The Development of a 3D Game

John Harrow
ID: 190012718
AC40001 Honours Project
BSc (Hons) Applied Computing
University of Dundee, 2023
Supervisor: Dr. Iain Martin

Abstract **-** *The project was to develop a 3D first-person shooter survival game using Unity and C#. The project was done using an Agile methodology. The game consists of a player killing waves of enemies which spawn in an environment while trying to survive as long as possible before they inevitably die. There were two phases of testing which involved testers playing the game and answering a questionnaire on their experience. The feedback from testers was helpful in finding issues and giving ideas for improvements in the game. Overall, the project successfully completed by creating a game that met all of the initial requirements and specification while adding on additional features that were asked for my testers.*

1 Introduction

This project was to create a 3D game. The main aim was to use all the experience of software development gained in the previous years of university to use a game engine to create a game that is challenging yet enjoyable to play. The key objective was to have developed a game that contains satisfying player movement and combat, increasingly difficult enemies to fight and an environment that immerses the player in the game.

1.1 Genres of Games

The first decision that had to be made for the project is what kind of game should be developed. There are many different popular genres of games such as strategy games, sandbox games and first-person shooter (FPS) games. The one that was chosen was an FPS game [1]. The reason that a FPS game was chosen was because it is currently a very popular genre of game that has a lot of fans [2], and that the developer has a lot of enjoyable experiences playing them so has knowledge of which features make an FPS game fun.

1.2 Game Engines

The next decision that had to be made was how the game was going to be developed. There were two options that were considered when choosing how the game was going to be developed and they were; using a game engine to develop the game or creating a game engine from scratch and using that to develop the game. If creating a game engine from scratch was chosen, then a large proportion of the work that was done in the project would've been focused on creating the engine rather than on developing an enjoyable playing experience [3]. It was decided that a game engine would be used so that all the useful features they provide could be used so that the features and feel of the game could be prioritized.

1.3 Methodology

The final initial decision that had to be made was how development and the workload would be structured. There are multiple effective ways that developers can approach a project but the main two that were researched were Agile [4] and Waterfall [5].

Waterfall is a methodology that provides a concrete and linear plan to follow that does not get regularly altered or changed. In waterfall, each stage of development is clearly defined and must be completed before moving onto the next stage. The main advantage of Waterfall is that all the requirements are firmly established early in the project and are not constantly in flux. A disadvantage is that issues may not be discovered until after a phase of development has been passed so it makes it more difficult to go back to find the error and make the changes required.

Agile is a methodology that is far more flexible than Waterfall and allows the phases of development to be split into small development cycles. An advantage of Agile is that it allows a lot more flexibility to change the direction of the project in response to new ideas or insights compared to Waterfall. A disadvantage is that the requirements are not as clearly defined when using Waterfall and it is easy to get caught up in less important features after getting feedback.

It was decided that Agile would be used as the idea for the game at the initial stage was not set in stone so agile would allow the direction of development to pivot and make changes to plans based on new ideas or feedback less of an issue.

2 Background

2.1 Researching Genres

When first thinking about what kind of game to make it was useful to research into the different genres and types of games that have already been made to give inspiration and ideas for what kind of game to make. There are many different genres of games but the ones that were found interesting and researched were Tower defense strategy games, racing games, sandbox games and FPS.

The first genre of game that was looked at was tower defense games [6]. A tower defense game consists of the player having a top-down view of a map that contains a track that enemies will slowly follow until they reach the end of the track at which point the player will take damage. The player can use points or currency to buy things like weapons and characters that will sit next to the track and attack the enemies that are following it. These types of games usually involve waves of enemies which increase in number, speed and health, which forces the player to use the currency they gain wisely to keep up with the enemy waves. A few games that were looked at and played for inspiration were Bloons Tower Defense [7] and Kingdom Rush [8].

The next genre that was researched was racing games. Most racing games are quite simple as they just consist of the player controlling a car while racing against other players or AI on a track. These games can be fun due to their competitive nature and some more advanced games have integration with physical steering wheels and pedals which can be used to control the vehicle as you could in a real car. Mario Cart [9] and Need for Speed [10] were the two main games that were looked at when deciding if making a racing game was the right choice.

Another genre of game that was researched was sandbox games. Sandbox games put players into a world that allows them to creatively interact with the environment that they are in. This genre of game usually does not contain a set end goal and it is up to the player what they want to do with what is provided to them which usually involve building structures or finding resources. A few popular games in this genre are Minecraft [11] and Animal Crossing [12].

The final genre of game that was researched, and the one that was chosen, was FPS. FPS games are usually based around gun and weapon combat in a first-person perspective. The player experiences all the action through the eyes of the character they are controlling. Most FPS games involve shooting at enemies which can either be controlled by other players or by the game. There are a few different types of FPS games that were considered.

There was the kind which emphasises a story which is viewed by the player through their character and has combat to give the player a more engaged feeling in the story, an example of this kind of game being The Last of Us [13].

Another type of FPS is the competitive kind which usually involve two teams of players fighting against each other while trying to fulfil objectives. These games can be fun as it allows players to test their skill against other players. Examples of these types of games being Valorant [14] and Counter-Strike [15].

The last type of FPS game that was looked at was survival FPS games. These consist of the player being put in an environment where their main objective is to survive as long as possible while enemies try to kill them. These can be structured by using waves of AI enemies to attack the player or to have the player fight against other players in an ever-shrinking environment until only one player is left alive. Examples of this type of game are Call of Duty: Zombies [16], Doom Eternal [17] and Warhammer40k: Darktide [18].

Researching all three of these types of games was useful as it allowed good aspects of each type to be taken and used in the game that was to be developed. The main features and structure of the game was mainly inspired by survival FPS games but gameplay features from the other two types of game were mixed in.

2.2 Researching Game Engines

A game engine [19] is a software framework that is designed to facilitate the development of games and contains a variety of useful libraries. Some of these libraries are a rendering engine for graphics, a physics engine and collision detection, sound and animation. The advantage of having all these libraries is that it saves the developer time as rather than creating their own complex engines for each aspect of the game, there are already pre-built ones that can be altered to fit the game they are developing. Having these libraries are a real advantage but there is still a lot of work to be done by developers to fit all the pieces they want together in their game.

There are a few popular game engines that had to be considered when choosing what engine would work best with the kind of game that was to be developed. The ones that were researched were Unity, Unreal Engine 4, Godot and GameMaker.

The first game engine that was looked at was Unreal Engine 4 [20]. Unreal Engine is currently one of the most popular game engines and in 2018 had more than 6.3 million users [21]. It has been used to make some very successful games such as the Gears of War series. The

main advantage of using Unreal Engine is that it can facilitate good graphics which allows fantastic looking games to be made with it. It also has support for games that are multiplayer. The only disadvantage is that it uses C++ as its scripting language which could be argued to be more difficult than C# which is what Unity uses.

The next game engine that was considered was Godot. Godot is a game engine that is slightly more focused and more supported towards 2D rather than 3D games. It allows the use of either its own GDScript which has great integration with the engine or C++ and C#. Using Godot may have had the added factor of learning to use the new script to get the best use of the engine, but the developer had already used Godot to make a game in the past. There haven't been many popular games made with Godot and most of the games that are made with it are 2D pixel art games which was not the style of game that was intended to be made.

Another game engine that was briefly researched was GameMaker. This engine was found to be interesting as it allows developers to create games with little programming knowledge required. It has been used to make some popular games like Hyper Light Drifter. However, this game engine was ruled out as it does not have very good support for making 3D games as it is mostly geared towards 2D games [22].

The final game engine that was considered is Unity. Unity is a popular game engine which can be used to create 3D games. Unity offers a scripting API in C# using Mono and supports multiple different platforms including PC, Console and Mobile. Unity has a free license for games that generate under $200,000 annually so is very popular with smaller companies and indie developers [23]. Unity has a large asset store which allows developers to buy and sell useful assets like 3D models and animations [24]. Some popular games that were made in Unity are Hollow Knight, Genshin Impact, Cuphead and Rust.

The game engine that was chosen to be used was Unity. Unity was chosen due to it allowing scripts to be made in C# which the developer already has experience in so it would not be as much of a learning curve to use. It also has a wide variety of tutorials and documentation, with a large quantity being focused on building FPS games, due to it being popular with indie developers which allows easier learning and less chance of getting stuck when running into an issue. Another advantage is that Unity's asset store contains an incredible amount of high-quality free assets to use in the project some of which include 3D models of things like characters, guns and buildings, and a wide variety of different sounds. Unity and Unreal Engine 4 were both found to have been a good pick for creating the game but the fact that Unity uses C# rather than C++, which is what Unreal Engine 4 uses, weighed the scales more in Unity's favour and when looking for tutorials and documentation for learning how to use the engine and making an FPS in particular, Unity had far more and better-quality resources.

2.3 Similar Games

There are a few FPS games that inspired the idea for the game that is being developed in this project. The first and most influential on this project is Call of Duty: Zombies [16]. This game mode, which is featured in many of the Call of Duty games consists of the player being spawned in a map and they must survive waves of zombies. What makes this game enjoyable is not only the challenge of surviving the ever-increasing difficulty of the waves of zombies but the satisfying gunplay and movement the player can achieve. The main objective of the game is to survive as many waves of zombies as possible before they inevitably are killed. The game features a variety of different guns and melee weapons that can be bought using points gained by killing zombies which keeps the game interesting as each gun has its own feel and advantages.

Another game that influenced the game that was to be developed is Valorant [14]. Valorant is a far more competitive game compared to Call of Duty: Zombies. It consists of two teams of players, one being attackers and one being defenders. Each player selects an agent at the start of the game and each agent has a unique set of abilities which adds more layers of depth to the gameplay. The goal of the attacking team is to either kill all the enemies or to plant a bomb and survive until it explodes while stopping the defenders from defusing the bomb, while the defending team does the same, but they are trying to stop the attackers from planting the bomb. What makes this game fun is that there is a lot of skill expression as if you have good aim with your weapons and have a lot of game knowledge you can use it to your advantage to defeat the enemy team.

Doom Eternal [17] is another game that had an impact on what features the game would have. Doom Eternal is a first-person shooter that revolves around the player going through different levels of the game while being actively encouraged to kill lots of demons and complete objectives. What makes this game stand out is how gruesome and brutal combat can be, which a lot of players can find satisfying and makes the game more enjoyable. The game allows the player to use multiple different weapons such as a shotgun, rocket launcher and plasma rifle. Compared to Call of Duty: Zombies and Valorant this game is more linear, and story driven as the player progresses through different which contain a variety of different enemies to fight and obstacles to overcome compared to the other two games which are more repetitive, and skill focused.

The final game that inspired the idea for the game was Halo Reach: Infection [25]. Infection was a multiplayer game mode within Halo Reach that consisted of a group of players being put on a large map and all, but one player would be on one team and the one player would be on the other team and be called infected. The goal of the players who were not infected was to use their ranged weapons to kill any players that were infected and to survive the longest. The infected players only had a melee weapon and when they killed a non-infected player, that player would become infected and join them in hunting down the players. The winner of the game mode would be the last player to not be infected. What made this game mode interesting was that as more of the players were infected, the more difficult it would be to survive.

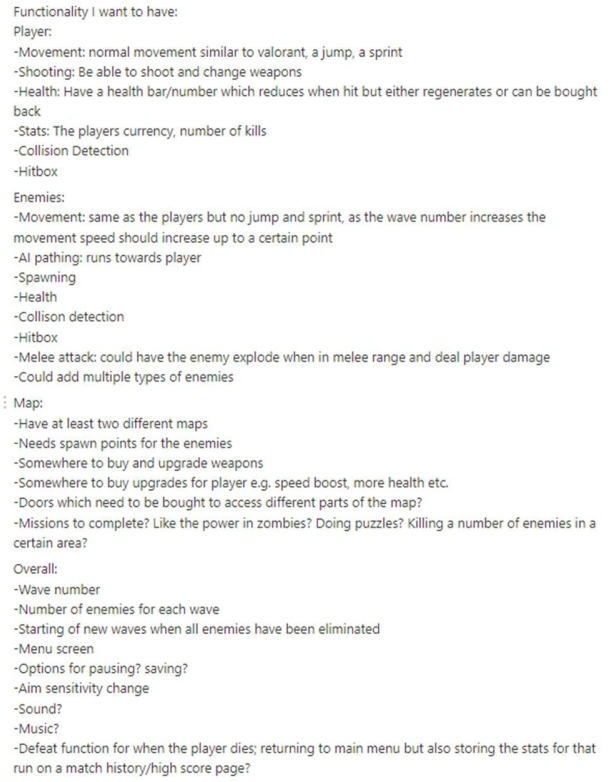
There are many features in the game that was developed that were inspired or replicated from the games just mentioned. The main premise of the game was heavily inspired by Call of Duty: Zombies as the main goal of the game is to survive waves of enemies that get increasingly difficult and Halo Reach: Infection also influenced this aspect as the concept of the game getting progressively more difficult until the inevitable death of the player was focused on getting right during development and testing. Valorant influenced how the guns were to function as in Valorant the player can express their skill fully as there is very little randomization to how the bullets will leave the gun which was an aspect that was thought to be a good addition to the game. Doom Eternal's non-stop action influenced the pace of the game that was developed as it was desired that the player never be idle and get bored.

3 Specification

At the beginning of the project, it was thought to be a good idea to set out roughly how the game would play, what functionality would be required and the priority of these features. It was also important to decide on an overall plan and work schedule for the project which was done using a Gantt chart.

3.1 Requirements

Once the initial idea for the game was set, it was time to decide which specific features would be included in the game and how it would be played. The initial game was first broken down into more general features that were desired to be in the game and were listed out. These features were split into the different categories of player, enemies, map and overall. Splitting the requirements up like this was useful as it allowed for the grouping of items to be viewed early on. Having them split up was also useful as some of the categories would both need the same functionality, so it was a reminder to make sure both have them as backlog items. These initial requirements can be seen in *Figure 1*.


Functionality I want to have:
Player:
-Movement: normal movement similar to valorant, a jump, a sprint
-Shooting: Be able to shoot and change weapons
-Health: Have a health bar/number which reduces when hit but either regenerates or can be bought back
-Stats: The players currency, number of kills
-Collision Detection
-Hitbox

Enemies:
-Movement: same as the players but no jump and sprint, as the wave number increases the movement speed should increase up to a certain point
-AI pathing: runs towards player
-Spawning
-Health
-Collison detection
-Hitbox
-Melee attack: could have the enemy explode when in melee range and deal player damage
-Could add multiple types of enemies

Map:
-Have at least two different maps
-Needs spawn points for the enemies
-Somewhere to buy and upgrade weapons
-Somewhere to buy upgrades for player e.g. speed boost, more health etc.
-Doors which need to be bought to access different parts of the map?
-Missions to complete? Like the power in zombies? Doing puzzles? Killing a number of enemies in a certain area?

Overall:
-Wave number
-Number of enemies for each wave
-Starting of new waves when all enemies have been eliminated
-Menu screen
-Options for pausing? saving?
-Aim sensitivity change
-Sound?
-Music?
-Defeat function for when the player dies; returning to main menu but also storing the stats for that run on a match history/high score page?

Figure 1: General Requirements

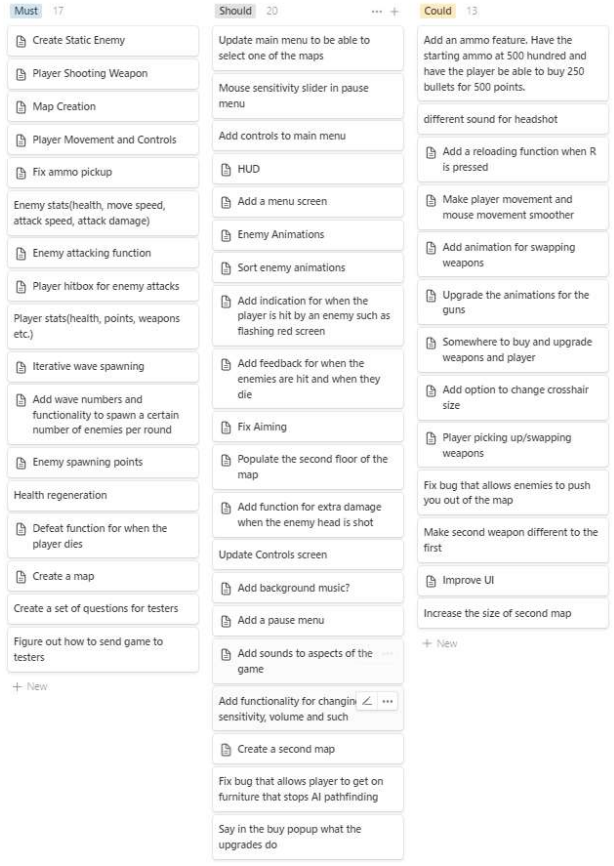


Figure 2: Backlog Items

Having the list of general features written out was helpful as it allowed them to then be broken down further into specific backlog items to be done which can be seen in *Figure 2*. These backlog items were then analysed and sorted into categories based on using MoSCoW analysis [26]. This type of analysis consists of splitting the items into the categories of must, should, could and wont. Having the backlog items categorised in this way allowed for easier prioritisation and selection of which item to be in each sprint.

3.2 Sprints

As the project was planned to be done using agile methodology, the backlog items were to be split into small sprints which were decided to last a week. The backlog items that were decided to be done for each sprint was based on the MoSCoW analysis and therefore the items that were deemed to be in the must category were prioritised in the first few sprints. Getting the musts done early allowed for an early playable prototype to be made and tested by players. An example of a sprint is shown in *Figure 3*.

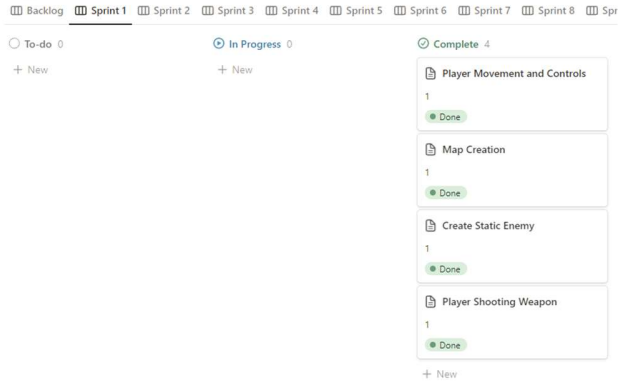


Figure 3: Sprint 1

3.3 Deliverables

The project was split into the different overall tasks that would need to be completed during the project. A Gantt chart was created to give an estimated timeline of when each of the tasks should be done by. This Gantt chart can be seen in *Figure 4*. (Full gantt chart can be seen in appendix [8])

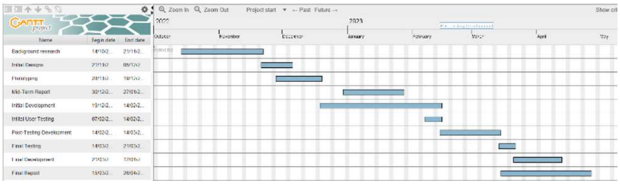


Figure 4: Initial Gantt chart

The gantt chart shown in *Figure 4* has been split into rows which each contain a deliverable and each deliverable is given a start and end date which then allows the duration to be shown on the chart. The deliverables that were decided on were background research, initial designs, prototyping, mid-term report, initial development, initial user testing, post-testing development, final testing, final development and final report. Each deliverable is shown on the chart in chronological order which made it easier to know which one to be focusing on throughout the project.

3.4 Meetings

Meetings with the advisor of studies were had every week which was useful as they were able to answer any questions that were had or clear up any misconceptions that were present at the beginning of the project. Useful feedback was also given by the advisor when ideas for the project were proposed and features in the game displayed. The minutes for schedule and minutes for the meetings can be found in the appendix [3].

4 Design

4.1 Map

The fist aspect of design that had to be done in the project was designing a map for the game to be played in. This was quite an important aspect as the quality of the map that was to be created would greatly influence the difficulty and enjoyment of the game. The initial idea for the map was to have the player spawn in a room and there be multiple connected rooms which would contain areas that the enemies would spawn from. A rough design was drawn up for how the map was to be created which is shown in *Figure 5*.

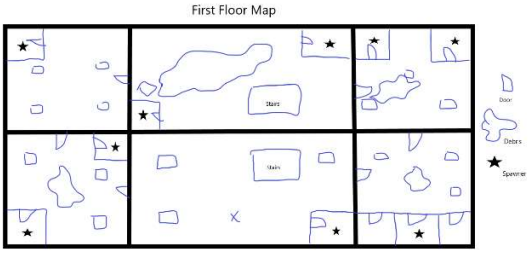


Figure 5: First floor map design

The map was greatly influenced during the background research phase as a great free environment asset was found on the unity store which fit the intended aesthetic well and fit the needs of what environmental aspects were required such as stairs, small rooms and tables to use as obstacles. The asset that was found was one of a dungeon which fit well with the enemies being intended to be zombies and contained prison cells which were the perfect spot to place the spawners for the enemies which is why it can be seen in *Figure 5* that there are lots of small room containing stars which represent where the spawners are to be places.

It was decided that it was important to have bits of debris in some of the rooms that would force the player into either a chokepoint where they couldn't just run around the enemies and to force them to get close to the entrance of the cells that the enemies were spawning from. The placement of the stairs and having a second floor was also another addition to try to increase the difficulty by creating addition choke points for the player to get caught in by enemies.

4.2 Design Patterns

Design patterns can be useful when developing a game. There are a wide variety of different patterns that can be used but the only two that were applicable for this project are the State Pattern and the component pattern.

The State Pattern [27] is a design pattern that lets objects change how they behave depending on their current state. Each state will have its own set of behaviours such as what functions to run and how to respond to an input. This pattern was chosen to be used as it was deemed to be required to create the AI for the enemies. The initial idea was for the enemy to have two fixed states which were to chase the player and to attack the player. Using the state pattern facilitated this as it was possible to create the two distinct states and have them swap from one to the other when the conditions were met for it to do so. Another place that the State pattern was used was when spawning the waves of enemies. It was required for the script that handles the spawning of enemies to have three different states which are shown in *Figure 6*.

```
public enum SpawnState { SPAWNING, WAITING, COUNTING };
```

Figure 6: Spawn State

The other design pattern that was seen fit to be used was the Component Pattern [28]. In Unity, when you want to create a new aspect of the game it usually involves creating an object and attaching different components to it. What the component pattern allows is for these components of the object to be independent and decoupled which allows for each component to be modified without having to worry about messing with any of the other components. A few examples of components that would be added to objects when building a game in Unity are scripts, meshes, colliders, audio sources and animators.

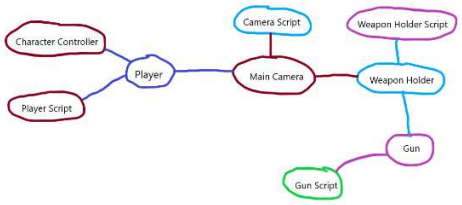


Figure 6.5: Player Object

Figure 6.5 shows an example of an object in the game and shows the relationship between components that are contained in the player object in the game that was developed. *Figure 7* then shows how the relationships are made in Unity.

Figure 7: Player Object in Unity

Another advantage of using this pattern is that it makes it easier to refer to components of objects. In Unity, you can create a variable that is an object reference and then use the GetComponent function that is built into unity to refer to a specific component of that object and interact with it as shown in *Figure 8*.

```
var color = gotHitScreen.GetComponent<Image>().color;
```

Figure 8: Line of code showing get component function.

5 Implementation and Testing

5.1 Sprint 1 and 2

The main goal of these two sprints was to get to grips with using Unity, coding in C# and creating some of the fundamental aspects of the game.

The first thing that was chosen to be created, and arguably the most important for an FPS, was the player movement and aiming. In Unity, it is necessary to create a Scene as it is the space that contains all the game objects that are created. The first task that was required was creating a player object so that all the necessary components could be added to it. It was necessary to add a Character Controller component which so that the current position of the player can be referenced in code and be moved. A Script called PlayerMovement was then added which got the input from key presses and moved the player accordingly on the x and z plane. The useful thing about Character Controllers is that it allows the built in move function to be used as seen in *Figure 9* to move the character.

```
controller.Move(move * speed * Time.deltaTime);

velocity.y += gravity * Time.deltaTime;

controller.Move(velocity * Time.deltaTime);
```

Figure 9: Movement Code

The next thing that needed to be done after creating the movement was to add the camera. Unity has a built in Camera component that was added to the player object which allows for things like field of view and depth to be manually changed. The script FirstPersonLook was created to control the camera based on mouse movement. Unity has a built-in function for getting mouse input called GetAxisRaw which can be seen used in *Figure 10*.

```
Vector2 mouseDelta = new Vector2(Input.GetAxisRaw("Mouse X"), Input.GetAxisRaw("Mouse Y"));
```

Figure 10: Getting mouse input.

Using these inputs from the mouse the player's camera is then rotated based on velocity and sensitivity of the mouse. This can be seen done in *Figure 11*.

```
// Rotate camera up-down and controller left-right from velocity.
transform.localRotation = Quaternion.AngleAxis(-velocity.y, Vector3.right);
character.localRotation = Quaternion.AngleAxis(velocity.x, Vector3.up);
```

Figure 11: Moving camera and player controller.

The next thing that was added to the player object was a script called PlayerStats to hold the players stats and to allows for them to take damage and die. This script contains variables holding things like current health, max health and points. It also contains functions for taking damage, for dying and for regenerating health. The regeneration function used a useful Unity feature called coroutines which allowed for the use of the built-in function WaitForSeconds which allowed for time to be waited between each addition of health. The regeneration function is shown in Figure 12

```
IEnumerator RegenerateHealth()
{
    if (health < maxHealth && health != 0f)
    {
        Debug.Log ("Regenerating Health..");
        yield return new WaitForSeconds(5f);
        health += 10f;
    }
}
```

The next task was to create the environment that the player was going to be fighting enemies in. A free asset on the Unity Asset Store was found that fit the aesthetic that was desired for the game which was of a dungeon. How assets work in Unity is that you download the asset, and it is available to be accessed in the Unity editor. In this case, all that was required was to drag and drop the assets that were desired from the file into the scene. Getting assets that are already made is useful as they already contain not only the art but a mesh renderer which allows a mesh collider to be made to that the player cannot move through it. The asset that was chosen can be seen in *Figure 13*.

Figure 13: Environment Asset.

Creating the environment that the player would play in was a case of putting together all the premade assets such as walls, floors, stairs and barrels by dragging and dropping them into the scene and moving them to the desired place.

The next task was creating an enemy for the player to fight. Initially, the enemy was just a red cylinder that had a script attached that made it follow the player and attack them when it entered a certain range, but it was updated later in the project. Creating the enemy consisted of creating a new object, giving it a collider so that it cannot move through walls and a script to tell it what to do. An important component that the enemy required was a Nav Mesh Agent. What this component does is allows for certain parts of the environment to be made available for the enemy to move on so when the enemy is chasing the player it can only go on the specified area. The blue

highlighted area in *Figure 14* shows where the nav mesh agent can go.

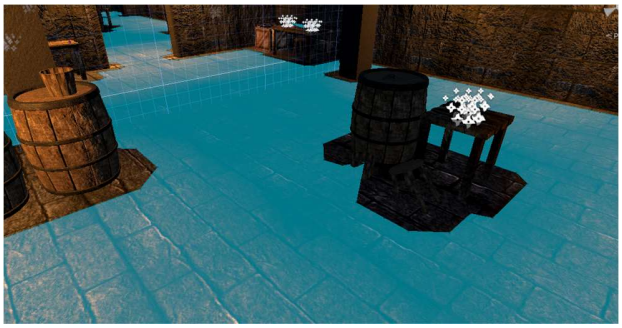


Figure 14: Baked area

It was decided that the enemy would have two states which were to chase the player and to attack the player when within a certain range. The script EnemyAI was created and added to the enemy to facilitate this. The first thing that was required in the script is to check the distance between the enemy and the player. Unity has a useful function called CheckSphere which will check for a specific object in a specified range. *Figure 15* shows the check for range then puts the enemy in either of the states depending on the range.

```
playerInSightRange = Physics.CheckSphere(transform.position, sightRange, whatIsPlayer);
playerInAttackRange = Physics.CheckSphere(transform.position, attackRange, whatIsPlayer);

if (playerInSightRange && !playerInAttackRange && !isDead) ChasePlayer();
if (playerInAttackRange && playerInSightRange && !isDead) AttackPlayer();
```

Figure 15: Distance Check

If the enemy was not within range to attack the player, they will just move towards the player using the nav mesh agent. *Figure 16* shows the SetDestination built in function being used which will cause the enemy to move at the speed specified in the nav mesh agent towards the position of the player.

```
agent.SetDestination(player.position);
```

Figure 16: Enemy Chasing

When in close range of the player it will attack the player if it has not already attacked within a certain time.

```
if (!alreadyAttacked)
{
    FindObjectOfType<AudioManager>().Play("EnemyAttack");
    animator.SetBool("isRunning", false);
    animator.SetBool("isAttacking", true);
    playerToAttack = GameObject.Find("First Person Player");
    PlayerStats stats = playerToAttack.GetComponent<PlayerStats>();
    stats.TakeDamage(attackDamage);
    alreadyAttacked = true;
    Invoke(nameof(ResetAttack), timeBetweenAttacks);
}
```

Figure 17: Enemy Attack Function

The code in *Figure 17* shows that when the enemy gets in range it will refer to the enemy stats component contained in the player object and get it to run the script that removes health from the player. A script was the created called EnemyStats which is very similar to the player stats in that is contains a function for the enemy to lose health and to die.

The last thing that was done in the first sprint was to create a way for the player to shoot the enemy that is chasing it with a gun and kill it. This backlog item was split into multiple parts which were shooting the gun, reloading and swapping weapons. Assets for the guns were conveniently found on the Unity asset store so what the guns looked like was already taken care of.

The first thing that was done was allowing for weapons to be swapped. To do this, an object called WeaponHolder was created, and the guns were made a child of this object. A script called WeaponSwitching was then created to allows for the weapons to be switched when an input was entered. In the script the gun objects were stored in an array so that they could be accessed easily and then when either the scroll wheel was changed or a number was selected, the selected gun would be changed. *Figure 18* shows how the key presses were retrieved.

```
if (Input.GetKeyDown(KeyCode.Alpha1))
{
    selectedWeapon = 0;
}
```

Figure 18: Gun Selection

Then a function was created to set all guns to being set inactive other than the selected weapon.

The next thing that was done was adding a Gun to each of the guns so that they can be fired at enemies and do damage. In the Gun script, variables were set for things like damage, range, current ammo and fire rate. A function was then created to fire a bullet when the left click on the mouse was pressed which is shown in *Figure 19*.

```
if (Input.GetButton("Fire1") && Time.time >= nextTimeToFire)
{
    nextTimeToFire = Time.time + 1f / fireRate;
    Shoot();
}
```

Figure 19: Fire Input

To create the shooting function, a useful built-in feature called Raycasts was used. What a raycast does is shoot a ray in a straight line in the target direction and return the information about the first object hit by the ray. This was useful as it allowed for when the ray hit an enemy for them to take damage. The code for a raycast is shown in *Figure 20*.

```
RaycastHit hit;
if (Physics.Raycast(fpsCam.transform.position, fpsCam.transform.forward, out hit, range))
```

Figure 20: Raycast

Using a raycast allowed for there to be a check of whether the object hit had had the component EnemyStats which is

shown in *Figure 21*. The hit objects component could then be stored in a variable and the function for the enemy to take damage could be triggered.

```
EnemyStats stats = hit.transform.GetComponent<EnemyStats>();
if (stats != null)
```

Figure 21: Component Check

Impact effects were then added where the raycasts hit and depending on what object the raycast hit. For example, *Figure 22* shows an impact effect being created when the enemy is hit and then being destroyed after 2.5 seconds.

```
GameObject impactGO = Instantiate(impactEffectZombie, hit.point, Quaternion.LookRotation(hit.normal));
Destroy(impactGO, 0.25f);
```

Figure 22: Impact Effect

The last thing to add for the gun was a function for the player to be able to reload bullets. This function was either run if the number of bullets became 0 or if the player pressed the key R. Creating the reload function was the first time that an issue was run into because it had to be made sure that the player could not reload their weapon and go into negative ammo. It was also the first time that Unity's animation player was used as it was desired for there to be an animation to let the player know that the gun is reloading. To create the animation for the reload, the whole weapon holder had to be rotated to that an animation didn't have to be created for each gun. Once the animation was created, a state machine for which animation the weapon holder should be in was created which can be seen in *Figure 23*.

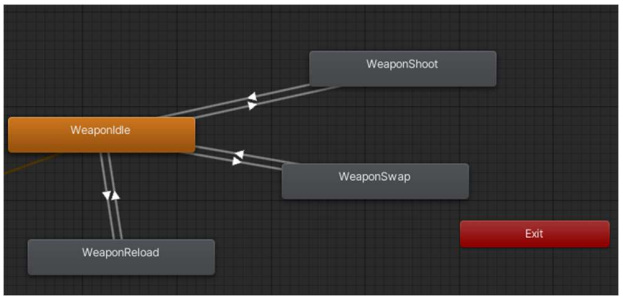


Figure 23: Animation Tree

Using this feature allowed for a Boolean variable to be made that could be used to turn the animation on and off and can be referenced in the script as seen in *Figure 24*.

```
animator.SetBool("Reloading", true);
yield return new WaitForSeconds(reloadTime - .25f);
animator.SetBool("Reloading", false);
```

Figure 24: Animation Trigger

The purpose of the reload function was to set the current ammo to the max ammo and then take the difference between those two variables away from the total ammo. This caused a bit of an issue due to the way it was coded initially as it used a loop to add a bullet one by one so that ammo couldn't be reloaded that was not available in the total ammo. The code for the loop can be seen in *Figure 25*.

```
while (currentAmmo != maxAmmo)
{
    if (totalAmmo != 0)
    {
        currentAmmo += 1;
        totalAmmo -= 1;
    };
}
```

Figure 25: Reloading Loop

The issue with this code is that it would crash the game if the gun was reloaded when the current ammo plus the total ammo was less than the total ammo as the loop would run endlessly. To fix this another if statement had to be created to fix this issue which can be seen in *Figure 26*.

```
if ((currentAmmo + totalAmmo) < 30)
{
    currentAmmo += totalAmmo;
    totalAmmo = 0;
}
```

Figure 26: Fixing Error

5.2 Sprint 3

As the main functionality for the combat between the player and the enemy had already been created in the first two sprints, it was time to create a way for the enemies to appear in the environment and to get increasingly difficult as the game progressed. The idea was to have multiple waves spawn one after the other once all the enemies in a wave were killed. To do this, spawn points were placed around the map and then a script called WaveSpawner was created to instantiate the enemy prefab on a randomly chosen spawn point.

The first thing that had to be done was to add functionality to the update method, which runs every frame, to spawn a wave if there are no enemies left and when the countdown before a wave has finished. This was done using a state machine which three different state which were spawning, waiting and counting. The script would start in the counting state and when in that state it will wait a specific number of seconds. Once the time has elapsed, it will then swap to the spawning state and begin to spawn a wave which can be seen in *Figure 27*. A function for spawning the wave was created and it takes the next wave to be spawned as a parameter and spawns the number of enemies specified in the wave which can be seen in *Figure 28*.

```
for (int i =0; i < _wave.count; i++)
{
    SpawnEnemy(_wave.enemy);
    yield return new WaitForSeconds(1f/_wave.rate);
```

Figure 28: Spawning wave

```
if (waveCountdown <= 0)
{
    if (state != SpawnState.SPAWNING)
    {
        StartCoroutine( SpawnWave ( waves[nextWave] ) );
    }
}
else
{
    waveCountdown -= Time.deltaTime;
}
```

Figure 27: Changing to spawn state.

The enemies are spawned individually in a function and are instantiated randomly based on the spawn point coordinates that are stored in the spawn point array. This can be seen in *Figure 29*.

```
void SpawnEnemy(Transform _enemy)
{
    Debug.Log("Spawning Enemy: " + _enemy.name);
    Transform _sp = spawnPoints[Random.Range(0, spawnPoints.Length)];
    Instantiate(_enemy, _sp.position, _sp.rotation);
}
```

Figure 29: Spawning enemies

Once all the enemies are spawned, the state is changed to waiting. While in the waiting state, the update function will constantly be running a function that checks if there are any enemies left still alive. If there are enemies alive, it will continue to stay in the waiting state and if there are no enemies left then it will swap to the counting state which will start the whole process of starting a wave again. The function that checks if an enemy us alive can be seen in *Figure 30*.

```
bool EnemyIsAlive()
{
    searchCountdown -= Time.deltaTime;
    if (searchCountdown <= 0f)
    {
        searchCountdown = 1f;
        if (GameObject.FindGameObjectWithTag("Enemy") == null)
        {
            return false;
        }
    }
    return true;
}
```

Figure 30: Checking if enemies are alive.

What the function in Figure 30 does is that it will check every second if there are no objects with the tag “Enemy”. If there are still enemies alive it will return true, if there are no enemies alive it will return false.

The last thing to add was to make the enemies stronger after each completed wave so code was added to the wave completed function which can be seen in *Figure 31*.

```
enemyHealth += 10;
//Increases movement speed of enemies after each wave
if (enemySpeed < 4f){
    enemySpeed += 0.2f;
}
```

Figure 31: Increasing enemy difficulty.

Once all the coding was done it allowed waves to be added to the array in the Unity editor. This allowed each wave to be assigned a name, the type of enemy to spawn, the number of enemies to spawn and the rate at which they spawn.

5.3 Sprint 4

This sprint was mostly made up of creating interfaces such as the heads-up display (HUD) and the main menu.

Figure 32: The HUD

Figure 32 shows what the final HUD looms like. Creating the display required using a thing in Unity called a Canvas. A Canvas allows for things like text, images and buttons to be overlayed on the screen and seen through the camera when the game is played. The first task was to add the text that was to be displayed on the screen which was done by adding text objects to the canvas and moving them to the desired location. It was then required to take information such as the current health amount, the number of points, current ammo and the wave number from different scripts. To do this, a script called PlayerUI was created to bring all the different variables needed to the same place and then assign them to the text objects that will be displayed on the screen. This was done using object referencing ang then getting access to that object’s components and then the values of variables contained in the component. An example of this being done is shown in *Figure 33*.

```
playerStats = GameObject.Find("First Person Player");
PlayerStats stats = playerStats.GetComponent<PlayerStats>();
```

Figure 33: Object reference

Once all the references to components were made it was then possible to use the built-in ToString function to assign the text objects the values contained in the variables. This can be seen being done in *Figure 34*.

```
healthAmount.text = stats.health.ToString();
waveNumber.text = waveSpawner.waveName.ToString();
points.text = stats.points.ToString();
```

Figure 34: Assigning text.

Extra functionality had to be coded so that the correct ammo for the gun that was currently active would be displayed. It just required getting the variable of the gun selected from the weapon holder script and creating an if statement for each gun which can be seen in *Figure 35*.

```
if (gun.selectedWeapon == 0)
{
    M4 = GameObject.Find("M4_8");
    Gun m4 = M4.GetComponent<Gun>();
    currentAmmo.text = m4.currentAmmo.ToString() + "/" + m4.totalAmmo.ToString();
}

if (gun.selectedWeapon == 1)
{
    AK74 = GameObject.Find("AK74");
    Gun ak74 = AK74.GetComponent<Gun>();
    currentAmmo.text = ak74.currentAmmo.ToString() + "/" + ak74.totalAmmo.ToString();
}
```

Figure 35: Gun selected correct ammo.

A major stumbling point was reached while creating this as an issue with the code was discovered due to an extra gun object with the same name as the one that was equipped by the player was put in the environment as decoration and was higher up in the unity hierarchy of objects so when the code for the gun was running it was referencing a gun object that had no scripts attached to it. It was a very easy fix of either removing or renaming gun in the environment. Many hours were spent refactoring code and looking for bugs in other scripts but sometimes that's how it goes.

The next thing that was created was the main menu. The creation of the main menu required a new scene to be made. The reason behind this is that when in the main menu the gameplay should not be running and when the play button is pressed it was desired that a new game would start. The main menu can be seen in Figure 36.

Figure 36: Main menu

The main menu consisted of buttons that when pressed would either swap to another scene or disable all the objects in the scene and make the options menu objects active. To make it so that the scene would change when a button was pressed, a script called MainMenu was created. What this script did was have functions for each of the buttons so that when the button was pressed the function would run and change the scene. This can be seen done in *Figure 37*.

```
public void PlayMap2()
{
    SceneManager.LoadScene(3);
}
```

Figure 37: Scene change function.

Unity stores scenes in an array and they can be accessed using the built-in function LoadScene that is shown in *Figure 37*.

Buttons in unity have the feature of being able to run a specific function when clicked. *Figure 38* shows how a function to can be selected in the button component to be run on click.

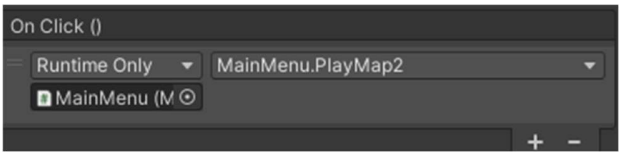


Figure 38: On click functions.

When the options menu was clicked it uses the same on click feature as shown above to set the main menu objects to be inactive and the options menu objects to be set active. This is shown in *figure 39*.

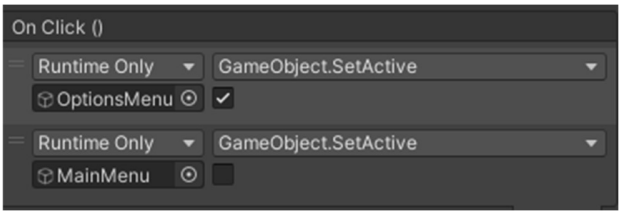


Figure 39: Options menu button

5.4 Sprint 5 and 6

These sprints consisted of doing user testing on the game that had been developed so far and then implementing any changes or additional features that were decided to be done after receiving feedback. The changes that were made based on the results of the testing can be found in the evaluation section. The only other thing that was done in the sprints what was planned to be done regardless of

the results of the user testing was updating the animations for the enemy.

It was decided that the enemy would be a zombie so an asset pack for a 3D zombie was found on the Unity asset store. The biggest advantage of finding this asset was that it already contained animations for things like walking, running and attacking. The animations that were chosen for the enemy to contain was a running animation and an attacking animation.

The first animation that was added was the running animation. To add the running animation, a new animation tree had to be created which can be seen in *Figure 40.*

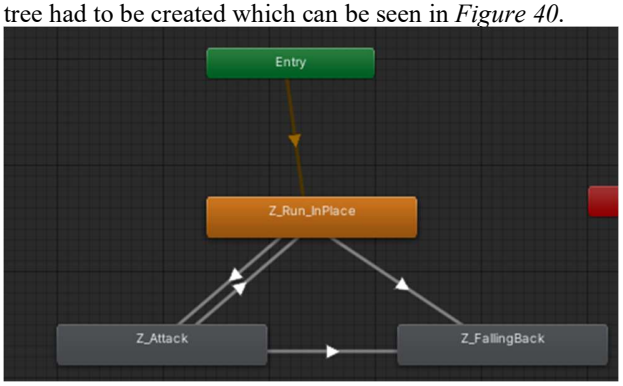


Figure 40: Enemy animations

How the tree works is that the boxes are the specific animation, and the arrows are the transitions between the animations. Within each arrow is a condition that checks a Boolean variable to see if the animation state should be changed. This can be viewed in *Figure 41*.

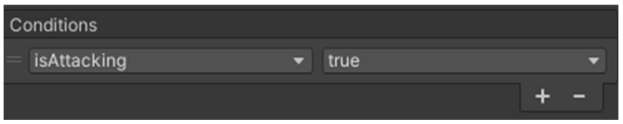


Figure 41: Boolean condition

What this condition allows is for the Boolean variable to be changed in a script so that the animation state can be changed when the enemy changes into a different state such as going from chasing the player to attacking the player. This can be seen being done in *Figure 42* where the SetBool built-in Unity function is changing the state of the variable so that the animation state is changed accordingly. In this case the animation is being changed from the running animation to the attacking animation.

```
animator.SetBool("isRunning", false);
animator.SetBool("isAttacking", true);
```

Figure 42: Animation changes

5.5 Sprint 7

This sprint consisted of updating the animations for the guns and adding buyable pickups for the player to upgrade.

For updating the animations for the gun, another two states had to be added to the animation tree which can be seen earlier on page 9 in *Figure 23.* Each of these animations were manually made using Unity's animation window which allows a key frame to be made at the initial point the animation is to start at and then it is possible to rotate or move the object and then create another key frame. This will create an animation of the object moving between the key frames and the speed and duration can be manually changed. Once the animations were created, it was a case of creating Boolean variables and transitions between the animation states then adding code to the scrips to change the Boolean variables when the animations are desired to be changed.

The developer ran into an issue while coding the animations as the shooting and swapping functions only last a frame before returning to the idle state instantly so the animation did not have time to start and would just go straight back to the idle state. The way the developer got around this was by using a coroutine which was mentioned earlier in the report to wait a small amount of time between starting the animation and returning to the idle animation state which can be seen in *Figure 43*.

```
animator.SetBool("Shooting", true);
yield return new WaitForSeconds(.005f);
animator.SetBool("Shooting", false);
```

Figure 43: Shooting animation.

The same thing was done in the weapon holder script to get the weapon swap animation working by changing the Boolean variable whenever the player swaps a weapon.

The next task in this sprint was to create pickup items that the player can buy using the points they gather from killing the enemies. The upgrades that were chosen to be available for the player were to increase the players movement speed, upgrade the players health and to upgrade the players' currently active gun.

To do this a new object was created and given a box collider. What a box collider can do is that it can be set to be a trigger so rather than stopping the player from entering, it allows functions to be run when the player enters or leaves the area of the box collider. The next thing that was done was to add a text object to the canvas so that when the player enters the collider, the text will appear on their screen telling them what to do.

What was required next was to code the functions required to allow the player to press a key and buy the upgrade using their points. This was done by creating a function for when the player enters the collider and when they exit. The box collider has built-in functions for when an object enters, is inside of or leaves the collider.

The function for when the player is inside the collider was created using the built-in OnTriggerStay function which can be found in *Figure 44*.

```
private void OnTriggerStay(Collider other)
```

Figure 44: Inside box collide function.

What this function does is take the object that enters the collider as a parameter, which is stored in the variable 'other'. Using this function allows for the object to be checked to see if it is the player which is shown in *Figure 45*.

```
if(other.gameObject.tag == "Player")
```

Figure 45: Checking object.

If the object is the player, then the text will appear on the screen and the player can input the key press to buy the upgrade if they have enough points to afford it. This function can be seen in *Figure 46* and in this case is the code for buying the speed upgrade.

```
if(other.gameObject.tag == "Player")
{
    PickUpText.SetActive(true);
    if(Input.GetButton("f"))
    {
        if (stats.points >= 2000f){
            stats.points -= 2000f;
            movement.speed += 3f;
            this.gameObject.SetActive(false);
            PickUpText.SetActive(false);
        }
```

Figure 46: Buying speed boost function.

Once the player leaves the collider the OnTriggerExit function is called and the text that has appeared when entering is set to inactive.

Creating the other upgrades was almost the exact same process of creating the object, adding a box collider, creating the text object in the canvas and the adding a script which does the exact same thing as shown earlier but the points value required is changed and rather than adding movement speed, health is increased, or weapon damage and max ammo is increased. *Figure 47* shows what buying an upgrade will look like from the players point of view.

Figure 47: Health upgrade

5.6 Sprint 8

This sprint consisted of creating a pause menu and adding sound to the game.

Creating the pause menu was like creating the main menu as it was a case of creating buttons on a canvas and running scripts when the buttons are pressed. However, in this case, rather than changing to a different scene when the game is paused, the pause menu just overlays the current game and freezes all the objects while it is open.

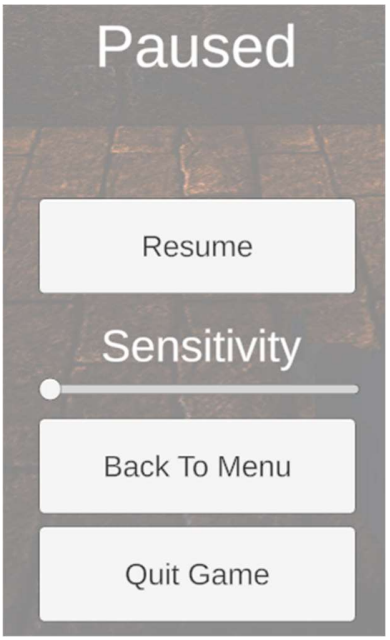


Figure 48: Pause menu.

Figure 48 shows an image of the pause menu. The buttons that were decided to be included were a resume button to continue playing the game, a button that sends the player

back to the main meu so they can start a new game and a button to close the game.

This was done by adding buttons to the same canvas that the HUD is contained in and setting them to be inactive. A script was then created called PauseMenu which contains all the functions necessary to make the buttons do what they are supposed to. In the update function, it waits for the key escape to be pressed and if it does it checks to see if the game is paused or not. If the game is paused, then in runs the resume function and if it is not paused it runs the pause function.

The pause function begins by making all the pause menu objects in the canvas become active. It then uses the built-in Unity time scale to pause the game by setting it to zero so no more time in the game elapses while the menu is active. It then unlocks the cursor so the player can use their mouse to select the buttons. The code for this function can be seen in *Figure 49*.

```
public void PauseGame()
{
    pauseMenu.SetActive(true);
    Time.timeScale = 0f;
    isPaused = true;
    Cursor.lockState = CursorLockMode.None;
```

Figure 49: Pause menu.

The resume function is almost the exact opposite to the pause function as it disables the menu objects, returns time to progressing normally and locks the cursor so that the player can aim properly with their gun. The send to main menu function just uses the LoadScene function that was mentioned earlier in the report when making the main menu to change the scene to the main menu scene. The quit function uses the built in Application.Quit function to close the game.

Once all these functions were completed it was a case of selecting the correct function in the buttons on click selector in the Unity editor which can be seen in *Figure 50*.

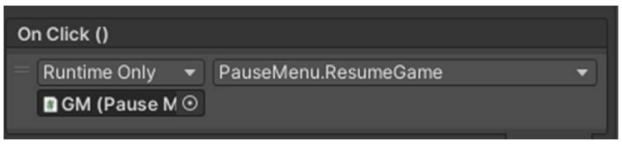


Figure 50: On click function select.

The next backlog task that was to be done in this sprint was to add sound and music to the game. There was a long list of all the different sounds that were desired in the game. A few examples of these sounds were a gun shooting sound, a gun reloading sound, an enemy attack sound, background music and the player being hit by an enemy.

To facilitate sound being added to the game, a game object was created called audio manager which was to hold all the sounds and allow for them to be referenced in scripts so that they can be played at specific times. To do this, a class called sound was made to hold all the variables that related to a sound such as their name, volume and the actual audio clip. A script called audio manager was then created to hold an array of sounds and then a function was created to be able to play the sounds which can be seen in *Figure 51*. What this function shown in *Figure 51* does is find the specific sound that is to be played from the array and then the sound is played.

```
public void Play (string name)
{
    Sound s = Array.Find(sounds, sound => sound.name == name);
    s.source.Play();
```

Figure 51: Playing sounds.

Having these scripts in place allowed for the variables of each of the sounds that were desired to be entered into the array using the Unity editor which can be found in *Figure 52*.

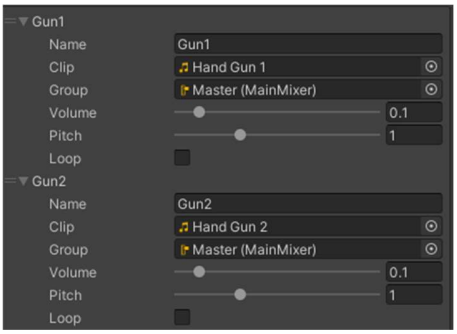


Figure 52: Audio clip array

Once all the audio clips that are necessary are added to the array, they can be referenced in other scrips to play the sound when they are required to be played. An example of this can be seen in *Figure 53* which finds the audio manager object that was created to handle the sounds and plays the desired audio clip.

```
FindObjectOfType<AudioManager>().Play("Reload");
```

Figure 53: Playing audio clips.

6 Evaluation / Testing

It was decided early on that there would be two sets of user testing. One would be done as soon as a playable version of the game was created and the other would be done once all the features that had been planned to be added at the start were built. The first set of testing was done during sprint 5 and the tasks that were added to the backlog based on user feedback were done in sprint 6.

6.1 Phase 1 of Testing

Microsoft Forms was the medium which was chosen to receive user feedback and all the resources were given to the testers using a private Microsoft share drive. The first step was to have the testers read through the participant information sheet which is a document containing information about the testing and an invitation to take part in the testing. The tester was then given the informed consent form to let them know what will be done during the testing and how their information will be used. The game was then sent to the testers by building the game in Unity which exports all the necessary files that are needed to run the game and contains an application file that when opened will run the game. The testers were then asked to play the game as much as they want and then fill out a Microsoft form which contained questions about their experience playing the game and what their thoughts were about it. There were six questions asked in the form which cab be seen in *Figure 54*.

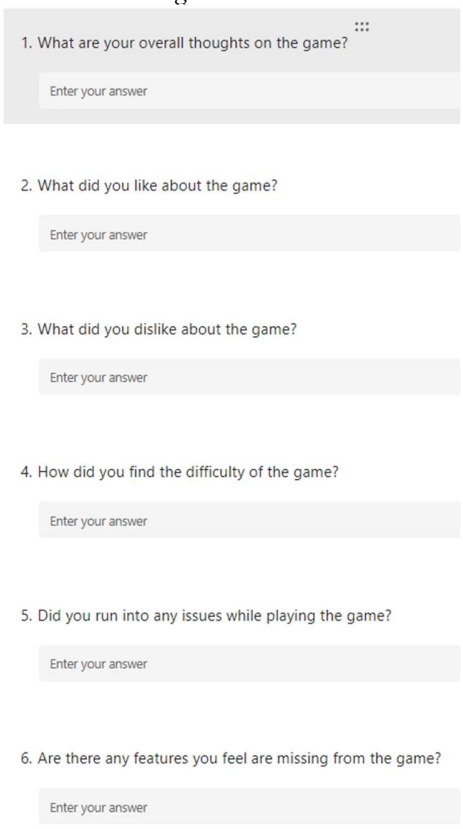


Figure 54: Questions for testers

There were six responses to the form and useful information was gained by taking their opinions into consideration. The full set of questions and responses can be found in appendix [10].

The first question asking about overall thought of the game did not give any information of real value other than the testers acknowledging that the game was still early in its development, but it was mentioned twice that they liked the atmosphere of the environment.

The second question asking what they liked about the game was more useful as it gave information about which features had been implemented well. It was noted again that the environment was aesthetically pleasing, and it was noted that the map design made the game more challenging which was intended so it was encouraging that it had been noticed by the player.

The third question about what they disliked held far more valuable information than the other two questions that had been asked to far in the form. It was noted by two different testers that they did not like the lack of feedback when hitting the enemy and when being hit by an enemy. This feedback was very useful as it was an aspect of the game that had not been considered previously and led to two items being added to the backlog which were to add more feedback when the enemies are hit and killed and some kind of notification when the enemies have hit the player other than the health score changing. It was also mentioned by three of the testers that there was no sound. This information was not so valuable as adding sound was already in the backlog and was planned to be done but it did reinforce the importance of sound and moved it further forward in the order or priority. The last thing of note that was mentioned in responses to this question was that the crosshair was too small and not visible enough so a task to change the crosshair was added to the backlog as it is important to be able to see where the bullets are going to go when shooting.

The fourth question, which was about the difficulty of the game, did not add too much information of value but did reinforce an already preconceived notion that the difficulty of the game would initially be very easy but that there would be a steep increase of difficulty once past around the fifth round. It was mentioned multiple times by the testers that they found the difficulty too easy in the first few rounds but far too difficult to survive when the rounds increased. This had already been planned to be improved by adding functionality to increase the players stats and weapon damage.

The fifth question about running into any issues was a little bit valuable as a tester discovered a bug relating to the enemies' animations not working properly so an item was added to the backlog to fix the issue. Another issue that was found was that when moving the player both

forward and to the side at the same time, the player would move faster than if they just moved one direction at a time. This was not an intended feature and felt unnatural when used so it was added to the backlog for that to be fixed.

The last question which was about what features were missing was very useful as many additional features that could be added to the game that were not considered by the player were mentioned which resulted in improvements to the game that would not have been made without the user feedback. The additional features that were added to the backlog from what was mentioned in this question were to be able to change the mouse sensitivity, which is a must in FPS games, to be able to reload the gun whenever the player wants rather than just when it gets to zero bullets left and a pause menu.

The full list of items that were added to the backlog can be seen in *Figure 55*.

Evaluation:

Add music and sound effects.

Add functionality to be able to change the aiming sensitivity.

Add functionality to be able to reload whenever the gun is not at full ammo rather than just when the ammo reaches 0.

Add functionality for headshots to do more damage so that skill can be expressed by the player.

Cross-hair changing functionality or bigger starting crosshair.

Add more feedback when the player is hit by an enemy.

Add functionality to be able to pause the game.

Figure 55: New backlog items based on user testing.

Once the analysis of the feedback was done, the rest of the sprint was spent fixing and adding features based of the feedback that was given.

The first thing that was added was a way to be able to reload the gun when a button is pressed. This was done by adding an if statement for when a button is pressed to run the reload function. This code can be found in *Figure 56*.

```
if (Input.GetButton("r") && currentAmmo != maxAmmo)
{
    if (totalAmmo == 0){
        return;
    }
    else
    {
        StartCoroutine(Reload());
        return;
    }
```

Figure 56: Reload on click function.

The next feature that was added was the ability to do more damage when the enemies are hit in the head. This was done by altering the enemy's collision object shape to cover only the body and then creating a new collision shape for the head. These collision objects can be seen in *Figure 57*, the green box covering the head and the green capsule covering the body are the collision objects.

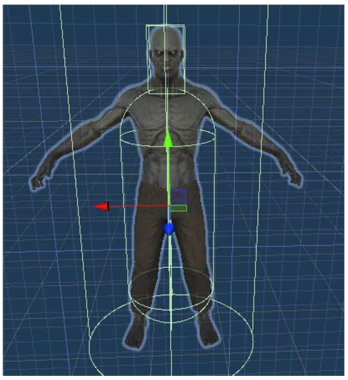


Figure 57: Collision objects

Having the two different collision objects allowed for a check to be made in the gun script when the raycast hits an object. When the head collision object is hit more damage will be done than when the body collision shape is hit. The code showing this check can be seen in *Figure 58*. As seen in the code below, the first line will check that the head collision shape has been hit, if it has then it will use the same take damage function, but it will double the damage being done.

```
else if (hit.transform.name == "Z_Head"){
    FindObjectOfType<AudioManager>().Play("Headshot");
    Debug.Log("Head hit");
    EnemyStats headshotStats = hit.transform.parent.GetComponent<EnemyStats>();
    headshotStats.TakeDamage(damage * 2f);
```

Figure 58: Headshot code

The next thing that was done was changing the crosshair to be more visible. This was done by increasing the size of the crosshair and changing it to a colour that did not blend in as much as the previous crosshair. The new crosshair can be seen in *Figure 59*.

Figure 59: New crosshair

The next thing that was done was to add a feature to indicate to the player when they are hit by an enemy. It was decided that the screen would flash red for a few seconds when the player was hit to make it obvious that they have taken damage. This was done by adding a red image to the canvas that had been sized up to fill the whole screen which has then had its alpha to zero so that it is invisible. The function for the player taking damage was then altered to contain code that would increase the alpha of the image so that the red screen would be visible which can be seen in *Figure 60*.

```
var color = gotHitScreen.GetComponent<Image>().color;
color.a = 0.8f;
gotHitScreen.GetComponent<Image>().color = color;
```

Figure 60: Getting hit screen.

Once the screen had turned red, a function to gradually reduce the alpha of the image was created so that the red screen would gradually fade away. The code for this function can be seen in *Figure 61* which shows a check for the alpha of the image being more than zero and for every frame that passes the alpha will be slightly decreased.

```
if (gotHitScreen.GetComponent<Image>().color.a > 0)
{
    var color = gotHitScreen.GetComponent<Image>().color;
    color.a -= 0.01f;
    gotHitScreen.GetComponent<Image>().color = color;
```

Figure 61: Hit screen decay.

6.2 Phase 2 of Testing

The second round of testing was decided to be done the same way as the first phase so that the difference in answers could be evaluated. The testing once again consisted of having the testers read and fill out the ethics forms then play the game for a while before answering the questions in the Microsoft form. The questions were kept the same as the six questions the first form so that it would be clear which areas had improved and which still required some work. All the questions and full answers can be found in appendix [10].

The first question of what their overall thoughts were on the game were far more positive than in the first phase of testing as multiple responses said that the game had improved and was more fun to play.

The second question of what they liked about the game was also responded to positively which testers mentioning that they liked the new additional features that had been added such as new animations, sound, buyable upgrades and the new map.

The third question of what they did not like about the game held far more informative and useful response than the first two questions. It was noted in multiple response that they felt that the second map was far too small for it to be enjoyable to play and that they preferred the first map. The developer took this into consideration and decided it would be better for the map to be bigger, so it was added to the backlog. It was also mentioned that the buy upgrade text that appears when you get in range of the upgrade does not tell the player what they are buying which is something that the developer had not previously noticed which was helpful and something that was a quick fix. It was also mentioned multiple times that the two different weapons were far too alike to the point that there was almost no need for there to be two guns, so it was decided to make the guns more unique.

The fourth question of how they found the difficulty did not have as much valuable information in the response. Most of the responses mentioned that they found the difficulty to be easy to being with but then get too hard as the game progressed. This was a feature, and the point of the game was to get the player to survive for as long as they can before they inevitably die so no items were added to the backlog to make changes in the difficulty.

The fifth question of if they had any issues while playing had some useful information as a few bugs were found by the testers. The first issue that was mentioned is that some of the testers were able to get on top of some of the small furniture which breaks the enemy pathing so that they will not attack the player. Another thing that was mentioned was that the sliders for sensitivity and volume did not stay in the same place when going in and out of the menus. The last issue that was found was that a tester managed to get outside of the map by getting pushed through furniture by an enemy. All three of these bugs were added to the backlog to get fixed.

The response to the last question of if they thought any features were missing held some valuable information. It was mentioned numerous times that the lack of variety of guns caused some players to get a bit bored of using the same gun all the time. This was deemed to be a valid point and a task to create more guns was added to the backlog. Another thing that was mentioned multiple times was that the lack of different enemies also made it somewhat boring after a while of killing the same zombie so a task to create a new type of enemy or different looking enemy was added to the backlog. The last thing of note that was contained in the responses is that the found the HUD to be a bit plain, so it was decided to add images and improve the user interface. The list of items that was added to the backlog can be found in *Figure 62*.

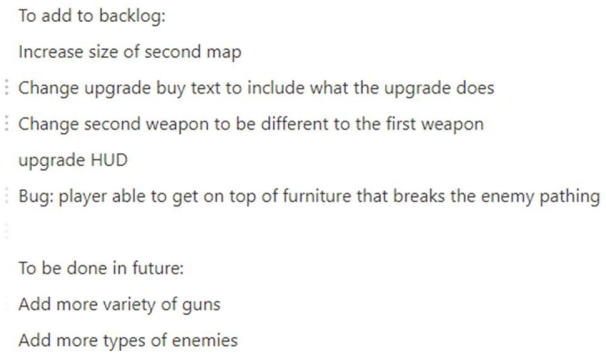


Figure 62: Phase 2 backlog items

It was decided due to only having limited time before the project would end that the creation of new guns and enemies would take too long to be able to get them done to

a good enough standard for them to be added to the final game so they were left out and could be done in future. However, all the other items that were added to the list were completed.

The first task of the ones added to the backlog that was done was increasing the size of the second map. The second map was created by turning all the objects in the first map scene into prefabs which is a Unity feature which allows whole objects and all their components to be saved in a file which can be dragged and dropped into a scene or referenced in code. A new scene for the second map was created and a different environment was added which was found from the Unity asset store which then had all the prefabs that were necessary for the game added the map. Increasing the size of the map was a case of opening more rooms of the environment that had already been created then updating the navigation mesh for the enemies to follow and then adding the new spawn points. The new room that was added to the second map can be seen in *Figure 63.*

Figure 63: New room

The next task that was done was changing the upgrade text to include what the upgrade does. This just required changing the text in popup text object on the canvas which can be seen in *Figure 64*.

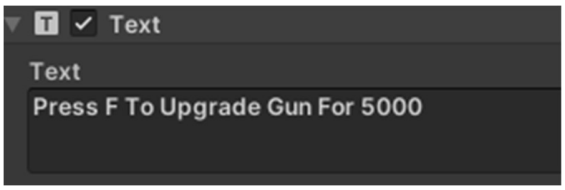


Figure 64: Changing text

Another task that was done was making the second gun different to the starting gun. It was decided that the fire rate and damage would be changed so that it would shoot slower but do more damage. Having this change would allow more skilled players to use the second gun to kill enemies quicker if their accuracy is good. The change was made by editing the value for each of the variables in the Unity editor.

The next thing that was done was altering the HUD to contain images to represent what the numbers at the bottom of the screen mean. This was done by adding images to the canvas next to the numbers and can be seen in *Figure 65*.

Figure 65: Added images.

The last thing that was done was removing the small pieces of furniture that were able to be climbed on. Rather than changing the height that the player can move up to be lower, it was decided just to remove the furniture that caused the bug. The reason that the furniture was deleted rather than changing the player was that if the step height that they can climb was reduced then they wouldn't be able to climb the stairs. Removing the furniture was done by just deleting them from the scene in the editor.

6.3 Ethics

Before starting the testing, it was important to make sure the whole process was done ethically. A group ethical submission was made for students that were doing testing that only required low risk activities such as questionnaires that do not ask for personal or sensitive details. This is what was used for the project and two documents were given to the participants to read over and sign to make sure they fully understood what they were getting themselves into and what the testing involved. The two sheets were a participant information sheet and an informed consent form which can both be found in appendix [4].

7 Description of the final product

The final product is a 3D FPS survival game that has the player fighting waves enemies while trying to stay alive as long as possible. When the game is opened, the player is greeted by the menu screen in which they can select either the first or second map, the options menu or quit. When a map is selected the player will appear in an environment which is either a dungeon or a castle. The environment contains furniture and debris which will impede the players movement and will contain portals which the enemies will spawn from. The player can move around the environment using their keyboard and they have two guns which can be used to shoot enemies that will spawn.

The user interface consists of the wave number and points score in the top left corner, and their health and number of bullets is shown at the bottom of the screen. The player can press the escape button to open the pause menu which

will pause the game and allow them to change their mouse sensitivity or go back to the main menu.

After about 5 seconds of being in the environment, the player will notice that a new wave has started, and zombies will start to appear from portals in set positions and will start chasing and attacking the player. The player will be able to run away from the zombies and shoot them with their gun to kill them. Once the player has killed all the zombies that have spawned in that wave, a new wave will spawn. The zombies in the new wave will be slightly faster and have more health than the ones in the previous wave which will make it harder for the player to survive as each wave passes.

Each time the player kills a zombie they will notice that their points score will increase, and these can be used to buy upgrades to their speed, health and guns by pressing the F key next to the buyable upgrade if they have enough points to buy it. They will also need to buy ammo as they have a limited supply of ammo that can run out.

The player will spend the time playing the game killing wave after wave of zombies until they inevitably die, and their objective is to try and survive as many rounds as possible before dying.

8 Appraisal

If the project were to be done again there are a few things that would have been done differently having the knowledge of hindsight. One of these things would be to have a far more consistent work schedule. As the project was done using an Agile methodology, each week several items would be selected to be done from the backlog and due to procrastination and getting side tracked, a lot of the work was done in the latter part of the week which meant that a lot of time was wasted, and less work was done that would have potentially been possible.

Another thing that was recommended to the developer multiple times but was not taken to heart that would be listened to if done again was starting the report earlier and while developing the game. Rather than having to rush and do lots of writing at the end of the project, doing a little bit every day would have made the final stages of the project a lot more relaxed and the final quality of the report would be a lot more refined and thought out.

One more thing that would have been done differently would be asking more questions during user testing and asking more detailed questions. During testing, due to the few numbers of questions, the quantity of feedback from testers was quite small as it was not indicated to them how much to write for each question so having more questions would have mitigated this problem and given more information to work with.

Using Unity was definitely a good choice as it allowed to important features to be focused rather than having to worry about developing things like graphics processing and collision detection which would've been time consuming and taken away from the quality of the final game. It was also a bonus that the scripts were written in C# as the developer already had brief experience using it.

Agile was also a good choice to use as it allowed changes to the plan of what tasks were to be done easy after getting new information from user testing. It also allowed for some testing to be done earlier as a minimum viable product was able to be created due to analysing the backlog and focusing on the most important features early on.

If advice was to be given to someone doing a similar project, it would be to build a habit of consistently doing work on the project every day and to write small bits of the report at the same time as working on development.

9 Summary and Conclusions

In summary, the report details what the aim of the project was, which was to create a 3D FPS survival game, the background research that was done to learn more about what sort of things could be in the game, the way that the project was planned to be done using Agile, what the requirements for the game were, how it was implemented and how the game was tested.

In conclusion, the developer is satisfied that all the functionality that was in the original specification was implemented, and many features were altered and added based on user feedback. They are also satisfied that a fully playable game has been created that can be enjoyed by players and is something that can in future be further developed.

10 Future Work

There were a few things that were mentioned as additional features that could be added in future that were not able to be implemented die to the time constraint. One of those being a wider variety of guns the player could use. This could be done buy adding similar functionality to the buyable upgrades but instead of changing the player or guns stats, it could add an extra gun or swap out the one that it currently equipped. This would allow for much more interesting gameplay as each gun could have a unique playstyle based on how the player wants to play.

Another additional feature that could be added in future is more types of enemies. Currently the game only has one type of enemy which is a zombie, and it was mentioned during testing that killing the exact same enemy over and over can get boring so adding more types of enemies or

different looking enemies could be added in future. This could be done by looking for assets for a different type of monster and using a similar enemy AI but giving them different movement speed, health and attacks.

Another good feature that could have been added but was only thought of too late in the project to be implemented was adding controller support. As the developer has only played games using mouse and keyboard for many years, they forgot that playing games with a controller was a thing so adding the functionality to be able to change the controls from using mouse and keyboard to be able to use a different kind of controller would open the game up to a wider range of players.

The last thing that could be considered in future is publishing the game so that players in the wider world can play the game rather than only people it is given to. There are multiple ways that a game can be published but the one that would make the most sense due to it being made as a PC game is to publish it on Steam [29]. Unity allows games to be published and even sold using their free licence until the game makes over $200,000 annually so there would not be any roadblocks to publishing the game.

Acknowledgments

The links for all the assets that were used including images, 3D models and sounds can be found in appendix [9].

References

[1] Wikipedia. First-person shooter. Available: https://en.wikipedia.org/wiki/First-person_shooter

[2] Wikipedia. List of video game genres. Available: https://en.wikipedia.org/wiki/List_of_video_game_genres

[3] Bryan Wirtz - Updated on April 11, 2023. Making your own video game engine: The beginners guide. Available: https://www.gamedesigning.org/learn/make-a-game-engine/

[4] Wikipedia. Agile software development. Available: https://en.wikipedia.org/wiki/Agile_software_development

[5] Adobe. Waterfall Methodology: A complete guide. Available: https://business.adobe.com/blog/basics/waterfall#:~:text=The%20Waterfall%20methodology%20%E2%80%94%20also%20known,before%20the%20next%20phase%20begins.

[6] Wikipedia. Tower Defence. Available: https://en.wikipedia.org/wiki/Tower_defense

[7] Ninja Kiwi. Play Bloons Tower Defence. Available: https://ninjakiwi.com/Games/Tower-Defense/Bloons-Tower-Defense.html

[8] Ironhide Games. Kingdom Rush. Available: https://www.ironhidegames.com/Games/kingdom-rush

[9] Wikipedia. Mario Cart. Available: https://en.wikipedia.org/wiki/Mario_Kart

[10] Wikipedia. Need for Speed. Available: https://en.wikipedia.org/wiki/Need_for_Speed

[11] Minecraft. Available: https://www.minecraft.net/en-us

[12] Animal Crossing. Available: https://www.animal-crossing.com/new-horizons/

[13] PlayStation. The Last of Us. Available: https://www.playstation.com/en-gb/games/the-last-of-us-part-i/

[14] Valorant. Available: https://playvalorant.com/en-gb/

[15] Wikipedia. Counter-Strike: Global Offensive. Available: https://en.wikipedia.org/wiki/Counter-Strike:_Global_Offensive

[16] Call of Duty. Cold war: Zombies. Available: https://www.callofduty.com/uk/en/blackopscoldwar/zombies

[17] Wikipedia. Doom Eternal. Available: https://en.wikipedia.org/wiki/Doom_Eternal

[18] Warhammer40k: Darktide. Available: https://www.playdarktide.com/

[19] Wikipedia. Game engine. Available: https://en.wikipedia.org/wiki/Game_engine

[20] Wikipedia. Unreal Engine. Available: https://en.wikipedia.org/wiki/Unreal_Engine

[21] Unreal Engine. Available: https://www.unrealengine.com/fr/blog/epic-announces-unreal-engine-marketplace-88-12-revenue-share

[22] Wikipedia. GameMaker. Available: https://en.wikipedia.org/wiki/GameMaker

[23] Wikipedia. Unity. Available: https://en.wikipedia.org/wiki/Unity_(game_engine)

[24] Unity. Asset Store. Available: https://assetstore.unity.com/

[25] Halo Fandom. Infection. Available: https://halo.fandom.com/wiki/Infection

[26] Wikipedia. MoSCoW method. Available: https://en.wikipedia.org/wiki/MoSCoW_method

[27] Game Programming Patterns. State. Available: https://gameprogrammingpatterns.com/state.html

[28] Game Programming Patterns. Component. Available: https://gameprogrammingpatterns.com/component.html

[29] Steam. Available: https://store.steampowered.com/

Appendices

Appendix [1]: Software and source code
Appendix [2]: User manual
Appendix [3]: Minutes of meeting with advisor
Appendix [4]: Ethics declaration form
Appendix [5]: Risk assessment form
Appendix [6]: Mid-term report
Appendix [7]: Poster
Appendix [8]: Design documents
Appendix [9]: Links to assets used
Appendix [10]: User testing results