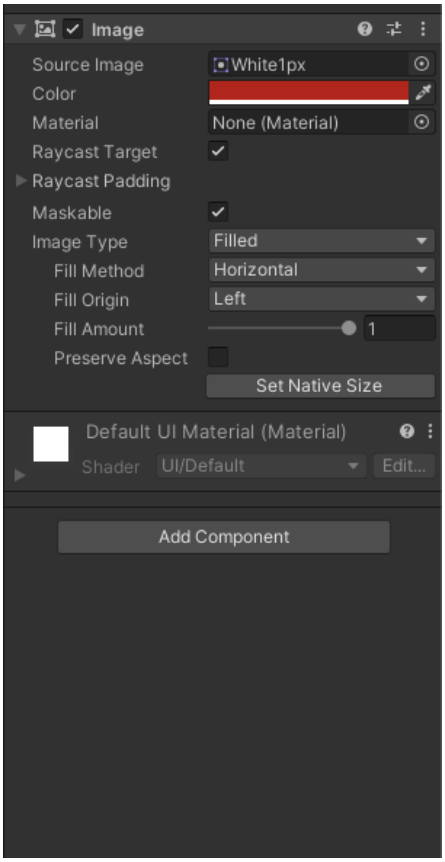


Figure 18. The UI image's properties.

To make the image run at the same time when the player is attacked, the "PlayerHealth" script is attached to the player. In Code 1 line 10, a new variable for the image was created and named "healthBar". Before that, to access the user interface elements, the script needs to use "UnityEngine.UI" in line 4. As seen in the below figure, after the variable was successfully created,

the “health” image was dragged into the “Health Bar”. To access the Fill Amount value of the “health” image, I will access the “healthBar” that I just made and access the “fillAmount” as Code 1 line 20. This function was called in the Update function because it needs to be updated every frame. Therefore, when the player is losing health, the health bar will be lost at the same time. The enemy’s health bar will be done the same as the player’s health, the only difference is the enemy’s Canvas became the children component of the Enemy so that it can move with the enemy as the Figure 10. Moreover, the amount of ammunition will use the UI Canvas to be shown in game view whenever it decreases or increases.

```
private void DisplayAmmo()
{
    int currentAmmo = ammoSlot.GetCurrentAmmo(ammoType);
    ammoText.text = currentAmmo.ToString();
}
```

Code 27. “DisplayAmmo” function from the Weapon’s script.

In this function, it retrieves the amount of ammunition for a specific ammo type from an “ammoType” parameter by using the “GetCurrentAmmo” method. It converts the amount to a string and assigns it to a text type called “ammoText” which was created in Code 14 line 17. After that, simply dragged the Text of UI to the “ammoText”. Furthermore, the UI Canvas also contains the buttons, which can help the user interact with the game. In this game, the buttons were used for the players to click on them to play, try again or exit when they lose the game. There are various scenes which were created in this game such as the start scene, game over scene, and the main scene.

Figure 19. The Start Scene.

From the beginning, when the player enters the game, the start scene will be shown as the gateway to the game's world. In other games, the start menu has a list of functionalities but in this game, this scene does not have many functions at the beginning, it has only the play button which will let the player come directly to the main scene. To make the Play button run after the player clicked on it, it should be attached with a script which will be dragged to the canvas. Therefore, the button can be executed by clicking. In the Code 28 line 19, the "PlayGame()" function was created which will be called whenever the player presses the play button. The method was made "public" so that it can be called from the button.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement;
5
6  public class SceneLoader : MonoBehaviour
7  {
8      public void GameReload()
9      {
10         SceneManager.LoadScene(0);
11         Time.timeScale = 1;
12     }
13
14     public void QuitGame()
15     {
16         Application.Quit();
17     }
18
19     public void PlayGame()
20     {
21         SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
22     }
23 }
24
```

Code 28. SceneLoader's script.

When the Play button is pressed, the scene will load to the next level thanks to the "SceneManger". To execute the "SceneManger" class, "using UnityEngine.SceneManagement" is needed to write that will allow the developer to manage and control various scenes in their game. The "SceneManager" contains different functions including the "LoadScene" which function is used to take the name of the scene, or a variable for the build index as the line 10 in the above code. In the "PlayGame()" function, the script is in the "LoadScene" function that will load to the next level in the queue of the build setting. To open the build setting, simply go to File > Build Settings. In this tab, the developer can press "Add Open Scenes" to insert the current scene, or they can easily click, hold the needed scene, and drag them to the window.

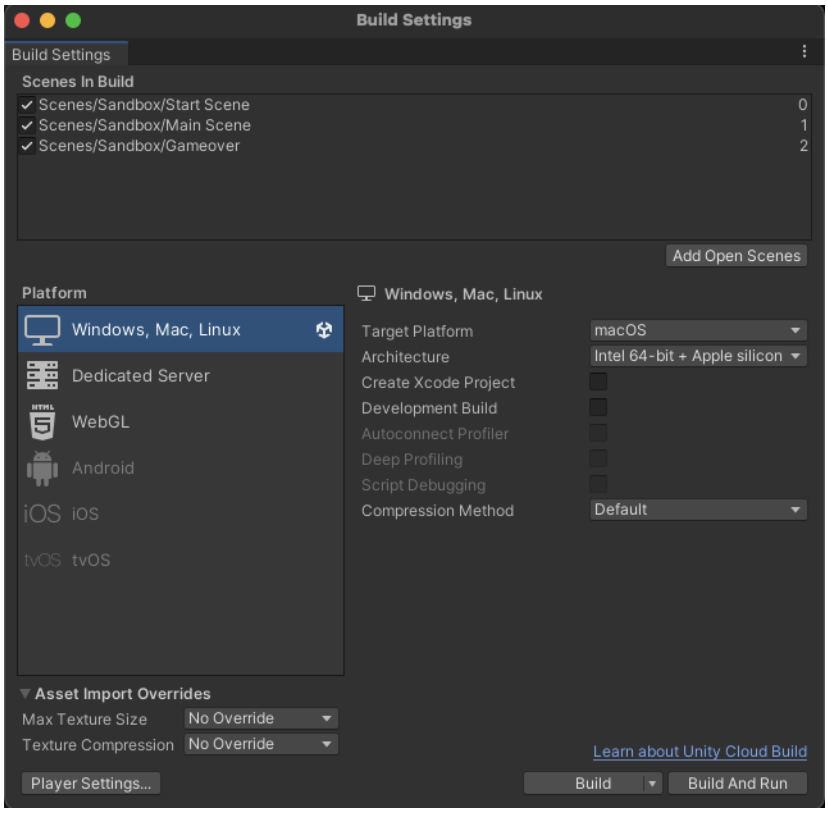


Figure 20. Build Settings window.

While in-game, if the player presses a predefined key as “Escape” key, the Canvas will be enabled and paused the game for them. This menu will allow the player freedom and flexibility while playing the game. The Pause scene will be created in the main scene so that the game is paused exactly where the player is like in the figure 21.

Figure 21. Paused game scene.

Same as the Start Menu, the Pause menu also need a script to function correctly. A script, named "SceneLoader", was attached to the canvas which contains a list of buttons of Paused menu, therefore, it can be controlled and implement functions by clicking on the buttons. After inserting the buttons to the canvas, it is required to insert a new list to the selected button in the Inspector window by simply clicking "+" on the "OnClick()" box in the Inspector window. After that, drag the Game Object that has the script attached to it. It makes all the public functions in the script can be chosen and executed.

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

public class PauseMenu : MonoBehaviour
{
    public static bool GameIsPaused = false;

    [SerializeField] GameObject pauseMenuUI;
    private void Start()
    {
        pauseMenuUI.SetActive(false);
    }
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (GameIsPaused)
            {
                Resume();
            }
            else
            {
                Pause();
            }
        }

    public void Resume()
    {
        pauseMenuUI.SetActive(false);
        Time.timeScale = 1;
        GameIsPaused = false;
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }

    void Pause()
    {
        pauseMenuUI.SetActive(true);
        Time.timeScale = 0;
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;

    }

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

```

Code 29. PauseMenu's script

The following figure shows the detail of the script attached to the Pause menu. Some different variables are created including "GameIsPaused", a Boolean value which can be turned to true or false,

a Game Object variable “pauseMenuUI”, two methods which store the methods to pause and resume the game-play, “Resume()” and “Pause()”, “QuitGame” the method will let the player out the game. In an Update function, those scripts mean that when the player presses “Escape” on the keyboard, the Boolean value will be changed. If the game is currently paused, it is equal to true then pressed the escaped or resume button by the game was already paused and so the game will be resumed. If it is not paused, the player presses the keycode while the game is running, the game will be paused. If it is true, the Game Object which was dragged to the “pauseMenuUI” will be active, and the “timeScale” will be turned to 0 to stop the game. The “timeScale” is a method of Time class, which displays the situation of the game, when the “timeScale” is set to 1, it means the game run normally, on the contrary, when it is set to 0, the game will be stopped immediately. Those functions will be set to the specified button in the Pause menu. Furthermore, while in-game, the cursor will be disappeared, it only is showed whenever the game is stopped. In Code 29 line 35, the state of cursor is locked while the game is running, it is invisible, but it will be visible and unlocked whenever the player presses “Escape” key.

Figure 22. Game Over scene.

After the player is out of health, the final scene as known as the game over scene as in Figure 22 will be displayed and show two options for the player to choose, the player can try to play the game again or they can quit the application. The button of canvas provides some features that allow user interact directly with the button. Every time, the player moves the cursor to any button, it will be showed as above figure so that the player can clearly see the chosen button. The “SceneLoader” script that is also attached with the Game Over Menu canvas, the play again button will be done as same as pause

button. The only difference is shown as in Code 30 because of the difference function, “GameLoad()” function, when the player press “Play Again” button, the scene will load to the first scene in the queue, that is the Start scene.

```
17
18      public void GameReload()
19      {
20          SceneManager.LoadScene(0);
21          Time.timeScale = 1;
22      }
23
24      public void QuitGame()
25      {
26          Application.Quit();
27      }
```

Code 30. SceneLoader’s script.

5 BUILDING THE GAME

In this final section, it will be explained how to build the game after the game is finished and ready to play. When the game is completed, the developer can build a game for the testing purpose as the real game in different application. In Unity, it is simple to build a game, choose the target platform and click the “Build and Run” button or “Build” button in the Build Setting window. Before building the game, all necessary details need to be checked carefully and the important scenes should be inserted in the Scenes In Build. As shown in Figure 59 below, there are different options to choose the platform which are offered by Unity such as MacOS, Windows, Linux, Android, or WebGL. After pressing the button, the game is built, and it will run automatically if the “Build And Run” option is selected. After that, the game is ready to share with others for them to try the new game, that is “The Last Run”.

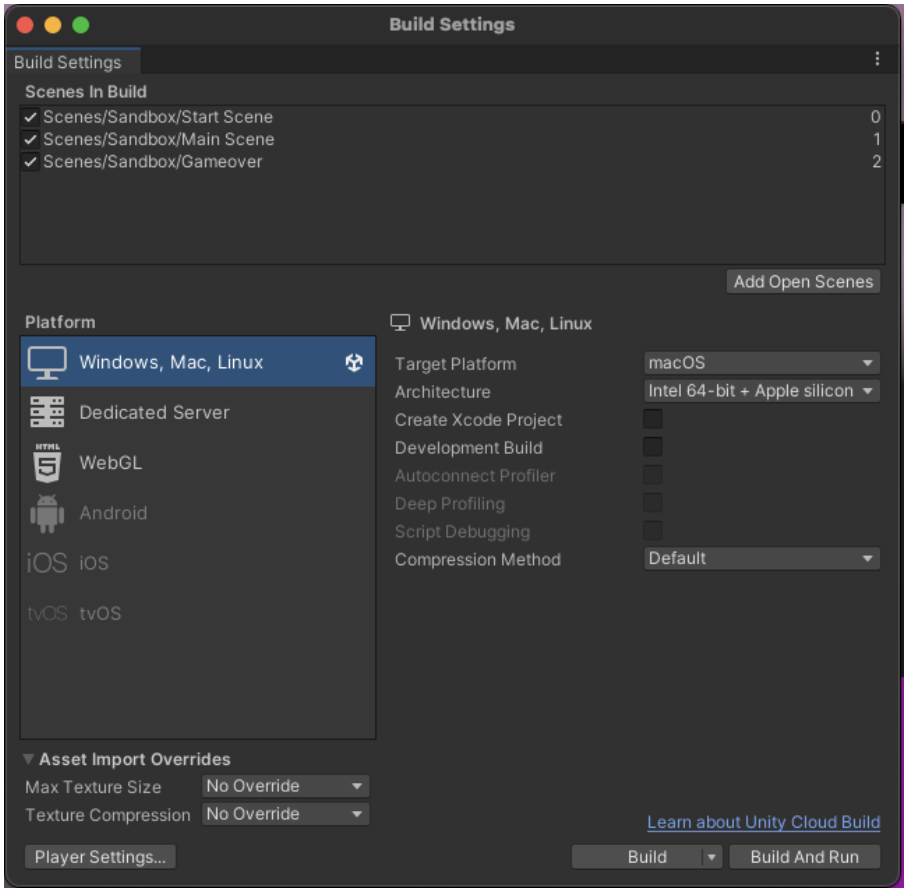


Figure 23. Build Settings window.

6 CONCLUSIONS

This thesis went through and explained step by step how to make a 3D game and how to develop it with the Unity game engine. Developing a 3D game is a hard and lengthy process but Unity assists developers with scripting, and developing interfaces, 3D game development has become easier than expected. Working on this game, took the most time to finish but it helped me get more knowledge and understanding about game development and programming in Unity. This project skipped some parts such as sound designs, and lights. It will be developed further and more improvement about the features, images, the scene, and sound designs in the future.

Overall, this thesis has shown a way to make a 3D game step by step. Developing a 3D game is an interesting and good way to learn programming, and it will explore how the developer create new things, their logical thinking, and imagination. After completing this thesis, the question about which game engine the developers should use to develop a game has answers. With significant changes and improvement, Unity has proved that they are the best choice for every developer because it has a simple user interface and understanding, and clear documentation for the developers.

7 REFERENCES

Adam Sinicki, 2021. *What is Unity? Everything you need to know*. Available at: https://www.androidauthority.com/what-is-unity-1131558/ Accessed 03 April 2023.

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 03 April 2023.

DocFX, 2023. *Character movement*. Available at: https://docs.unity3d.com/Packages/com.unity.charactercontroller@1.0/manual/character-movement.html Accessed 03 April 2023.

John French, 2022. *Interfaces in Unity (how and when to use them)*. Available at: https://gamedevbeginner.com/interfaces-in-unity/ Accessed 03 April 2023.

J.Clement, 2022. *Video game industry – Statistics & Facts*. Available at: https://www.statista.com/topics/868/video-games/#topicOverview Accessed 16 March 2023.

Helsinki Times, 2022. *What make the Finnish gaming market explode in the past few years*. Available at: https://www.helsinkitimes.fi/business/21594-what-made-the-finnish-gaming-market-explode-in-the-past-few-years.html Accessed 16 March 2023.

Matther Byrd. 2021. *25 Best First-Person Shooter Games Ever Made*. Available at: https://www.denofgeek.com/games/best-fps-games-ever-console-pc/ Access 24 April 2023.

Unity, 2023. *Unity 2022.2.13*. Available at: https://unity.com/releases/editor/whats-new/2022.2.13#release-notes Accessed 03 April 2023.

Unity, 2015. *UNITY 5 IS HERE*. Available at: https://unity.com/our-company/newsroom/unity-5-here Accessed 03 April 2023.

Unity, 2017. *Unity 2017.1 Now Available*. Available at: https://unity.com/our-company/newsroom/unity-2017-1-now-available Accessed 03 April 2023.

Unity Documentation, 2023a.2, *Toolbar*. Available at: https://docs.unity3d.com/2023.2/Documentation/Manual/Toolbar.html Accessed 05 April 2023.

Unity Documentation, 2023b .2, *Hierarchy*. Available at: https://docs.unity3d.com/2023.2/Documentation/Manual/Hierarchy.html Accessed 05 April 2023.

Unity Documentation, 2023c.2, *The Game view*. Available at: https://docs.unity3d.com/2023.2/Documentation/Manual/GameView.html Accessed 05 April 2023.

Unity Documentationd, 2023.2, *The Scene view*. Available at: https://docs.unity3d.com/2023.2/Documentation/Manual/UsingTheSceneView.html Accessed 05 April 2023.

Unity Documentation, 2023e.2, *The Inspector window*. Available at: https://docs.unity3d.com/2023.2/Documentation/Manual/UsingTheInspector.html Accessed 05 April 2023.

Unity Documentation, 2023f.2, *The Project window*. Available at: https://docs.unity3d.com/2023.2/Documentation/Manual/ProjectView.html Accessed 05 April 2023.

Unity Documentation, 2023g.2. *The status bar*. Available at: https://docs.unity3d.com/2023.2/Documentation/Manual/StatusBar.html Accessed 05 April 2023.

Unity Documentation, 2023h.2. *Audio*. Available at: https://docs.unity3d.com/2023.2/Documentation/Manual/Audio.html Accessed 05 April 2023.

Unity UI 1.0.0. *Canvas*. Available at: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html Accessed 17 May 2023.

Unity Documentation, 2021a. *Navigation System in Unity*. Available at: https://docs.unity3d.com/2023.2/Documentation/Manual/nav-NavigationSystem.html Accessed 23 May 2023.

Unity Documentation, 2021b. *Physics.Raycast*. Available at: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html Accessed 23 May 2023.

APPENDIX 1/1

Game Scripts

PlayerHealth.cs

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

public class PlayerHealth : MonoBehaviour
{
    public float health = 100f;
    public float maxHealth;
    public Image healthBar;

    void Start()
    {
        maxHealth = health;
    }

    void Update()
    {
        healthBar.fillAmount = Mathf.Clamp(health / maxHealth, 0, 1);
    }

    public void TakeDamage(float damage)
    {
        health -= damage;
        if (health <= 0)
        {
            SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
            Cursor.lockState = CursorLockMode.None;
            Cursor.visible = true;
        }

    }
}
```

APPENDIX 1/2

SceneLoader.cs

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

public class SceneLoader : MonoBehaviour
{
    public static bool GameIsPaused = false;

    [SerializeField] GameObject pauseMenuUI;
    public void PlayGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
    }

    public void GameReload()
    {
        SceneManager.LoadScene(0);
        Time.timeScale = 1;
    }

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

    private void Start()
    {
        pauseMenuUI.SetActive(false);
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (GameIsPaused)
            {
                Resume();
            }
            else
            {
                Pause();
            }
        }
    }

    public void Resume()
    {
        pauseMenuUI.SetActive(false);
        Time.timeScale = 1;
        GameIsPaused = false;
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }

    void Pause()
    {
        pauseMenuUI.SetActive(true);
        Time.timeScale = 0;
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;

    }
}
```

APPENDIX 1/3

EnemyAI.cs

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

public class EnemyAI : MonoBehaviour
{
    [SerializeField] Transform target;
    [SerializeField] float chaseRange = 5f;
    [SerializeField] float turnSpeed = 5f;
    [SerializeField] float damage = 40f;

    PlayerHealth objective;
    EnemyHealth health;
    NavMeshAgent navMeshAgent;
    float distanceToTarget = Mathf.Infinity;
    bool isProvoked = false;

    void Start()
    {
        navMeshAgent = GetComponent<NavMeshAgent>();
        objective = FindObjectOfType<PlayerHealth>();
        health = GetComponent<EnemyHealth>();
    }


    void Update()
    {
        distanceToTarget = Vector3.Distance(target.position, transform.position);

        if (health.IsDead())
        {
            enabled = false;
            navMeshAgent.enabled = false;
        }

        if(isProvoked)
        {
            EngageTarget();
        }
        else if(distanceToTarget <= chaseRange)
        {
            isProvoked = true;
        }
    }

    private void EngageTarget()
    {
        FaceTarget();
        if(distanceToTarget >= navMeshAgent.stoppingDistance)
        {
            ChaseTarget();
        }
        else if(distanceToTarget <= navMeshAgent.stoppingDistance)
        {
            AttackTarget();
        }
    }
```

APPENDIX 1/4

```
private void ChaseTarget()
{
    GetComponent<Animator>().SetBool("attack", false);
    GetComponent<Animator>().SetTrigger("move");
    navMeshAgent.SetDestination(target.position);
}

public void OnDamageTaken()
{
    isProvoked = true;
}

private void AttackTarget()
{
    GetComponent<Animator>().SetBool("attack", true);
}

private void FaceTarget()
{
    Vector3 direction = (target.position - transform.position).normalized;
    Quaternion lookRotation = Quaternion.LookRotation(new Vector3(direction.x, 0, direction.z));
    transform.rotation = Quaternion.Slerp(transform.rotation, lookRotation, Time.deltaTime * turnSpeed);
}

void OnDrawGizmosSelected()
{
    // Display the explosion radius when selected
    Gizmos.color = new Color(1,0,0,1);
    Gizmos.DrawWireSphere(transform.position, chaseRange);
}

public void AttackHitEvent()
{
    if (objective == null) return;
    objective.TakeDamage(damage);
    Debug.Log("bang bang");
}
```

APPENDIX 1/5

EnemyHealth.cs

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

public class EnemyHealth : MonoBehaviour
{
    public float hitPoints = 100f;
    public float maxPoint;
    public Image enemyHealthBar;

    bool isDead = false;

    public bool IsDead()
    {
        return isDead;
    }

    void Start()
    {
        maxPoint = hitPoints;
    }

    void Update()
    {
        enemyHealthBar.fillAmount = Mathf.Clamp(hitPoints / maxPoint, 0, 1);
    }

    public void TakeDamage(float damage)
    {
        BroadcastMessage("OnDamageTaken");
        hitPoints -= damage;
        if (hitPoints <= 0)
        {
            Dead();
        }
    }

    private void Dead()
    {
        if (isDead) return;
        isDead = true;
        GetComponent<Animator>().SetTrigger("dead");
    }
}
```

APPENDIX 1/6

Weapon.cs

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

public class Weapon : MonoBehaviour
{
    [SerializeField] Camera FPCamera;
    [SerializeField] float range = 100f;
    [SerializeField] float damage = 30f;
    [SerializeField] ParticleSystem muzzleFlash;
    [SerializeField] GameObject hitEffect;
    [SerializeField] Ammo ammoSlot;
    [SerializeField] AmmoType ammoType;
    [SerializeField] float timeBetweenShots = 0.5f;
    [SerializeField] Text ammoText;

    bool canShoot = true;

    private void OnEnable()
    {
        canShoot = true;
    }

    void Update()
    {
        DisplayAmmo();
        if (Input.GetMouseButtonDown(0) && canShoot == true)
        {
           StartCoroutine(Shoot());
        }
    }

    private void DisplayAmmo()
    {
        int currentAmmo = ammoSlot.GetCurrentAmmo(ammoType);
        ammoText.text = currentAmmo.ToString();
    }

    IEnumerator Shoot()
    {
        canShoot = false;
        if (ammoSlot.GetCurrentAmmo(ammoType) > 0)
        {
            PlayMuzzleFlash();
            ProcessRaycast();
            ammoSlot.ReduceCurrentAmmo(ammoType);
        }
        yield return new WaitForSeconds(timeBetweenShots);
        canShoot = true;
    }

    private void PlayMuzzleFlash()
    {
        muzzleFlash.Play();
    }

    private void ProcessRaycast()
    {
        RaycastHit hit;
        if (Physics.Raycast(FPCamera.transform.position, FPCamera.transform.forward, out hit, range))
        {
            CreateHitImpact(hit);
            EnemyHealth target = hit.transform.GetComponent<EnemyHealth>();
            if (target == null) return;
            target.TakeDamage(damage);
        }
        else
        {
            return;
        }
    }
```

APPENDIX 1/7

WeaponSwitcher.cs

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

public class WeaponSwitcher : MonoBehaviour
{
    [SerializeField] int currentWeapon = 0;

    void Start()
    {
        SetWeaponActive();
    }

    void Update()
    {
        int previousWeapon = currentWeapon;

        ProcessKeyInput();

        if (previousWeapon != currentWeapon)
        {
            SetWeaponActive();
        }
    }



    private void ProcessKeyInput()
    {
        if (Input.GetKeyDown(KeyCode.Alpha1))
        {
            currentWeapon = 0;
        }
        if (Input.GetKeyDown(KeyCode.Alpha2))
        {
            currentWeapon = 1;
        }
        if (Input.GetKeyDown(KeyCode.Alpha3))
        {
            currentWeapon = 2;
        }
    }

    private void SetWeaponActive()
    {
        int weaponIndex = 0;

        foreach (Transform weapon in transform)
        {
            if (weaponIndex == currentWeapon)
            {
                weapon.gameObject.SetActive(true);
            }
            else
            {
                weapon.gameObject.SetActive(false);
            }
            weaponIndex++;
        }
    }
```

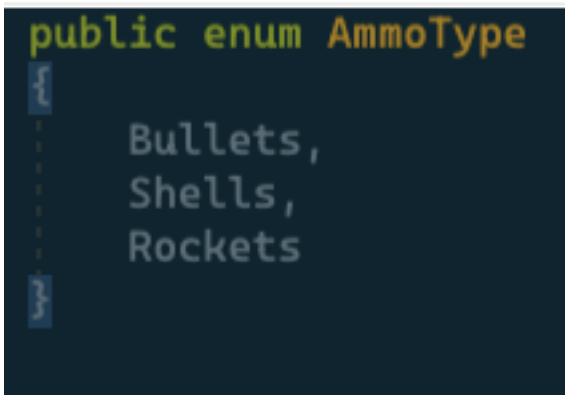
APPENDIX 1/8

Ammo.cs

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

public class Ammo : MonoBehaviour
{
    [SerializeField] AmmoSlot[] ammoSlots;

    [System.Serializable]
    private class AmmoSlot
    {
        public AmmoType ammoType;
        public int ammoAmount;
    }

    public int GetCurrentAmmo(AmmoType ammoType)
    {
        return GetAmmoSlot(ammoType).ammoAmount;
    }

    public void ReduceCurrentAmmo(AmmoType ammoType)
    {
        GetAmmoSlot(ammoType).ammoAmount--;
    }

    public void IncreaseCurrentAmmo(AmmoType ammoType, int ammoAmount)
    {
        GetAmmoSlot(ammoType).ammoAmount += ammoAmount;
    }

    private AmmoSlot GetAmmoSlot(AmmoType ammoType)
    {
        foreach (AmmoSlot slot in ammoSlots)
        {
            if (slot.ammoType == ammoType)
            {
                return slot;
            }
        }
        return null;
    }
}
```

APPENDIX 1/9

AmmoType.cs



APPENDIX 1/10

AmmoPickup.cs

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

public class AmmoPickup : MonoBehaviour
{
    [SerializeField] int ammoAmount = 5;
    [SerializeField] AmmoType ammoType;

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            FindObjectOfType<Ammo>().IncreaseCurrentAmmo(ammoType, ammoAmount);
            Destroy(gameObject);
        }
    }
}
```