Universitatea Politehnica din Timișoara

ZOMBIE BLITZ - FIRST PERSON SHOOTER GAME

Candidate:
Vlad-Mihai BIG

Scientific coordinator:
Asist. drd. ing. Raul BRUMAR
Ș.l. dr. ing. Stelian-Nicolae NICOLA

Session: June 2024

REZUMAT

Acestă teză prezintă dezvoltarea proiectului "Zombie Blitz," un joc video de tip first-person wave shooter. Obiectivul acestui proiect a fost să creeze un joc video distractiv și captivant care să încorporeze mecanici de joc precum sistemul de arme, inteligența artificială a inamicilor, scalarea dificultății, interfața utilizatorului, iar experiența să fie îmbunătățită prin elemente vizuale și audio, cum ar fi sistemul de animație, efectele vizuale și sunetul realist. Proiectul a fost conceput intr-un mod care să permită adăugarea ușoară de noi caracteristici de joc sau extinderea celor existente, iar eficiența a fost întotdeauna prioritatea principală în programarea mecanicilor. Jocul a fost dezvoltat folosind motorul Unity, cu C# ca limbaj de programare principal.

ABSTRACT

This thesis presents the development of "Zombie Blitz," a first-person wave shooter video game. The objective of this project was to create a fun and engaging video game that incorporates game mechanics like the gun system, the enemy artificial intelligence, difficulty scaling, the user interface, and with the experience being enhanced by visual and audio elements such as the animation system, visual effects, and realistic audio output. The project was designed to allow for an easy addition of new game features, or expansion of the existing ones, and efficiency has always been the top priority when programming the mechanics. The game was designed using the Unity engine, with C# as the primary programming language.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SNIPPETS

1. INTRODUCTION

1.1 CONTEXT

In the past two decades the video game industry has been growing at a fast pace, and advancements have been made in essentially all aspects, ranging from realistic graphics to more complex gameplay, and better story. Currently there are estimated to be 3 billion gamers worldwide, and this number is expected to increase in the following years.[1] The gaming industry is estimated to be worth around $280 billion in 2024.[2][3] These statistics reflect the significant impact that gaming has on entertainment and culture. The industry is only expected to grow more, considering the latest technological advancements, such as the AI revolution or the growing popularity of virtual reality games.

The role of programmers in game development is to create the logic behind the game, and to bring to life the vision of the designers, while making sure that the game runs smoothly and as intended. Because game development can offer a combination of technical challenges and creative opportunities, it is seen as an attractive career path by many computer engineering students.

First-Person Shooter (FPS) describes the genre that contains all the video games having gameplay centered on gun fighting or generally weapon-based combat, with the key aspect that everything is seen from a first-person perspective, meaning that the player is experiencing the action directly from the eyes of the main character.[4]

Figure 1.1: First-Person Shooter example

The screenshot above taken from the game Counter-Strike: Global Offensive, which is one of the most popular first-person shooters in the world, and shows what a typical first-person perspective looks like.

Wave shooters form another video game genre, with the main principle being the influx of groups of enemies (called waves) at set time intervals. The waves get larger and more difficult as the player progresses through the game. Usually this type of games have endless waves, with the main objective of the player being to see how long he can survive, however not all wave shooters have an endless game loop.

We will talk more about the Unity game engine and development environment in the next chapter, but for now it is important to note that it is one of the most popular game development platforms, because of its versatility (being able to support both 2D and 3D games) and its user-friendly interface. It is an immensely popular tool for indie game developers or smaller studios, and to get a better idea of how widespread Unity is, it's worth noting that about half of the mobile games on the market are developed in Unity.[5]

1.2 MOTIVATION

After spending a semester abroad through an exchange program where I studied game development, and because I have always had a passion for gaming and game design, I decided on creating a video game for this diploma thesis project.

Developing this kind of project requires good programming knowledge for the implementation of the gameplay mechanics. Being skilled in algorithmics and being able to comfortably manipulate data structures are both useful aspects when trying to insure the efficiency and stability of the logic that runs the game. Understanding the principles of Object-Oriented Programming is essential to ensure the many components of the project interact with each-other in the manner that is intended. Creativity paired with problem-solving skills are needed throughout the development process, which are skills that a good software engineer should posses.

The reason a first-person wave shooter was chosen as the genre for the game is because it is a fairly simple initial concept, but it also allows for creative freedom, and it can be expanded through more complex gameplay mechanics.

1.3 OBJECTIVE

The objective of the project is the creation of a software game that allows the user (player) to take on the challenge of surviving multiple waves of attacking enemies using a varied arsenal of weapons, while seeing everything through the eyes of the main character(first person perspective). The game will offer a fast-paced, action-packed experience, which is meant to be fun but at the same time challenging.

The project incorporates a multitude of gameplay functionalities that interact with each other to provide this experience, with the player controller, gun mechanics, enemy AI and wave system working together to create the shooter aspect of the game. This core element is expanded through the audio-visual effects, including UI, animations, sounds, music, gun effects and more.

Another important objective is to create a project that can be scaled and expanded without the need of refactoring the existing code elements, while maintaining good technical performance through the use of efficient programming and project structure. These requirements necessitate good usage of object oriented programming.

1.4 SPECIFICATIONS

The project will consist of a Unity game project, that is meant to be played on PC. The hardware requirements for running this project are low. From the tests conducted on 4 different machines of varying capabilities, it seems that the following minimum requirements would ensure a smooth experience:

- Operating system: 64-bit Windows 7
- CPU: Intel Core 2 Duo 1.8GHz, AMD Athlon X2 64 2.4GHz
- GPU: Nvidia GPU GeForce GTX 660 / AMD GPU Radeon HD 7870
- RAM: 6GB
- Storage: 4GB available space

The software is designed to be played by a single player, using a keyboard and mouse for input, with the output being displayed on a monitor screen of any resolution. An audio output device is not required to play the game, but it is strongly recommended.

2. STATE OF THE ART

In this chapter we will analyze a few games that are similar in concept or mechanics to "Zombie Blitz".

2.1 LEFT 4 DEAD

Left 4 Dead is a co-op action horror game developed by Valve, for the PC and Xbox 360. The game "casts up to four players in an epic struggle for survival against swarming zombie hordes and terrifying mutant monsters." It is one of the most popular and influential video game from the zombie genre.[6]

Figure 2.1: Left 4 Dead game logo

The game is designed around four-player cooperative play, with players work together to survive against hordes of zombies and special infected enemies. The goal is to move from safe house to safe house, culminating in a final escape sequence. In addition to the standard hordes of zombies, players face unique special infected enemies with distinct abilities:

- Boomer: A bloated zombie that can vomit bile, attracting more zombies to the players.
- Hunter: A fast, agile infected that can pounce on players from a distance.
- Smoker: A zombie with a long tongue that can grab and drag players away from the group.
- Tank: A powerful, hulking zombie that can deal massive damage. This was the inspiration for the "Monster Zombie" in Zombie Blitz.
- Witch: A passive enemy that becomes highly dangerous if disturbed.

Players have access to a variety of weapons, including pistols, shotguns, rifles, and melee weapons. They also find items like health kits, pain pills, and Molotov cocktails to help them survive.

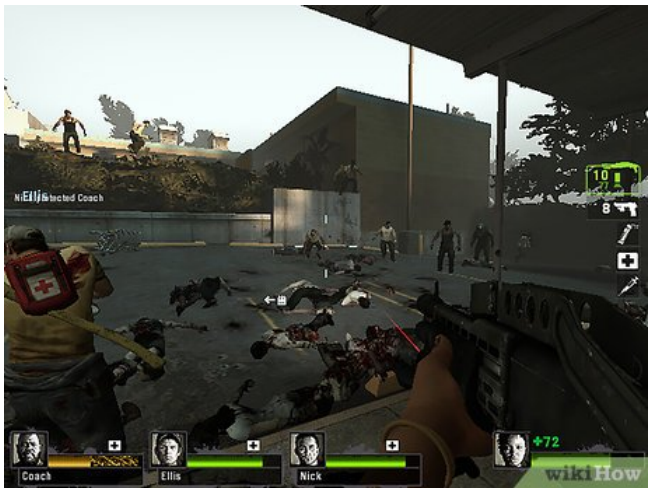

Figure 2.2: Gameplay screenshot from Left4Dead

2.2 NO MORE ROOM IN HELL

Figure 2.3: "No More Room In Hell" game logo

"No More Room in Hell" is a cooperative first-person shooter and survival horror game developed as a modification for the Source Engine by Lever Games. The game is heavily inspired by classic zombie movies, particularly George A. Romero's "Night of the Living Dead" series. It was released as a free-to-play game on Steam in 2011.[7]

No More Room In Hell places emphasis on cooperation, with up to eight players teaming up to survive against waves of zombies. Similar to Left 4 Dead, cooperation and teamwork are crucial for survival. The game emphasizes realism with limited ammunition, a lack of crosshairs, and the need to manage health and resources carefully. Players must scavenge for supplies and make every shot count. The game features different types of zombies, including slow shamblers, fast runners, and children zombies. This variety requires

players to adapt their strategies accordingly.

Players have access to a range of weapons, from melee weapons like axes and crowbars to firearms like pistols, shotguns, and rifles. A feature that sets it apart from other zombie shooters like Left 4 Dead is the fact that besides the arsenal of weapons, the players must use tools such as flashlights and radios, which are crucial for survival.

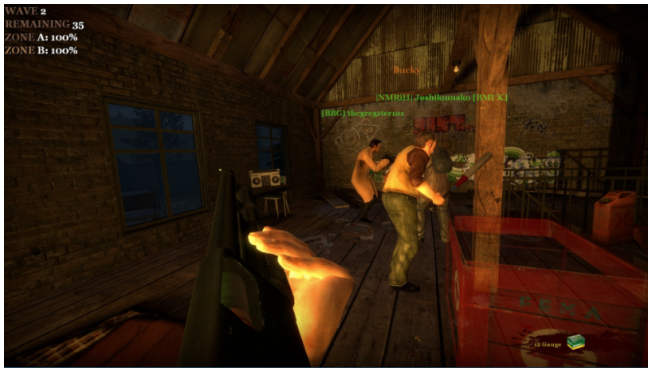

Figure 2.4: Gameplay screenshot from "No More Room In Hell"

The game features two game modes : Survival and Objective

- In Survival Mode, players must defend against waves of zombies while completing various objectives to survive as long as possible. This is an example of a game where the waves mechanic that "Zombie Blitz" features is used.
- In Objective Mode, players work together to complete a series of objectives to escape or achieve a specific goal. This mode often includes more narrative elements and structured progression.

2.3 ZOMBIE

I had to include a game that is similar in scope and resources to my own project, "Zombie Blitz". The previous games we talked about were developed over the course of years by entire teams of developers and designers working full-time to deliver the end product. This makes it easy for this project to appear lackluster or unpolished, but it is important to keep in mind that this was a game designed by only one person, in a much shorter time, and it is meant as more of a proof-of-concept rather than a game designed for commercial use.

However, the game we will discuss next can be a good example of what a beginner indie developer might accomplish, which gives us a better perspective of the quality of "Zombie Blitz" as a project.

Figure 2.5: "Zombie" game logo

"Zombie" is a single-player, first-person wave shooter in which the goal of the player is to survive as long as possible, in the face of countless zombies trying to take him down. The game was developed by a developer called EcoShooter, and released in 2018 on Steam.[8] The game costs only two Euros, and there are very few game reviews for this game (5 reviews), denoting the small number of players that have played this game and fact that this is a very small project developed and released on Steam with the hope of racking a few sales before it gets lost in the countless sea of games that are available on that platform.

Figure 2.6: Gameplay screenshot from "Zombie"

The game features a fairly simple gameplay loop: You are surrounded by zombies, and need to keep shooting the undead to keep your "adrenaline" meter from reaching zero which results in a "Game Over". The graphics are very simple, and the player has only one weapon available. A time counter and a counter for the number of zombies killed are a way through which the player can keep track of score. The game is similar to "Zombie Blitz" in complexity, however it does not feature the same wave-based gameplay, with the zombies instead being continuously generated during the course of the game.

2.4 COMPARISON

We've looked at three games of different scopes from the "Zombie Shooter" category, with Left 4 Dead being a titan of this genre, No More Room In Hell being a more modest but still influential release, and a smaller project in "Zombie", showing what can be realistically expected from a beginner indie developer. Next we'll have a closer look and compare what gameplay features are offered by each of these games, including the project developed for this thesis, "Zombie Blitz".

Game	Left 4 Dead	No More Room In Hell	Zombie	Zombie Blitz
Wave-based	No	Yes, in Survival Mode	No	Yes
Multiple weapons	Yes	Yes	No	Yes
Multiple zombie types	Yes	No	No	Yes
Co-op gameplay	Yes	Yes	No	No
Multiple maps	Yes	Yes	No	No

Table 2.1: Weapon stats

As we can see, Zombie Blitz tries to implement some of the key features found in the most influential games of the genre, like the player having access to an arsenal of weapons, and multiple zombie types that add diversity and challenge to the gameplay. "Zombie Blitz" cannot be fairly compared to these games, but it tries to take as many positive aspects from them and, using the limited resources that were available to me, integrate these aspects to create a fun, and more importantly extendable proof-of-concept game project.

3. TECHNOLOGIES USED

3.1 UNITY

Figure 3.1: Unity Logo

3.1.1 UNITY GAME ENGINE

Unity is a cross-platform 2D and 3D game engine that was first developed in 2005 by Unity Technologies. It was created with the purpose of providing more developers access to game development tools, which was a novel venture back when the engine first came out. Unity's main focus is to make it as easy as possible for game developers of any skill level to use the engine.[9]

Unity is the most popular game engine in the world, having the biggest user base and with a large pool of games developed in Unity across digital game stores like Steam, Epic, GOG etc.[5]

The biggest factor that contributed to the immense popularity of Unity (and the reason I chose this engine for the development of my project) is the intuitive interface of the editor and the extensive documentation that is available, providing a user-friendly experience for a developer, all while being a free service(with entirely optional scalable licencing options). The process of finding and importing free, community-made assets for the game is an easy one, which was another relevant factor when it came to picking the correct engine for this project. Some other reasons for Unity's worldwide popularity include its' versatility, being suitable for the development of 2D, 3D, Augmented Reality and Virtual Reality applications, and the option for cross-platform development from a single codebase.[9]

"Zombie Blitz" was developed using Unity version 2022.3.8f1

3.1.2 UNITY REAL-TIME DEVELOPMENT PLATFORM

The official name for the development platform through which video games are created in Unity. It offers a multitude of quality-of-life features, like the ability to edit while the game is running, the properties of game objects like size, location, rotation, or custom values created by the developer, like damage output, acceleration and such. This was immensely useful when play-testing, and debugging because as a developer I could quickly find the settings and properties for the game that offer the best experience.

Figure 3.2: Screenshot from the Unity scene editor

In the figure 3.2 we can see a typical layout of the scene editor view in Unity. Although the layout can be heavily customized, this is one of the most popular styles, with the hierarchy of the game objects from the current scene on the left, the scene view in the middle, list of components attached to the currently selected game object on the right, and the console and project file tree in the bottom of the screen.

In the Unity editor, every level is called a scene, and everything contained in that scene is called a game object. A game object can be an element from the environment like a building or a tree, the player, his weapon, or a collection of game objects, like multiple body parts grouped together to form the player's body. A game object can also be empty, not representing any element from the scene, but still serve a purpose like the Enemy Manager, Game Manager, Event Manager.[10]

Components are the functional pieces that you attach to a game object to give it certain properties and behaviors. We can think of a game object as an empty container or a blank slate, and components as the various tools and features you can add to make the

game object do something specific. The most basic component, that every game object (even empty ones) have is the transformer, which simply stores information about the game object's position, rotation and scale. Other examples of components are colliders, audio sources, audio listeners rigidbodies, and perhaps most importantly scripts -which allows the developer to add custom functionality and logic to game objects through code.[10]

3.1.3 MECANIM ANIMATION SYSTEM

Unity has its own integrated animation system called Mecanim. The "Animator" component is attached to game objects that need animation, and the "Animator Controller" is used to manage the animations and their transitions. At its essence, the Animator Controller is a state machine that defines how and when the animations transition from one to another, with each individual animation representing a state. Transitions are based on conditions and parameters.[10]

Each individual animation sequence is called an "animation clip" and can represent a movement or an action, like walking, jumping, and such. They can either be recorded through Unity's Animation window, or imported from external applications like Blender or Maya.[10]

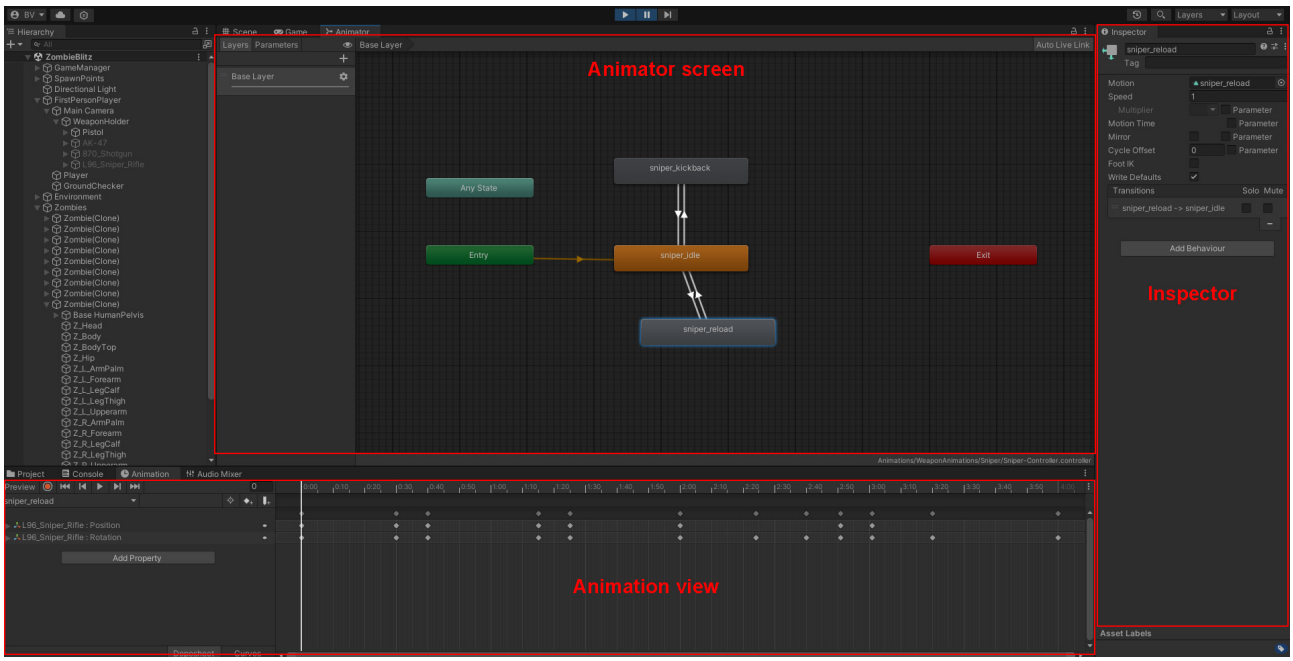

Figure 3.3: Screenshot from the Unity animator view

Figure 3.3 contains 3 important elements:

1. The animator screen, showing the animation clips and transitions that together form the Animator Controller for the sniper weapon. This can be thought of as the logic behind the whole animation system for that weapon.

2. The inspector. Here can be found more information about the animation clip or transition that is currently selected. Things like the speed of the animation, the transitions tied to

it, and in the case of transitions, the conditions that trigger them.

3. The animation view shows the properties that are manipulated at different timestamps throughout the animation. A series of positions and rotations at different moments for a game object is what ultimately results in the animation. Unity does the transition from one position to another automatically, and the speed of this movement depends on how long is the timeframe between the two moments.

3.1.4 NAVMESH SYSTEM

Unity's built-in NavMesh system is used for implementing AI-driven navigation and pathfinding in 3D environments. It allows entities (called agents) to intelligently navigate through the game world, avoiding obstacles and efficiently reaching their destinations.

A Navigation Mesh or NavMesh for short, is an abstract data structure made up of a collection of two-dimensional convex polygons generated from the level geometry, that define which areas of the environment are traversable by the agent.[11][12]

The navmesh is "baked" (generated) from the level environment[13], while taking into consideration user-defined properties like agent radius, maximum slope or step height. Then the NavMesh Agent component can be added to game objects, which allows for further customization.This enables the use of the navigation system from scripts by calling methods tied to the NavMeshAgent component of the game object.[10]

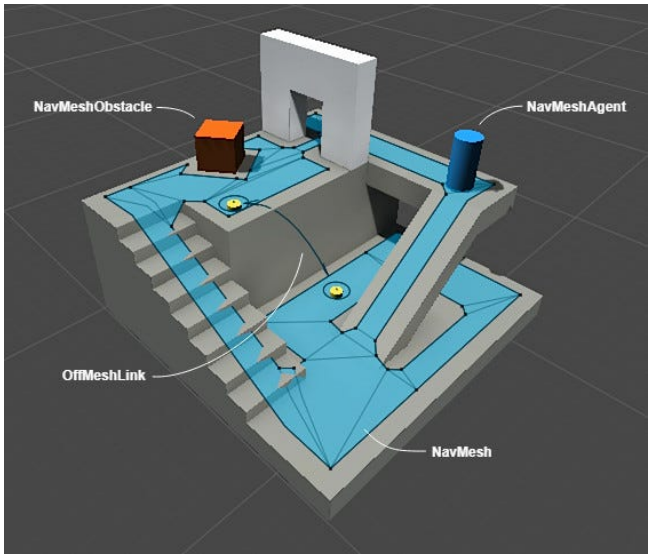

Figure 3.4: A NavMesh example[14]

In figure 3.4 we can see a basic example of a NavMesh in a scene, as the blue highlighted area on the ground. The NavMesh avoids the orange obstacle.

3.1.5 UNITY ASSET STORE

An important factor that allowed for the creation of this game project was the existence of the Unity Asset Store, because as a simple programmer without any knowledge in 3D modelling and design, I needed a platform through which I could easily import visual assets to the project. The asset store offers community made assets ranging from character models, environment props, textures, effects and more. Most importantly, one can also find a multitude of free assets here.

Not all of the models used in this project were imported from the asset store, with roughly one quarter of the assets being imported from sources like Mixamo, DevAssets, Freesound.org and others.

3.2 VISUAL STUDIO 2022

Figure 3.5: Visual Studio Logo

Visual Studio is an integrated development environment created by Microsoft and used for the development of software applications like mobile apps, websites and web services. It supports many of the most popular programming languages including C Sharp, C++, Python and JavaScript. The code editor uses IntelliSense (the code completion component) and allows for refactoring and real-time code analysis. The integrated debugger works as both a source-level debugger and as a machine-level debugger.[15][16]

I picked Visual Studio as the IDE for this project because of its versatility and features. This also seemed the best development environment to use for programming using C#, which is the language that was used for writing the scripts in this project.

3.3 C#

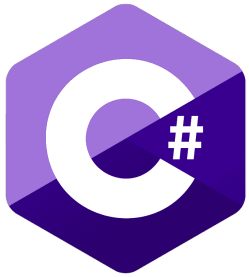

Figure 3.6: C logo

C# is a general-purpose, object-oriented programming (OOP) language created by Microsoft and used to develop a wide range of programs, including enterprise software, video games, and mobile apps. C# originates from the C family, and is similar to other popular languages like C++ and Java, which makes it easy for programmers to switch to C# or vice versa.[17]

Because it is a language built around the principles of OOP, the programs gain a clear structure which allows code to be more easily maintained and reused, lowering development costs. Being a high-level programming language, it is easier to learn because commands use words similar to everyday language rather than abstract code.[17]

C# is a popular programming language in game development due to its overall efficiency and compatibility with the Unity game engine.

3.4 GIT/GITHUB

Git is a free and open source distributed version control system developed by Linus Torvalds in 2005. The main feature that sets Git apart from most other software configuration management tools is its branching model, which allows having multiple local branches that can be entirely independent of each other. [18]

While the use of Git is mostly intended for projects where multiple developers work in parallel, I still chose to work with git throughout the development process of "Zombie Blitz" because it allowed me to easily track changes, revert mistakes manage code history in a reliable and flexible manner.

GitHub is a web platform that uses Git software, and provide users with features like access control, software feature requests, and task management, on top of the already existing version control capabilities provided by Git. It has many significant benefits, including seamless collaborations, templates, integrations, and public repositories that give your projects exposure if desired.

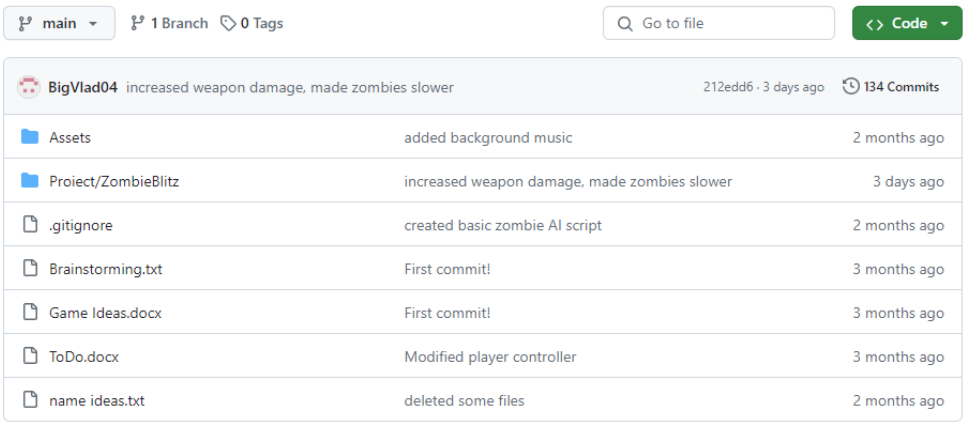

Figure 3.7: The GitHub repository for "Zombie Blitz"

Figure 3.7 shows my repository for this project, with 134 commits spanning across 3 months.

4. STRUCTURE AND DESIGN

4.1 GAME FLOW

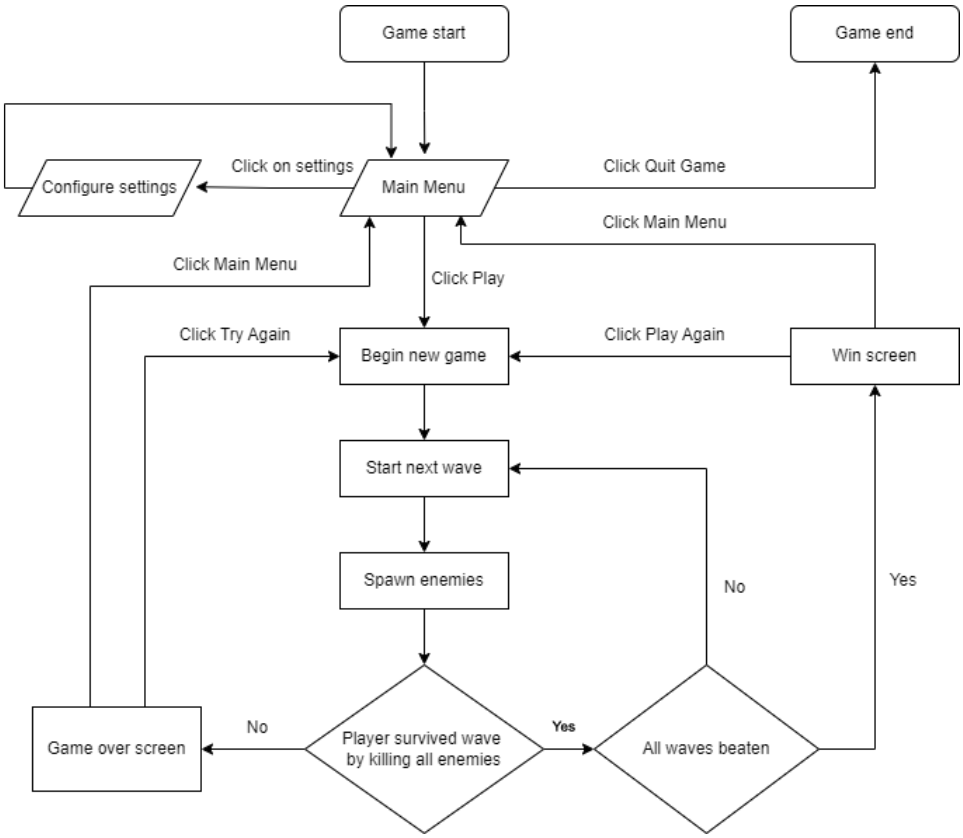

Figure 4.1: Zombie Blitz game flowchart

Figure 4.1 represents the flowchart of the game. The first thing the player sees is the main menu, where he is presented with three options: play the game, go to the settings screen, or quit the game.

When the player presses play, the scene is changed and game objects are loaded, including the map assets, the player, weapons, sound clips, effects and scripts. Then the player has a few seconds to move and look around before the first wave starts and zombies are spawned at random locations throughout the map. This time-frame occurs between all waves. Then once the enemies appear, they will immediately chase towards the player, who needs to eliminate them while also trying to keep the distance or else he will be overwhelmed and lose the game.

If the player manages to eliminate all the zombies, the cycle will continue until all waves are cleared. At that moment, the player will be presented with the win screen, which will display the number of enemies defeated as a form of score tracking. The player can then return to the main menu or he can click play again to start over.

If at any moment the player loses too much health by failing to keep a distance from

the zombies, the player loses the game and is presented with the "Game Over" screen, which similarly to the Win Screen, will show some stats like the number of enemies defeated and the number of waves survived. Again, the player can choose to go back to the main menu or he can click "Try Again" for another chance at winning the game.

In the settings screen, the player can use the volume slider to configure the sound volume of the game. In the current version of the game the settings screen is more of a proof-of-concept, with more options planned in the future such as a difficulty setting or and "endless" game mode.

4.2 WAVE SYSTEM

The whole game revolves around the concept of "wave shooter", where player has to survive attacks from multiple waves (groups of enemies) of increasing difficulty. There are two approaches in game design with regards to the way in which waves appear:

- In the first approach, after spawning a wave, wait for the player to defeat all the enemies. Once he succeeds, give the player a short time window to prepare for the next enemy attack. Only after the time window expires, the game will proceed to the next wave. This approach leads to a more systematic gameplay. In the timeframe between sets, the player may acquire upgrades or seek a better location to defend against the next wave.
- In the second approach, once the enemies have finished spawning, immediately start the countdown until the next wave. Once the countdown is finished, start spawning enemies regardless of whether the player has beaten all the enemies from the previous wave. This approach makes the game feel faster and more intense, as the player will want to try to eliminate his enemies as soon as possible to avoid getting overrun once the next wave starts.

For my project I chose to go with the first option, because it allows the player a bit of breathing time between waves, time which could also be used in future versions of the project for acquiring upgrades or obtaining ammunition.

4.3 DIFFICULTY FACTORS

Playing Zombie Blitz is intended to be fun, but also challenging. Because of the limited scale of this project, I wanted the game to have an "arcade" feel, meaning it's designed to be played in short intervals, with the player losing frequently. If the game would be too easy, then the player would complete it once, without too much effort, and never play it again. Instead, if the game is challenging, there is an incentive for the player to keep playing the game, in order to see how far he can get, compared to his previous attempts.

To this end, there are four main aspects to this project working together when it comes to difficulty and balancing. These are:

1. The enemies

2. The waves
3. The weapons
4. The map design

4.3.1 ENEMIES

The fist and most important element that creates the challenge in the game are the enemies. Without their presence, all the other aspects of the game become redundant and the player lacks any purpose or objective. Their appearance is what drives the gameplay forward.

As the name of the project suggests, the enemies in this game take the form of zombies. Their only objective is to chase down the player and attack him.

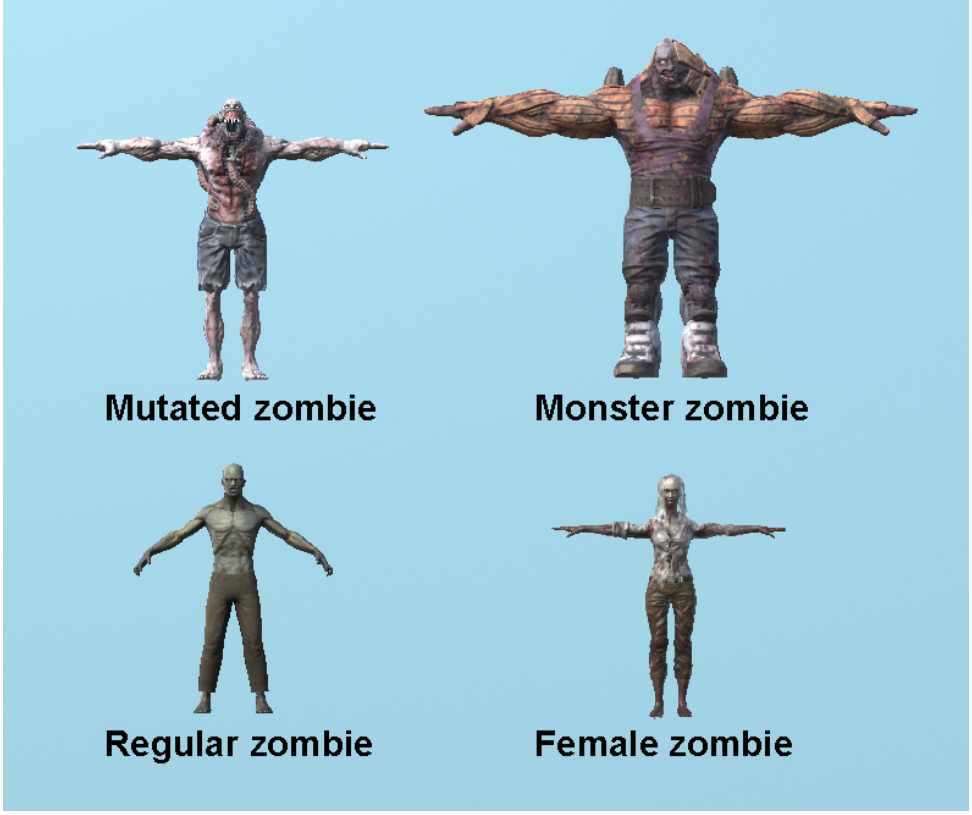

Figure 4.2: Zombie types

As shown in the image above, the game contains four enemy types, each with his unique look and different stats to add diversity to the gameplay: regular zombie, female zombie, mutated zombie, and monster zombie.

To implement these variations between the enemy types without having to write multiple scripts or overcomplicating the existing "enemy AI" script, I made use of the "Scriptable

Object" element from Unity. According to the manual, "ScriptableObject is a serializable Unity class that allows you to store large quantities of shared data independent from script instances." In essence this allows us to save data as an asset in our project files, which can be used at runtime.

How this was done is we created a class called EnemyData which inherits from ScriptableObject. This class stores static information about the zombies, such as movement speed or attack damage. Then we can easily create files with the .asset extension for each zombie type, which store information in the format of the EnemyData class and can be accessed at runtime.

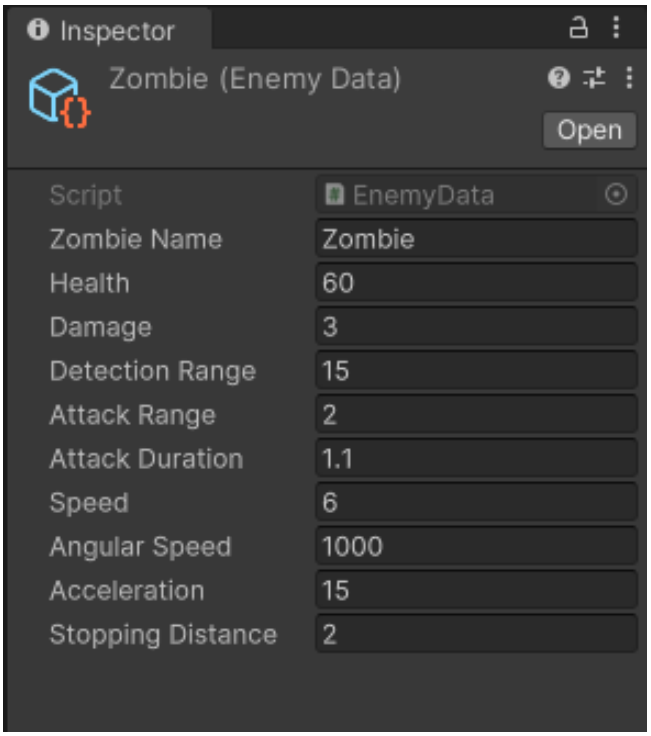

Figure 4.3: Regular zombie stats

Figure 4.3 shows the scriptable object file which stores information about the regular zombie. The explanation for each of the stats:

- Zombie name: the type of zombie this scriptable object refers to. This is useful for scenarios where we need to take into consideration the enemy type.
- Health: the maximum health points this zombie type has.
- Damage: how much damage the zombie's attacks cause.
- Detection Range: this feature is disabled in the game, but it was used as an experiment during development. It causes the zombie to start following the player when the latter gets withing Detection Range meters from him, and stop following when the player exits the detection range. In the final game this feature is disabled and the zombies always follow the player no matter the distance between them.

- Attack Range: The maximum distance from the player, in meters, at which the zombie can initiate an attack on the player.
- Attack Duration: How long one attack lasts for this zombie type, in seconds.
- Speed: The maximum speed at which this enemy type can move.
- Angular Speed: How fast this zombie type can turn and rotate when running, to adjust his movement trajectory.
- Acceleration: How fast the zombie accelerates (or decelerates) when moving.
- Stopping Distance: The maximum distance from the player, at which the zombie will stop chasing. In practice this field is always equal to Attack Range, but it does not always have to be this way. One can consider a scenario where Stopping distance is one meter, but the attack range is two meters. The zombie will run until he is within one meter of the player, and since he is also within attacking range, he will initiate an attack. Then, let's say that the player moves back 0.5 meters, so still within attack range, then the zombie will keep attacking, and only when the player exits his attack range will the zombie start moving towards the player again, and seek to be withing stopping distance from him.

Each zombie type has different stats, in order to add diversity and make the player's approach require an element of strategy when dealing with these types.

Zombie Type	Regular zombie	Female zombie	Mutated zombie	Monster Zombie
Health	60	50	300	1500
Damage	3	3	6	15
Attack Range	2	2	3	3
Attack Duration	1.1	1.1	1.3	1.6
Speed	6	7	9	4
Angular Speed	1000	1200	300	2000
Acceleration	15	15	10	15

Table 4.1: The stats for each zombie type

In the table above we can better analyze the values and stats for each zombie type. A fact that might not be very obvious from just looking at the table is that all of these zombies were designed to represent some classic enemy archetypes.

The regular zombie is the most basic enemy type, and the first one that the player encounters. Their stats are designed to be balanced, with relatively low health and damage output making them easy to defeat. Their movement is not very fast and not very slow either. This enemy type is meant to form the crowd amidst which the other enemy types will stand out.

The female zombie is very similar to the regular zombie, with slightly lower health, but being able to move faster.

The mutated zombie represents a more dangerous enemy because of his increased damage, and it is also harder to take down due it having a lot more healthpoints. On top of this, the mutated zombie is the fastest out of all the enemy types, but its low angular speed and acceleration make it more clunky the the other zombies, which allows skilled players to maneuver around this zombie.

The monster zombie is meant to serve as a sort of "mini-boss", being extremely hard to stop, having a massive health pool and huge damage. The downside of this enemy is that it is slow in both movement and attack speed, giving the player a chance to evade his attacks.

4.3.2 WAVES

Except perhaps for the last enemy type, the monster zombie, by themselves most enemies in the game would not pose much of a threat to the player. The player can take down a regular zombie or woman zombie using only two bullets (even only one if the head is hit). However, the notion of "strength in numbers" applies well here, and zombies become very dangerous when they group together to form hordes charging at the player. When we add the fact that because of the way the map is designed, the zombies can come from many directions it becomes obvious that a careless player will quickly find himself surrounded and overrun by enemies, which will spell the end of the game for him.

For each wave the number and type of enemies was picked manually. The goal was to make the first waves easy, to serve as a sort of tutorial portion of the game, in which the player may get used to the controls and mechanics, and then as the player advances through the game we make the waves increasingly more difficult by adding more enemies.

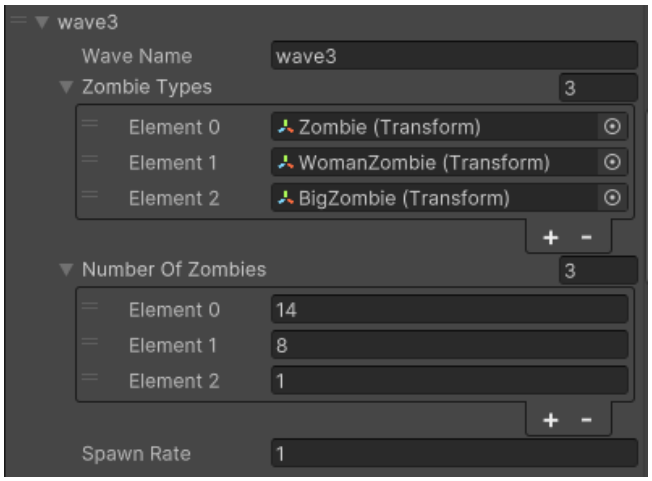

Figure 4.4: Example of a wave as seen from the editor

In figure 4.4 we can see what a wave looks like from the editor. There are two arrays, "Zombie Types" which the wave spawner uses to know which zombie type to spawn, and "Number of Zombies" which represents the number of zombies of the corresponding type that will be spawned during that wave. For example, in the image above Element 0 of "Zombie

Types" is the regular zombie, and Element 0 of "Number of Zombies" is 14, which means in this wave there will be 14 regular zombies, with the same principle applying for the other zombie types, which means there will also be 8 female zombies and one mutated zombie (called BigZombie in the game files). Another element worth mentioning is the Spawn Rate, which represents the time in seconds that passes between two enemies spawning. This is all just an interface for the code behind it, meant to make the testing phase easier because one can easily change the types and number of zombies for each wave directly from the editor. We will talk more about the code aspect of the wave system and the Wave class in the "Implementation" chapter.

Wave Number	Spawn Rate	Regular	Female	Mutated	Monster
1	3	5	3	0	0
2	2	10	6	0	0
3	1	14	8	1	0
4	0.5	16	8	3	0
5	1	20	10	5	1
6	1	0	0	8	3
7	0.5	25	15	10	4

Table 4.2: The number of enemies for each wave

Table 4.2 shows a more detailed look at the structure of each wave. Waves one and two are meant to be easy, to allow for the player to get accustomed to the controls and mechanics of the game. Only regular and female zombies appear in these waves, at a slower rate compared to the latter waver.

Wave three shows the introduction of the Mutated zombie, and the fourth wave is where the game starts to become challenging, with the number of enemies being increased substantially and the small spawn rate making two enemies appear every second. The player needs to always be aware of his surroundings to avoid getting surrounded and overrun.

The fifth wave marks the appearance of the Monster zombie, who serves as a sort of "mini-boss", because of how dangerous he is.

Wave six has no regular or woman zombies, so the total number of zombies decreases compared to previous waves, but the downside for the player is that there will be only more mutated and monster zombies, with eight and three respectively. This wave also serves as a sort of calm before the storm that is the upcoming wave

The seventh and final wave throws a massive horde of zombies at the enemy, with a very low spawn rate meaning the situation will quickly escalate for the player, who will have to try his ultimate best to survive this wave and defeat the zombies.

Once the last wave is completed, the player wins the game and is presented with the win screen.

4.3.3 WEAPONS

The four weapons the player holds are his only means of self-defence and survival against the oncoming hordes of zombies. These are the pistol, the AK-47, the shotgun and the sniper. Each weapon has its strengths and weaknesses, and some are better in certain use cases than others.

Figure 4.5: The weapons available in the game

For storing data about each weapon we used the same approach as we used with zombie types. A GunData class which extends Scriptable object was created, and objects of this class (called scriptable objects) will store information about each weapon.

The difference in this case is that while the zombie types scriptable objects only stored constant values, the scriptable objects for weapons also store one property which changes its value during runtime, namely the "Current Ammo". This is possible because at any point in the game there will only be one weapon of each type, while in the case of zombies, all agents of one type will take their data from the same scriptable object, so for example damaging one enemy would cause all enemies of that type to lose health.

To work around this, the health capacity of the zombie type is copied locally in the script that runs the AI, and so changes to that value will only affect that particular game object (zombie).

Figure 4.6: AK-47 gun data

In the figure above we can see the weapon stats for the AK-47. The explanations for each of the stats:

- Weapon Name: the name of the weapon;
- Damage: the damage output of the weapon;
- Range: how far in meters the weapon can shoot. Beyond this distance the weapon will not cause damage;
- Magazine Size: the number of bullets the weapon holds when reloaded
- Current Ammo: the number of bullets currently loaded in the weapon;
- Fire Rate: how fast the weapon can shoot, in bullets per second;
- Reload Time: how long it takes to reload the weapon in seconds;
- Is Automatic: a boolean value. If true then the player can hold down the mouse button to fire continuously. If false then the player has to click once every time he fires.
- Shooting Sound: stores the audio clip that is played when the weapon is fired.
- Reload Sound: stores the audio clip that is played when the weapon is reloaded.
- Dry Fire Sound: stores the audio clip that is played when the player tries to fire the weapon but the current magazine is empty (has no bullets).

- Dry Fire Duration: the length of the Dry Fire Sound clip. This field is required to make sure that the Dry Fire sound is not played multiple times overlapping.

Weapon	Pistol	AK-47	Shotgun	Sniper
Damage	45	35	90	180
Range	25	30	10	200
Magazine Size	12	30	6	5
Fire Rate	3	6	1	0.8
Reload Time	2	3	4	4
Is Automatic	No	Yes	No	No

Table 4.3: Weapon stats

Table 4.3 shows a more detailed description of each weapon's stats:

- While not particularly excelling in any aspect, except perhaps for its' quick reload time, the pistol is meant to be used more as a support weapon, when the other weapons are out of ammo and the player has no time to reload. Nevertheless, the pistol is still a very useful weapon when used correctly.
- The AK-47 will be perhaps the most used weapon, because of its high rate of fire and magazine size, making it perfect for crowd control against the weaker zombie types. The AK is also the only automatic weapon in the game, however the player needs to control his aim well, otherwise he will waste most of his bullets and quickly run out of ammunition.
- The shotgun is a formidable close range weapon that deals high damage, with the downside being its low magazine size and long reload time. It cannot shoot very far, so it is best used against stronger zombies like the mutated or monster zombie, as a well placed headshot will deal huge amounts of damage equivalent to multiple bullets from the pistol or AK-47.
- The sniper is by far the most powerful weapon when it comes damage, dealing devastating blows to enemies at what is in practice limitless range (there is no scenario where the player will need to shot something more than 200 meters away). Of course, the very limited ammunition capacity and long reload time offset this, and make this weapon highly effective for players with good aim, and not so useful for players who miss a lot of shots. Still, the sniper is probably the most effective weapon against the monster zombie (who has a health pool of 1500) especially when considering the body part damage multipliers, which we will talk about next.

The game includes a realistic body part damage multiplier system. This means that when the player shoots an enemy, the damage that is dealt depends on the part of the enemy's body that was hit. Therefore, a headshot will deal significantly more damage than a leg-shot or an arm-shot. The modified damage is calculated by multiplying the weapon's base damage

with a constant value dependant on the body part that was hit. The body parts and damage multipliers are as follows:

- Torso: x1.0 damage (normal damage);
- Head: x2.0 damage;
- Arms: x0.66 damage;
- Legs: x0.66 damage;

As we can see, aiming for the head can very rewarding, however the target is very small. Shooting the body will deal normal damage, and the limbs will decrease the damage dealt because zombies do not really care if their limbs are shot.

To explain the way we check what body part was hit, we first need to talk about colliders. In Unity, a collider is "a component that defines the shape of a GameObject for the purposes of physical collisions."[10].

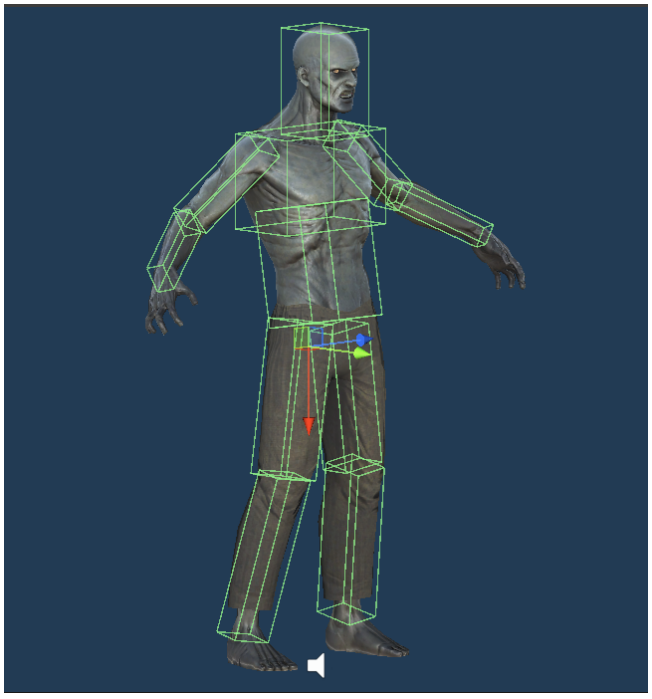

Figure 4.7: The colliders of the regular zombie

Colliders can be multiple shapes, but in this project we used the most basic (and efficient) type of collider, the box collider, which takes the shape of a parallelepiped. By combining multiple such colliders, we obtain a composite collider which resembles the overall shape of the object we want to emulate. Figure 4.7 demonstrates this point, as we can see there are box colliders tied to the major body parts, which come together to resemble the zombie. While not perfect, this level of detail is good enough for our game.

Now we can better explain how the code checks to see which body part (if any) was hit.

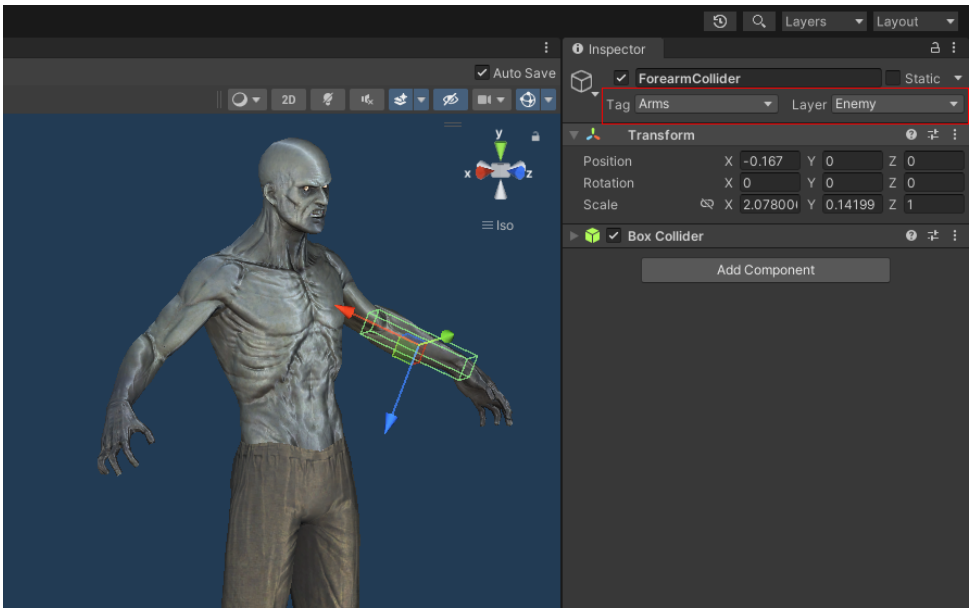

Figure 4.8: Layer and Tag of the forearm collider

When we shoot a bullet from any weapon, we cast a ray (an invisible, straight line in space) forward. We check to see if the ray intersects with any object's collider (which would mean the trajectory of the bullet will make it hit something). If we have a collision, we check the layer of the object that was hit. Layers are a useful feature for grouping game objects to more easily manage collisions. If the object that was hit has the layer "Enemy", that means we have hit a zombie, and it's time to calculate the damage that was dealt. For this, we look at the "tag" of the game object whose collider we've hit. We have four tags for this, which are the four body-parts' damage multipliers: Body,Head,Arms and Legs. As we can see in figure 3.8, when selecting the zombie's forearm, we can see the collider attacked to that body part, as well as the tag (Arms) and layer (Enemy) in the inspector.

4.4 MAP DESIGN

The game map represents the geometric space in which the gameplay takes place. Good map design is important because it creates the aesthetic feel of the game and adds to player immersion. It also helps drive gameplay forward, through smart environment design. Without a good map that brings to life the creator's vision and shifts the focus on the strong points of the game, while hiding the weaker aspects, a game with a good concept and mechanics will appear mediocre at best.

The game map in Zombie Blitz was created to resemble a city overrun by the zombie apocalypse. There are plenty of crashed cars, fallen trees and lampposts, trash and clutter everywhere. All this helps with immersion, but many objects were placed in a certain manner to create scenarios in the gameplay, with choke points, or open spaces, where the player can strategize and use the environment to his advantage.

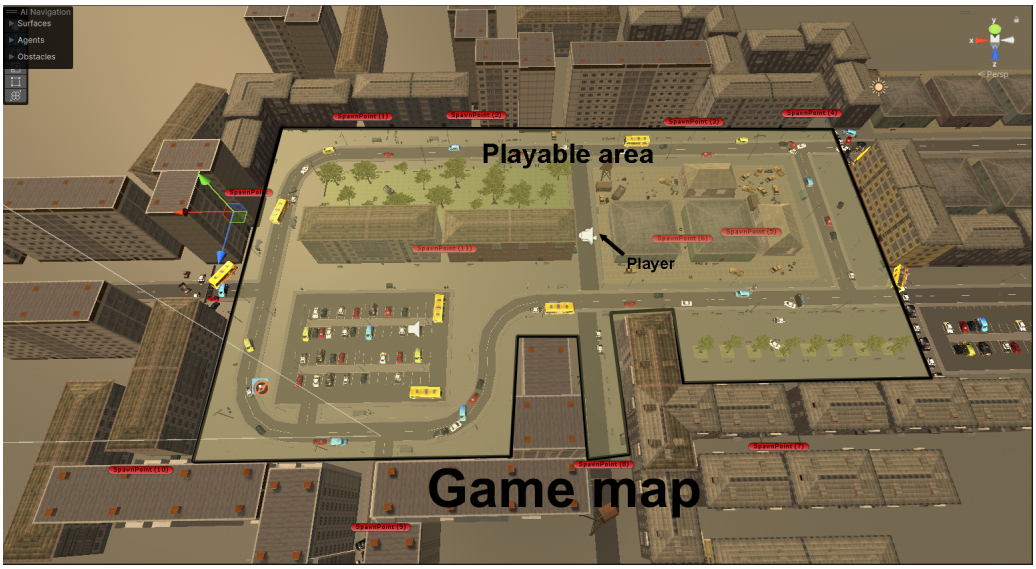

Figure 4.9: Zombie Blitz game map

Figure 4.9 shows an aerial view of the game map. The playable area is quite large, with two main roads united by several alleys and streets. The player spawns in the middle of the map, and may choose to go in any direction he decides. The playable area is closed off seamlessly using prop arrangements like car blocks or barriers that block the player from proceeding further without ruining immersion.

The red labels in the image represent the spawn points. These are the locations at which zombies spawn during gameplay. During the spawning phase at the beginning of the wave, for each zombie a random spawn-point where he will appear is chosen. This makes the enemies come from all directions, and in this way the game will always keep the player on alert due to the possibility of a zombie attacking him from behind. Most of the spawnpoints are located in small alleyways on the edge of the map. This was done in order to make sure the player does not see the enemies appearing out of thin air, thus improving immersion.

5. IMPLEMENTATION

5.1 GENERAL

There are a total of 27 scripts (not including static values script and scriptable objects) written for this project. These scripts, which are simply C# code files run during gameplay, manage a variety of functions ranging from player and camera movement, to enemy AI, "gunplay" system, or user interface (UI).

The game contains 4 scenes: The Main Menu, the "Zombie Blitz" scene where the gameplay takes place, and the WinScreen and GameOver scenes. When a new scene is loaded, all game objects and data (including scripts) from the previous scene are deleted.

Before we talk about each section in particular, we will go over the Static Values class and the Game Manager script.

The Static Values class stores data that is needed outside the "Zombie Blitz" scene (the level where the actual gameplay takes place).

```
1 using UnityEngine;
2 public class StaticValues : MonoBehaviour
3 {
4     public static int wavesSurvived;
5     public static int zombiesKilled;
6 }
```

Snippet 5.1: Static Values class

Snippet 5.1 shows the Static Values class. It currently stores the number of waves the player has survived and the number of zombies he managed to kill. These two values are needed to be displayed in the scenes "GameOver" and "WinScreen", as a way for the player to track his score.

The Game Manager script is responsible for changing the scene in the scenarios when the player loses or wins the game.

```
1  public IEnumerator GameOver()
2  {
3      GameObject.Find("Crosshair").SetActive(false);
4      FindAnyObjectByType<PlayerMovement>().enabled = false;
5      YouDiedText.SetActive(true);
6      Time.timeScale = 0.3f;
7      yield return new WaitForSeconds(2);              //display "You Died
           !" text for 5 seconds, then change to the game over screen
8      Time.timeScale = 1f;
9      Cursor.lockState = CursorLockMode.Confined;
10     Cursor.visible = true;
11     StaticValues.wavesSurvived = FindAnyObjectByType<WaveSpawner>().
           getWaveNumber()-1;       //these values are needed in the game
           over screen so they are stored in static values
12     StaticValues.zombiesKilled = FindAnyObjectByType<PlayerData>().
           getZombiesKilled();       //
13     SceneManager.LoadScene("GameOver");
14 }
```

Snippet 5.2: GameOver method from the Game Manager

Snippet 5.2 contains the code for the GameOver method. It disables the player's crosshair, disables player movement, display a text on the screen informing the player that he lost the game, and adds a slow motion effect to the game to make this event more dramatic.

In line 7 we tell the script to wait a number of seconds, (in which the player can still look around without moving), before continuing to parse the code. We accomplish this using the yield return statement, which is not a return in the classic sense, but creates a facility whereby the code can resume from where it left off. By calling yield return WaitForSeconds, we return the function WaitForSeconds(2), which is called, thus stopping the the execution of the code for two seconds, after which the script resumes from the line after the yield return.

Next, we set the time scale back to normal to stop the slow motion effect, enable the mouse cursor, and store in the StaticValues class the number of waves survived and zombies killed.

The last step is to make the switch to the GameOver scene.

A similar approach is used in the GameWon function for when the player manages to survive all the waves.

5.2 PLAYER

The player character represents the collection of game objects which together with their attached components and scripts create the avatar that the player controls in the game world allowing him to interact with the environment and complete the game's objective.

As we can see in figure 5.1, the FirstPersonPlayer game object represents the player character. The FirstPersonPlayer is made up of three game objects:

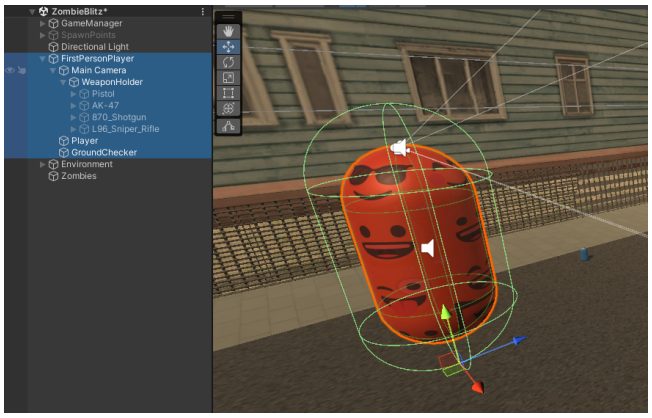

Figure 5.1: The player character and its game objects

1. The MainCamera, which has the Camera component, and in turn contains the WeaponHolder game object, which we will cover in the Gunplay section.

2. The Player game object which is simply a 3D object in the shape of a capsule. Because the game is played in a first-person perspective, the player will never see his own avatar except for the weapons he holds. Therefore there was no reason to invest time in creating or finding a realistic depiction of the player.

3. The GroundChecker, which is an object used in the PlayerMovement script.

In this project we use 5 scripts for implementing player-related functionalities: PlayerData, PlayerCamera, PlayerMovement and PlayerSound. Next we will cover their implementation in more detail.

5.2.1 PLAYERDATA SCRIPT

This script manages information about the player, such as his health and number of enemies he killed. It is also responsible for calling the GameOver method from the GameManager in the case of player death.

```
1    float startHealth= 100;
2    float currentHealth;
3    int zombiesKilled= 0;
4    bool playerAlive = true;
```

Snippet 5.3: PlayerData fields

The PlayerData class contains four fields. startHealth represents the player's health points at the start of the game. currentHealth indicates the player's health points during the game. Initially this field is set to match startHealth in the Start method (which is called once when the script is initialized). Then during the course of the gameplay, this value may change in the event that the player is hit by zombies, and once it reaches zero, the player dies. The

boolean playerAlive is true while the player's current health is a positive value, and is set to false once the currentHealth reaches zero (or a negative value). This flag is used to make sure the Die method is only called once and not multiple times. The field zombiesKilled simply counts the number of zombies the player has killed during the span of the current game.

There are two important methods contained in the PlayerData class, these being the TakeDamage and Die methods. There are also two getters (for the currentHealth and zombiesKilled) and a setter for zombiesKilled.

```
1  public void takeDamage(float damage)
2  {
3      gameObject.GetComponent<PlayerSounds>().playerHitSound();
4      currentHealth -= damage;
5      if (currentHealth <= 0)
6      {
7          Die();
8      }
9  }
10 void Die()
11 {
12     if(playerAlive)      //this is to make sure we don't call GameOver
           more than once.
13     {
14         playerAlive = false;
15         StartCoroutine(FindObjectOfType<GameManager>().GameOver());
16     }
17 }
```

Snippet 5.4: takeDamage and Die methods of PlayerData

The public method takeDamage is called by the ZombieController script when the enemies manage to hurt the player. On line 3 we call the playerHitSound method of the PlayerSounds script which is attached to this game object (i.e. the player). This makes the player produce grunting noises when hit by zombies. Then on line 4 we decrease currentHealth by the value of the "damage" parameter, which represents the damage capability of the zombie type that hit the player. Next we call the Die method if the new currentHealth value is less than or equal to zero.

The Die method calls the GameOver method from the GameManager. It sets playerAlive to false to make sure the GameOver method is called only once. The reason for this is because when the player dies, the game does not end immediately, instead it goes on for a few more seconds. In this time, the zombies continue to hit the player and so takeDamage is called multiple times, and by extension the Die method because the player's health is smaller than 0. This would mean the GameOver method is called multiple times, so to avoid that, we only call it once, when playerAlive is still set to true, and then set this flag to false.

5.2.2 PLAYERCAMERA SCRIPT

PlayerCamera is responsible for converting the player's mouse movement into camera movement in the game.

```
    public float sensX;
    public float sensY;
    public Transform playerBody;
    float rotationX;
    float rotationY;
```

Snippet 5.5: PlayerCamera fields

The public values sensX and sensY are used for configuring the mouse sensitivity when moving the camera. playerBody is a reference to the player's in-game body, needed to be able to make it rotate with the camera. rotationX and rotationY are the current rotations of the camera on the X and Y axes.

```
void Update()
{
    float mouseX = Input.GetAxisRaw("Mouse X")  * sensX;
    float mouseY = Input.GetAxisRaw("Mouse Y")  * sensY;

    rotationY += mouseX;
    rotationX -= mouseY;

    rotationX = Mathf.Clamp(rotationX, -90, 90);       //for the X
        rotation we need to clamp the value to make sure the player is
        not able to look behind him by looking up or down too much

    transform.rotation = Quaternion.Euler(rotationX, rotationY, 0); //
        rotate the camera
    playerBody.rotation = Quaternion.Euler(0, rotationY, 0);    //
        rotate the player's body when looking sideways. We do not rotate
        the player's body when looking up/down
}
```

Snippet 5.6: PlayerCamera Update method

The Update method is part of the MonoBehaviour class, which is the base class from which every Unity script derives. It is called once per frame, making it ideal for tasks that need to be checked or performed frequently, like handling player input in our case. Using Input.GetAxisRaw() method we obtain a value that represents the mouse input along the desired axis and we multiply it with sensX or sensY to apply the desired sensitivity to the camera movement. We apply these inputs to change the rotationY and rotationX fields. If the player moves the mouse to the right, the input is represented as a positive float value, on the

X axis. We apply this rotation on the Y axis of the camera, to rotate it to the right. The same is done for the mouse movement on the Y axis, only this time we subtract this value from the Y rotation of the camera. This is done because if a player moves the mouse up, the input is a positive float. But in order to rotate the camera upwards, which is the desired effect, we need a negative rotation on the X axis of the camera.

On line 9 we simply limit the possible rotation on the X axis to a value between -90 and 90, to make sure that the player is limited to looking directly down or directly up, but not be able to look behind by rotating his view on the X axis.

Next, we apply the obtained X rotation and Y rotation to the camera, and also rotate the player's body along the Y axis, when the camera rotates sideways.

5.2.3 PLAYERMOVEMENT SCRIPT

PlayerMovement is the script responsible for moving the player character when the player presses the corresponding keys on his keyboard.

```
1  //general
2  public CharacterController characterController;
3  public Transform PlayerBody;
4  public float movementSpeed = 12f;
5  public float gravity = -10f;
6
7  //input-related
8  float horizontalInput;
9  float verticalInput;
10 Vector3 movementDirection;
11 Vector3 velocity;
12
13 //ground-related
14 public Transform groundChecker;     //an empty GameObject placed on the
       foot of the player. Will cast a small invisible sphere to check if
       the player is on the ground
15 public float groundCheckRadius;     //radius of invisible sphere
16 public LayerMask Ground;
17 bool isGrounded;
18
19 //jump-related
20 public KeyCode jumpKey = KeyCode.Space;
21 public float jumpHeight=3f;
22 public float jumpCooldown;
23 bool canJump;
```

Snippet 5.7: PlayerMovement fields

The first two field of the PlayerMovement class is a reference to the Character Controller component of the player character game object. This component is useful for implementing character movement and collision detection without the need to write complex

physics code. The second field is a reference to the transform component of the PlayerBody game object which we talked about in the explanation for figure 4.2 It is represented by the red capsule from the screenshot. The next two fields are the movementSpeed which dictates the speed at which the player can move, and gravity which is the force with which the player is being pushed back on the ground when he jumps. These values are set to public for easier assignment directly from the editor.

The next four fields are related to input and movement. The horizontalInput and verticalInput are values which will represent the state of the left/right and up/down keys respectively. movementDirection is a vector in 3 dimensional space and it will represent the direction in which the player will move by combining horizontal input, vertical input, and gravity force applied (if any). The velocity field will indicate the speed at which the player is moving towards the ground, accelerated by the gravity force. The next four fields are used for knowing when the player is touching the ground. When the player jumps, we apply a force that launches him in the air, then we continuously add the gravity force in the downward direction to pull him back to the ground. But we need a way of knowing when the player has returned to the ground. groundChecker is a reference to the game object mentioned in the explanation of figure 5.1. It is essentially a point placed at the feet (or bottom end) of the player character. This point will cast a small invisible sphere in space, of radius groundCheckRadius, which will check to see if it detects any collision with the ground. If it does, it means the player is touching the ground, since the sphere is very small in radius. The Ground field, of type LayerMask is simply a reference to the Layer called Ground, so that we know which group of game object we consider to be part of the "ground". We talked about Layers briefly in section 3.3.3. As mentioned, they are a way to group game objects for easier collision management. In the game editor, we assign to every game object from the ground, like roads and sidewalks, the layer Ground. The boolean field isGrounded is true if the player is currently on the ground, and false if the player is not touching the ground (is in the air).

The last four fields are related to the player jump mechanic. jumpKey stores the key code of the key that when pressed will result in the player jumping. By default this key is set to "SPACE". The jumpHeight represents how strong the player's jump will be, the jumpCooldown is the time in seconds which the player must wait between jumps, and canJump is a boolean that is true when the player can jump and false when the player cannot jump either due to being already in midair or the jumpCooldown having not expired.

```
void CheckInput()
{
    horizontalInput = Input.GetAxisRaw("Horizontal");
    verticalInput = Input.GetAxisRaw("Vertical");
    if (Input.GetKey(jumpKey) && canJump && isGrounded) {
        canJump = false;
        jump();
        Invoke(nameof(resetJump), jumpCooldown);     //invoke resetJump
            after jumpCooldown time
    }
}
```

Snippet 5.8: PlayerMovement's CheckInput method

The CheckInput method is used for taking in the player's input. When taking in values from the keyboard, GetAxisRaw method returns a value between -1 and 1, with 0 being returned when no key is pressed. We take in the input for both the horizontal axis (left/right movement), and vertical axis (forward/backward movement).

If the player has pressed the jump key, and the conditions for jumping are met, then we invoke the jump method which simply launches the player upwards by a force equal to jumpHeight multiplied by gravity negated. We set canJump to false, and invoke the method resetJump after jumpCooldown seconds. resetJump will set the boolean canJump back to true.

```
void Update()
{
    isGrounded = Physics.CheckSphere(groundChecker.position,
        groundCheckRadius,Ground);   //cast an invisible sphere from the
        player's foot to check if player is on the ground
    if (isGrounded && velocity.y < 0f) {
        velocity.y = 0f;
    }
    CheckInput();
    if(horizontalInput != 0 || verticalInput != 0)
    {
        StartCoroutine(gameObject.GetComponent<PlayerSounds>().
            footstepSound());
    }
    movementDirection = PlayerBody.forward * verticalInput + PlayerBody.
        right * horizontalInput;
    characterController.Move(movementDirection * movementSpeed * Time.
        deltaTime);
    velocity.y += gravity * Time.deltaTime;
    characterController.Move(velocity * Time.deltaTime);
}
```

Snippet 5.9: PlayerMovement's Update method

In the Update method, which is called once every frame, we first check if the player is touching the ground, by projecting a small invisible sphere with the center at groundChecker position (which is on the player character's feet). If the sphere intersects with anything belonging to the Ground layer, then the player is grounded. Otherwise, the player is in the air and therefore not grounded. Then, we set the velocity on the y axis to zero to disable the force that was pulling the player towards the ground while he was falling.

The next step is to call the CheckInput method to take in the player's input. If the player has provided any input on the horizontal or vertical axes, this means the player character will start moving, so we call the method footstepSound from the PlayerSounds script to play some footstep audio clips. On line 12 we use the player's input to set the movementDirection vector. We use PlayerBody.forward and PlayerBody.right to make the player's input and direction be relative to the Player Character's current orientation. The final steps are move the player using the movementDirection we obtained, multiplied with the movementSpeed that we've set and once more multiplied with deltaTime, which represents the time that passed between the previous frame and current frame. We also add the gravity force to the player's Y velocity, to make the player accelerate downward and move faster towards the ground the more time he spends in the air. Then we apply this velocity to move the player downward.

Multiplying the player's input and movement by deltaTime ensures that the movement is frame-rate independent. To understand why this is important we can consider two hardware systems running the same code, with one having a frame-rate of 120fps (frames-per-second) and the other having only 30fps. This would mean that on the first system, the same line would be executed 120 times per second as opposed to only 30 times on the second system. If said line of code happens to add a certain force on an object, it would mean that the first system would apply four times more force than the second system, which results in game-play inconsistency across different platforms. To work around this, we multiply these added forces by deltaTime, which is a smaller value when the fps are higher, because less time has elapsed since the previous frame, and a larger value when the fps are lower, which balances the end result and makes it independent of the frame-rate.

5.2.4 PLAYERSOUNDS SCRIPT

PlayerSound is the script responsible for playing audio clips produced by the player character. All audio clips used in this game were acquired from sources like freesound.org or similar sites which offer free, unlicenced community-created sound clips.

```
1 public AudioSource audioSource;
2 public AudioClip[] footstepSounds;
3 public AudioClip playerHit;
4 public AudioClip[] hurtSound;
5 bool soundPlaying = false;          //this is used to make sure we don't
      overlap footstep sounds
```

Snippet 5.10: PlayerSounds fields

The first field, audioSource, is a referrence to the Audio Source component from which sounds will originate. This component is attached to the FirstPersonPlayer game object.

The field footstepSounds is an array of audio clips which stores several footstep sound samples. playerHit stores the thumping sound (similar to a punch being landed) clip that is played when the player is hit by an enemy. Additionally, a groaning sound is played, which is picked from the hurtSound audio clip array.

```
1 public void playerHitSound()
2 {
3     audioSource.PlayOneShot(playerHit);
4     if(Random.value > 0.5f)      //50% of the time a groan sound is
          played when the player is hit
5     {
6         audioSource.PlayOneShot(hurtSound[Random.Range(0, hurtSound.
              Length)]);      //play a random groan sound from the
              selection
7     }
8 }
```

Snippet 5.11: PlayerSounds's playerHitSound method

This method is called when an enemy manages to damage the player. Firstly, the playerHit sound clip is played every time the player is hit. Then, fifty percent of the time the player will also produce a groan sound, picked randomly from the hurtSound audio clip array. This adds an element of randomness to the game, because the player will not always groan when hit, and it will not always be the same groan sound.

```
public IEnumerator footstepSound()
{
    if (!soundPlaying)  //if we're not already playing a footstep sound
    {
        soundPlaying = true;
        AudioClip footstepSound = footstepSounds[Random.Range(0,
            footstepSounds.Length)];    //pick a random footstep sound
        audioSource.volume = .7f;
        audioSource.pitch = Random.Range(0.8f, 1f);     //pick a random
            pitch
        audioSource.PlayOneShot(footstepSound);
        yield return new WaitForSeconds(footstepSound.length);  //set
            soundPlaying to false after we wait for the sound to finish
            playing
        soundPlaying = false;
    }
    else            //if we're already playing a sound do nothing
    {
        yield return null;
    }
}
```

Snippet 5.12: PlayerSounds's footstepSound method

We've seen how this method is called in the PlayerMovement script when the player moves. Because this call will happen every frame in which a movement key is pressed, this may result in dozens of calls per second. We don't want to play dozens of footstep sounds per second, as the result would be a very irritating overlap of footsteps, so the first step in this method is to make sure we're not already playing a sound clip. If we are, then the method will do nothing. If not, then we pick a random footstep sound to be played from the footstepSounds audio clip array. Before we play the sound, we apply a random pitch to it, to make the clip sound sometimes higher and other times lower. This makes the footstep sounds feel less repetitive, and the random clip combined with the random pitch applied to it add diversity to the game's audio experience. After we play the footstep sound, we use the yield return command combined with the method WaitForSeconds, to make the system wait a certain time before proceeding to the next line. We want the execution of this method to wait for a time equal to footstepSound.length (which is the length in seconds of that audio clip). After this time has passed, we set soundPlaying to false, which signals the script that a new footstep sound may be played. This is how we avoid overlapping footstep sounds.

5.3 WEAPONS SYSTEM

The system that is supposed to provide the fun aspect to the game, and keep the player engaged, is the weapons system. Variety is important, so in this game the player may choose to use any of the four available weapons, each with distinct strengths and weaknesses. It is also important that the player's input is rewarded with instant feedback, in the form of animations, sounds and effects. The feedback is meant to be satisfying for the player,

because otherwise it will quickly put him off from playing the game.

There are three scripts which cover the functionality of the weapons in "Zombie Blitz":

1. The WeaponSwitching script, which activates and deactivates the game objects that represent each gun, depending on which weapon is currently equipped by the player.
2. The WeaponSway script, which adds a realistic animation that depicts the weapon swaying in space when the player moves the mouse (looks around).
3. The GunScript, the most complex of the three scripts, it implements every functionality that a particular weapon might have, like shooting, reloading, animations and sounds.

We will discuss each of the scripts in more detail in the following pages.

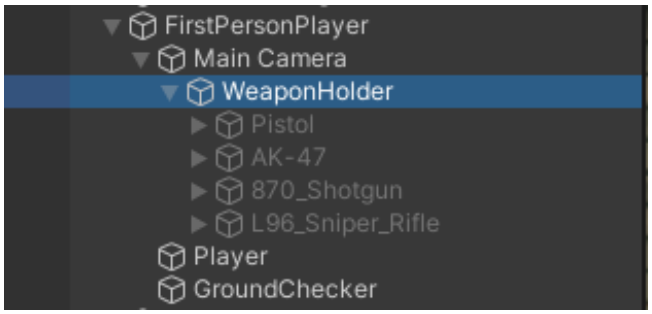

Figure 5.2: The WeaponHolder game object

In the game scene, the whole weapons system is contained within the WeaponHolder game object. This WeaponHolder has four components, an audio source, an animation controller, and the WeaponSwitching and WeaponSway scripts. The WeaponHolder has four children game objects, each representing one of the four weapons available to the player: the pistol, the AK-47, the shotgun and the sniper. At any point during gameplay, only one of these weapons are "active" meaning they exist in the game space and their components and scripts are functioning. The other three weapons are inactive, so they are invisible and their functionality is disabled.

5.3.1 WEAPONSWITCHING SCRIPT

As mentioned, this script enables the player to switch between the available weapons, by either using the number keys 1-4 on the keyboard, or using the scroll wheel on the mouse

```
private AudioSource audioSource;
public AudioClip weaponSwitch;
public weaponsEnum selectedWeapon;

void Start()
{
    audioSource = GetComponent<AudioSource>();
    selectedWeapon = weaponsEnum.PISTOL;    //initially the pistol is
        equiped
    SelectWeapon();
}
```

Snippet 5.13: WeaponSwitching's fields and Start method

The AudioSource field and weaponSwitch sound are used for playing a sound effect when the player switches weapons.

For keeping track of what weapon is currently selected, we first created an enumeration structure called weaponsEnum, which contains four constants corresponding to each of the weapons in the game. We then use the field selectedWeapon which is of type weaponsEnum for knowing which weapon is currently selected.

Inside the Start method, which is only called once when the script is initialized, we set the reference to the Audio Source component which will play the weapon switch sound. This audio source component is different from the one contained in the player-character, which all the player-related scripts reference to. This audio source component is attached to the WeaponHolder, and will be the source of every weapon-related sound, including gunshot sounds, reloading sounds, and switching sound. Also within the Start method we set the selectedWeapon to PISTOL, because when the game starts, the default equipped weapon will be the pistol. We call the SelectWeapon method, which will activate the pistol game object.

```
void SelectWeapon() {
    weaponsEnum gun = weaponsEnum.PISTOL;
    foreach (Transform weapon in transform) {
        if (gun == selectedWeapon)
        {
            weapon.gameObject.SetActive(true);
           audioSource.PlayOneShot(weaponSwitch);
        }
        else {
            weapon.gameObject.SetActive(false);
        }
            gun++;
    }
}
```

Snippet 5.14: WeaponSwitching's SelectWeapon method

The SelectWeapon method is responsible for enabling the game object corresponding to the currently enabled weapon, and disabling the other three, unequipped weapons' game objects.

This is done by first initializing a weaponsEnum variable called gun, whose initial value will be the PISTOL constant, because the Pistol is the first game object contained within the WeaponHolder. Then, for each of the children contained in the WeaponHolder game object, we check to see if we've found the desired selectedWeapon. If we've found it, we enable that game object (weapon). If not, we disable the respective game object. We increment the value of gun at every step.

If this approach seems confusing, we can imagine this iteration as a for loop, using an index and checking if the current index is equal to a value between 0 and 3, with 0 meaning the pistol, 1 the AK-47 and so on, and this value representing the weapon that is currently being equipped. When the index is equal to the mentioned value, we enable the element with that index from the list of game objects that the WeaponHolder contains. Because the order of the weapons game objects within the WeaponHolder is the same as the order in the enumeration, the correct weapon will be equipped.

```
void Update()
{
    bool weaponChanged = false;

    //switching through mouse wheel
    if (Input.GetAxis("Mouse ScrollWheel") > 0f) {
        selectedWeapon++;
        if (selectedWeapon > weaponsEnum.SNIPER) {
            selectedWeapon = weaponsEnum.PISTOL;
        }
        weaponChanged = true;
    }
    if (Input.GetAxis("Mouse ScrollWheel") < 0f)
    {
        selectedWeapon--;
        if (selectedWeapon < weaponsEnum.PISTOL)
        {
            selectedWeapon = weaponsEnum.SNIPER;
        }
        weaponChanged = true;
    }
```

Snippet 5.15: WeaponSwitching's Update method - reading input from scroll wheel

```
    //switching through numpad keys
    if (Input.GetKey(KeyCode.Alpha1) && selectedWeapon != weaponsEnum.
        PISTOL)
    {
        selectedWeapon = weaponsEnum.PISTOL;
        weaponChanged=true;
    }
    if (Input.GetKey(KeyCode.Alpha2) && selectedWeapon != weaponsEnum.
        AK47)
    {
        selectedWeapon = weaponsEnum.AK47;
        weaponChanged = true;
    }
    if (Input.GetKey(KeyCode.Alpha3) && selectedWeapon != weaponsEnum.
        SHOTGUN)
    {
        selectedWeapon = weaponsEnum.SHOTGUN;
        weaponChanged = true;
    }
    if (Input.GetKey(KeyCode.Alpha4) && selectedWeapon != weaponsEnum.
        SNIPER)
    {
        selectedWeapon = weaponsEnum.SNIPER;
        weaponChanged = true;
    }
    if (weaponChanged) {
        SelectWeapon();
    }
}
```

Snippet 5.16: WeaponSwitching's Update method - reading input from keyboard

Inside the Update method, we keep a boolean variable called weaponChanged, to know whether the currently equipped weapon has changed since the last check. Then we check for player input, which may come either in the form of mouse scroll, which results in the incrementation or decrementation of the selectedWeapon field (and manually overflowing/underflowing this value if necessary), or in the form of a key press which results in the equipping of the weapon corresponding to that key press, with the key '1' meaning the Pistol, '2' meaning the AK-47 and so on. If at any point player input is detected which will results in the changing of the currently equipped weapon, we set the value of weaponChanged to "true", and only then will the method SelectWeapon be called.

5.3.2 WEAPONSWAY SCRIPT

The WeaponSway script adds a realistic sway animation to weapons when the player moves the camera. Similar to the real-life scenario, when a when a person holding a weapon

would look around, the weapon would not remain completely rigid in his hand, instead it would tilt at different angles due to momentum.

```
public float swayAmount = 0.02f;
public float maxSwayAmount = 0.06f;
public float smoothAmount = 6f;
private Vector3 initialPosition;
```

Snippet 5.17: WeaponSway's fields

The float value swayAmount is a constant that determines how much the weapon will sway depending on the player's movement. The maxSwayAmount is an imposed limit to this sway, because otherwise a very quick mouse movement may result in an unnatural and unrealistic amount of weapon sway. The smoothAmount is a constant that will help create a smoother sway motion. The Vector3 initialPosition will represent the position of the weapon before the sway. This field is set in the Start method, to match the position of the selected weapon game object.

```
void Update()
{
    float moveX = -Input.GetAxis("Mouse X") * swayAmount;
    float moveY = -Input.GetAxis("Mouse Y") * swayAmount;
    moveX = Mathf.Clamp(moveX, -maxSwayAmount, maxSwayAmount);
    moveY = Mathf.Clamp(moveY, -maxSwayAmount, maxSwayAmount);

    Vector3 finalPosition = new Vector3(moveX, moveY, 0f);
    transform.localPosition = Vector3.Lerp(transform.localPosition,
        finalPosition + initialPosition, Time.deltaTime * smoothAmount);
}
```

Snippet 5.18: WeaponSway's Update method

In the Update method, we always check for mouse input from the player, and multiply that value by the desired swayAmount, with a larger value resulting in more sway, and vice versa. We store the results into two local variables, one for each axis. We clamp this values to make sure they are less than maxSwayAmount to avoid an unnaturally large weapon sway.

We create a new vector in three dimensions called finalPosition by combining these two values, which will represent the direction in space and amount of sway the weapon will experience.

Finally, to create a smooth movement animation, we use the Lerp method, which linearly interpolates between two points, namely the first two parameters. Essentially the weapon is moved to a point between the starting position and the end position, calculated with respect to the smoothAmount constant multiplied with Time.deltaTime to make it frame-rate independent.

5.3.3 GUNSCRIPT

The GunScript is the most important script related to the weapons system, because it implements the core elements of the game's shooter aspect: firing weapons, applying damage to enemies, keeping track of bullets, reloading, applying visual and audio effects, and animations.

```
[SerializeField] GunData gunData;   //will store information about the
    weapon
public Camera playerCam;
public Animator gunAnimator;
public GameObject muzzleEffect;
public GameObject impactEffectEnvironment;
public GameObject impactEffectZombie;
public AudioSource audioSource;

//fire-time related
private float nextFireTime = 0f;     //when the weapon can fire next
private float nextDryFireTime = 0f; //when the weapon is out of ammo, a
    dry fire sound is played, and the time between sounds is calculated
    based on dryFireDuration
```

Snippet 5.19: GunScript's fields

As we can see from the code section above, many of the fields of the GunScript class store references to different game objects or components from the game scene. GunData will point to the scriptable object that contains the information about the weapon to which the GunScript is attached. This means that each weapon will have a different GunScript component, whose fields hold different information depending on the weapon. The same is true for the other fields, with gunAnimator holding a reference to the animator of that specific weapon, muzzleEfect being a reference to one of the four available muzzle fire effects. The field impactEffectEnvironment holds the reference to the effect that is played when the player shoots any surface other than a zombie. ImpactEffectZombie will hold a different effect, played for this scenario. AudioSource will reference the AudioSource somponent attached to the WeaponHolder.

The float field nextFireTime will store a timestamp, which represents the time when the player will be able to fire his gun again. This is calculated based on the fire rate of the weapon. A different field named nextDryFireTime was needed for calculating the next time the player may attempt to fire in the scenario when his magazine is empty, and a dry fire sound is played.

```
void Update()
{
    if(PauseMenu.isGamePaused == true)
    {
        return;
    }
    if (gunData.reloading)
    {
        return;
    }
    if (Input.GetKeyDown(KeyCode.R))
    {
        audioSource.PlayOneShot(gunData.reloadSound);
        StartCoroutine(ReloadWeapon());
        return;
    }
    if (Input.GetButton("Fire1"))
    {
        if (gunData.currentAmmo > 0 && Time.time >= nextFireTime)
        {
            nextFireTime = Time.time + 1f / gunData.fireRate;
            Shoot();
        }
        else if (gunData.currentAmmo == 0 && Time.time >=
            nextDryFireTime) {    //dry fire
            nextDryFireTime = Time.time + gunData.dryFireDuration;
            audioSource.PlayOneShot(gunData.dryFireSound);
        }
    }
}
```

Snippet 5.20: GunScript's Update method

In the update method, we do nothing if the game is paused, or if the player is currently reloading his weapon. If the player is not reloading, we check to see if the reload key was pressed, and if that is the case, we play the reload sound and start the coroutine ReloadWeapon, and use the return statement to end the method call here. If the player did not press the reload key, we check to see if the fire key was pressed. If this is true, then the player is attempting to fire his weapon, and he is only able to do this if he has any bullets left in the magazine, and the current time has passed the nextFireTime. If these conditions are met, the nextFireTime is calculated by adding to the current time the value of one divided by the fireRate of the weapon (which is in bullets per second). Finally we call the Shoot method, which we will discuss in more detail next. The other scenario is when the player has attempted to fire his weapon, but his magazine is empty, in which case the dryFireSound clip is player, but only if the current time has passed the nextDryFireTime, which is simply equal to the length of the dryFireSound clip. This ensures that we don't get multiple overlapping dry fire sounds.

```
void Shoot()
{
    audioSource.PlayOneShot(gunData.shootingSound);      //fire sound
    gunAnimator.SetTrigger("RECOIL");                    //add recoil
    muzzleEffect.GetComponent<ParticleSystem>().Play(); //add muzzle
        flash
    gunData.currentAmmo--;
    HandleShot();

}
```

Snippet 5.21: GunScript's Shoot method

This method is called when the player successfully fires his weapon. The method will play the weapon's shooting sound, activate the gun animator's "RECOIL" field which triggers the weapon's recoil animation, and then play the muzzle flash effect which simulates the fire that exits a real gun's barrel when fired. Lastly, we decrease the number of bullets the weapon holds by one, and call the HandleShot method which is responsible for checking if the bullet hits anything, and apply damage accordingly.

```
void HandleShot()
{
    RaycastHit hit;
    if (Physics.Raycast(playerCam.transform.position, playerCam.
        transform.forward, out hit, gunData.range))
    {
        ZombieController targetScript = hit.transform.
            GetComponentInParent<ZombieController>();
        if (targetScript != null)
        {
            if (hit.transform.gameObject.tag.Equals("Head"))
            {
                targetScript.TakeDamage(gunData.damage * 2f);
                Debug.Log(Time.time + " " + hit.transform.name + " " +
                    gunData.damage * 2f);
            }
            else if (hit.transform.gameObject.tag.Equals("Body"))
            {
                targetScript.TakeDamage(gunData.damage);    //if hit
                    target, apply damage
                Debug.Log(Time.time + " " + hit.transform.name + " " +
                    gunData.damage);
            }
            else if (hit.transform.gameObject.tag.Equals("Legs"))
            {
                targetScript.TakeDamage(gunData.damage * 0.66f);
                Debug.Log(Time.time + " " + hit.transform.name + " " +
                    gunData.damage*0.66f);
            }
            else if (hit.transform.gameObject.tag.Equals("Arms"))
            {
                targetScript.TakeDamage(gunData.damage * 0.66f);
                Debug.Log(Time.time + " " + hit.transform.name + " " +
                    gunData.damage * 0.75f);
            }
        }
        HandleImpactEffects(hit);
    }
}
```

Snippet 5.22: GunScript's HandleShot method

The HandleShot method will compute the bullet's trajectory, check to see if this trajectory intersects with any object, and apply damage if the object intersected is an enemy. It also plays the impact effect depending on the surface it hits.

To calculate the bullet's trajectory, we use the method Raycast, which will draw and invisible line in space, of length equal to GunData.range, starting from the player's eye and directed in his forward direction. The method will return false if the line does not intersect with anything, and in this case HandleShot will do nothing. If the ray cast intersects with

any object, information about this collision like the name of the object, the coordinates of the impact point, its distance from the ray's origin, is stored in the hit variable of type RaycastHit, which is passed as an out out parameter to the Raycast method.

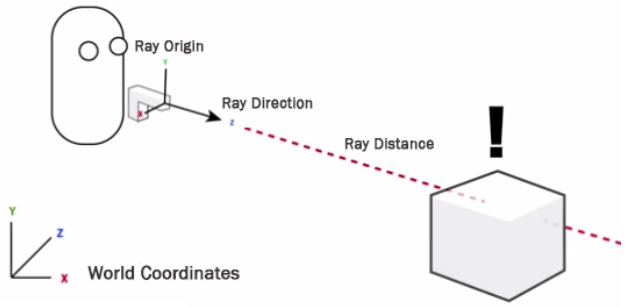

Figure 5.3: Raycast illustrated

The next step is to check if the game object we've hit has a parent that contains a "ZombieController" script. This denotes that we've hit an enemy. We check the parents because the bullet will hit a game object that is only one body part, like an arm or a torso, and this game object is contained within the actual zombie game object, which is the one that holds the ZombieController script. If we did indeed hit a game object belonging to a zombie, we check the object's tag to know which body part has been hit, and apply the damage accordingly. Lastly, we call the method HandleImpactEffects to play the correct impact effect depending on the surface that was hit.

```
void HandleImpactEffects(RaycastHit hit) {
    if (hit.transform.gameObject.layer != 6)    //any layer other than
        the player character layer
    {
        if(hit.transform.gameObject.layer == 7)         //impact effect
            for zombie
        {
            GameObject impactPoint = Instantiate(impactEffectZombie, hit
                .point, Quaternion.LookRotation(hit.normal)); //add
                bullet hole
            Destroy(impactPoint, 1f);       //destroy bullet hole
        }
        else            //impact effect for environment
        {
            GameObject impactPoint = Instantiate(impactEffectEnvironment
                , hit.point, Quaternion.LookRotation(hit.normal)); //add
                bullet hole
            Destroy(impactPoint, 3f);       //destroy bullet hole
        }
    }
}
```

Snippet 5.23: GunScript's HandleImpactEffects method

In this method, we use the information stored within the hit variable of the HandleShot method to check the layer of the object that was hit by the ray cast. If the layer's code is 7, which is the Enemy layer, we instantiate an impactEffectZombie effect at the location where the bullet hit. We call Destroy with the parameter 1f to make the engine destroy (remove) the impact effect's game object after one second. We have the same approach in the case where the layer hit represents the environment layer, only this time we create an effect that represents a bullet hole on the surface and some particles being launched from it to simulate a real life bullet impact.

```
IEnumerator ReloadWeapon()
{
    gunData.reloading = true;
    gunAnimator.SetBool("RELOADING", true);

    yield return new WaitForSeconds(gunData.reloadTime);
    //following lines will be executed only after reloadTime seconds
    gunAnimator.SetBool("RELOADING", false);
    gunData.currentAmmo = gunData.magazineSize;
    gunData.reloading = false;
}
```

Snippet 5.24: GunScript's ReloadWeapon coroutine

As we've seen in the Update method, ReloadWeapon is called when the player presses the reload key. This starts the coroutine, which sets the reloading field of the gunData scriptable object to true, which as we've seen leads to the Update method to return without doing anything, and set the boolean parameter "RELOADING", from the gun animator to true, which will trigger the reloading animation for that weapon. Next we use the yield return statement to pause the execution of this method for gunData.reloadTime seconds. After this time has passed, we set the RELOADING parameter back to false, refill the gun's ammunition, and set reloading to false.

5.3.4 ANIMATIONS AND EFFECTS

We've covered the scripts that govern the functionality of the whole weapons system, and now we will have a look at the animators and effects attached to these weapons to better understand how these visual effects are obtained.

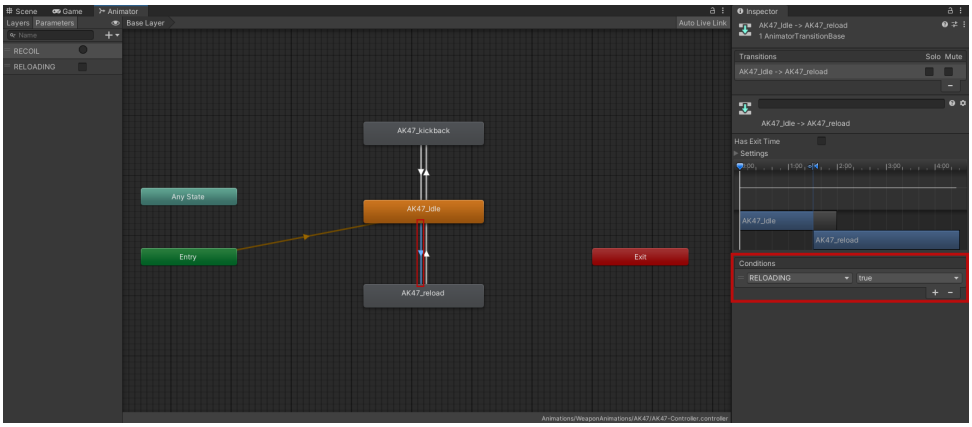

Figure 5.4: AK-47 animator

An animator is similar to a state machine, with each animation clip representing a state, and transitions between states being performed depending on conditions about the parameters of the animator. In the figure above, we can see what the animator of the AK-47 looks like, and this model is used by the other weapons as well, only with their own animation clips instead. As we can see, there are three states, each representing an animation clip:

1. AK-47_Idle is the default state, and the animation depicts the player's weapon being slowly raised and lowered to simulate the player's breathing.

2. AK-47_kickback is the animation clip being played when the player shoots his weapon. It simulates the kickback force experienced in real life when a gun is fired. This animation is played when the RECOIL parameter is triggered in the GunData script's Shoot method.

3. AK-47_reload is played when the player presses the reload key. The animation that ensues imitates what the process of reloading that weapon in real life would look like.

In the figure we can also see directed lines going from one state to another. These lines represent the transitions, which dictate how the change from one animation clip to another is performed. On the right side we can see the conditions that cause the transition to be initiated. In the case of the transition from the idle animation to the reload animation, we can see the condition is for the RELOADING parameter of the animator to be set to true. The opposite is true for the transition in the other direction, from the reload animation to the idle animation, where the RELOADING parameter needs to be set to false.

When talking about the gun animations, it is worth mentioning that I have recorded each animation clip for each weapon myself.

Figure 5.5: AK-47 muzzle flash effect

The muzzle effect that is played when the player fires his gun is realized by the MuzzleEffect game object which is placed at the tip of the gun's barrel, and contains a ParticleSystem component. This ParticleSystem is what creates the visual effect we see when the gun is fired, and this happens when the component's Play method is called from the Shoot method of the GunScript. This ParticleSystem was imported for this project and was not created by me.

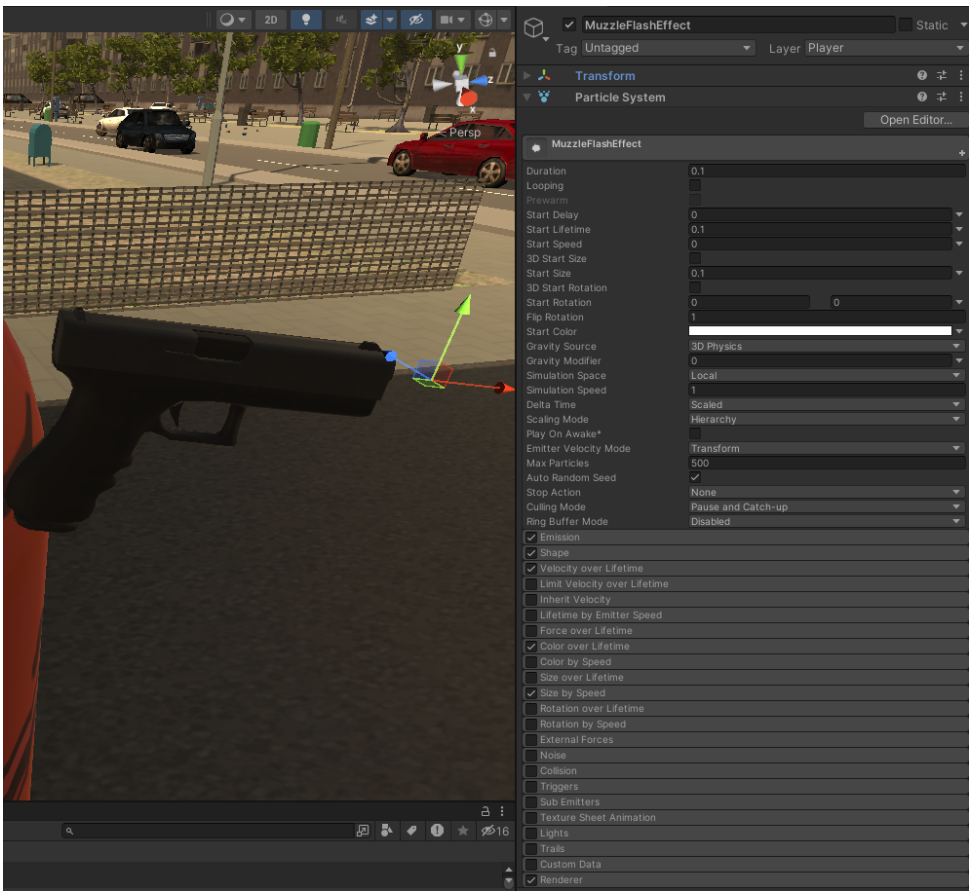

Figure 5.6: MuzzleFlash game object

Bullet impact effects are achieved in the same way, with the only difference being that the game objects that contain the particle system which generates the effects are not

existing in the game scene at runtime, instead they are created from a prefab from the project files, and deleted after a few seconds.

5.4 ENEMIES SYSTEM

The enemies are what provide the conflict and challenge in "Zombie Blitz". Without the danger they create for the player, the player would have no objective, no purpose except for running around the game map and firing his weapons at random objects. When you introduce an element of danger, you give the player a purpose within the game (survive the attack), and create the fun aspect of the game (survive using the weapons that are available). This is the interaction that is at the core of the gameplay loop of "Zombie Blitz".

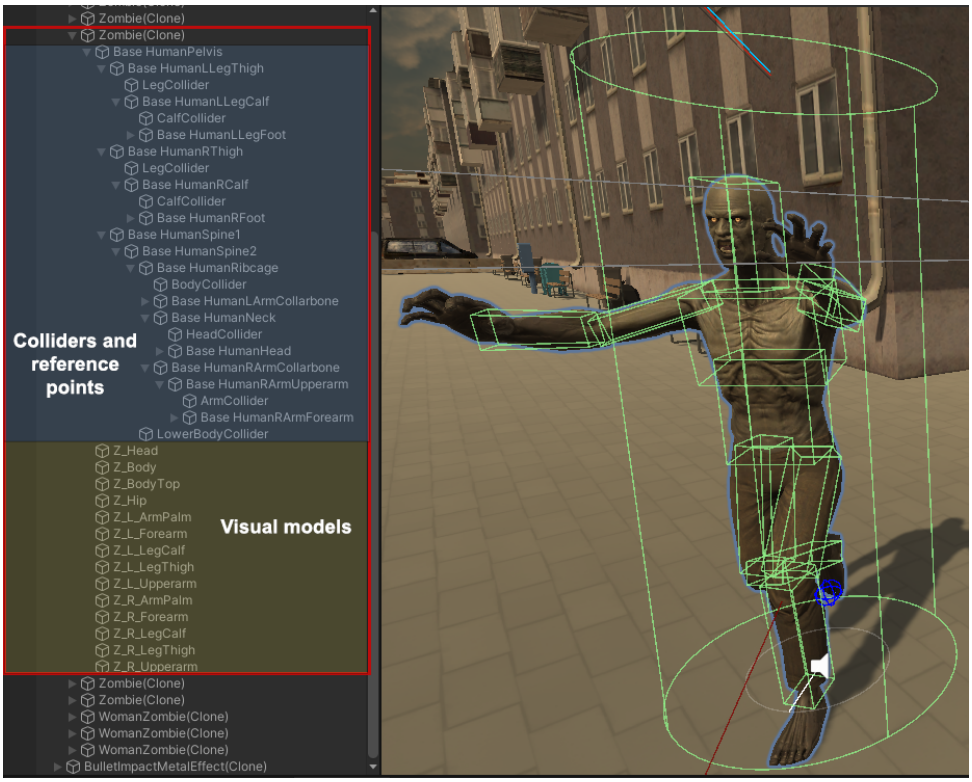

Figure 5.7: Zombie game object

As we can see in figure 5.7, a zombie game object is made up of a multitude of game objects, serving two main purposes. In the upper section, highlighted in blue are the zombie's body reference points. This are simple empty game objects which are placed and tied together at different coordinates to make up the key points of a humanoid body. For example, the main reference point, relative to which all other reference points are created, is the pelvis. From the pelvis, the spine reference point might be attached, perhaps placed higher on the Y axis. Then, attached to the spine is the ribcage, and so on. It is important to mention that these components are tied to each other hierarchically, so for example if the ribcage is moved, then the elements attached to it like the arms and head will move, but the spine will not because it is the parent of the ribcage. These reference points simulate the skeleton, and to them we attach the visual models, highlighted in yellow. These represent individual body parts, like the forearm or the head, and are simple visual meshes that are

attached to the reference points. These visual meshes have been imported and attached to the reference points by the Unity Editor.
This was the state in which the zombie models were imported, however, this would not allow for physics collisions, and the zombies would be a purely visual element, and thus the player would not be able to shoot the zombies or interact with them. Therefore, I had to create the colliders myself, which we can see as the green cuboid shapes that form up the zombie's body. These are also attached to the reference points.

There are two scripts that implement the enemies' functionality:

1. ZombieController: this script is the zombie's artificial intelligence, and initiates its every movement and action.

2. ZombieSounds: this script governs the logic behind the zombie's sound clip generation.

5.4.1 ZOMBIECONTROLLER SCRIPT

An independent instance of this script is attached to every zombie in the game scene. As we will see, this script guide's the zombie's behaviour and also stores information about the zombie.

```
public EnemyData enemyData;
NavMeshAgent agent;
Animator animator;
Transform target;
float health;
float distanceToPlayer;
bool targetInRange;
bool attackStarted;
float attackEndTime;
bool isAlive;
```

Snippet 5.25: ZombieController's fields

The script will store the enemyData scriptable object that corresponds to the zombie type that this script instance is attached to. It also stores references to the NavMeshAgent, Animator and Transform components. The field health represents the current health of the zombie, and distanceToPlayer is the distance in meters to the player character. There are two boolean values, targetInRange indicating whether the player is close enough to initiate an attack, and attackStarted, which is true if the zombie has initiated an attack. The float value attackEndTime will store the time stamp when the current attack will be completed, and the player will be dealt an amount of damage if he is still in range at that time. Another boolean value isAlive shows whether the zombie is still alive in the game scene.

```
void Start()
{
    animator = GetComponent<Animator>();
    agent = GetComponent<NavMeshAgent>();
    target = PlayerManager.instance.player.transform;
    health = enemyData.health;
    isAlive = true;
    setNavMeshStats();
}
void setNavMeshStats()
{
    agent.speed = enemyData.speed;
    agent.angularSpeed = enemyData.angularSpeed;
    agent.acceleration = enemyData.acceleration;
    agent.stoppingDistance = enemyData.stoppingDistance;
}
```

Snippet 5.26: ZombieController's Start method

In the Start method, we set the appropriate references to the desired components, set the current health to be equal to the starting health as dictated by the enemyData scriptable object, set isAlive to true and set the NavMesh component's stats to match the stats from the scriptable object. This makes the zombie have the speed, acceleration and range that is set in zombieData.

```
void Update()
{
    BehaviourAlwaysChase();
}
void BehaviourAlwaysChase()     //zombies always chase the player
{
if (health > 0)
    {
        distanceToPlayer = Vector3.Distance(transform.position, target.
            position);
        targetInRange = (distanceToPlayer <= enemyData.attackRange);
        if (!targetInRange)
        {
            ChasePlayer();
        }
        else
        {
            Attack();
        }
    }
}
```

Snippet 5.27: ZombieController's Update method

In the Update method, the method that defines the zombie's behaviour is called. In this version of the game, the zombies always chase the player, as given by the BehaviourAlwaysChase method. There is also a method called BehaviourChaseWhenPlayerInSight, which will make the zombies only chase the player when he is within a certain range from them, and stand idle otherwise. This behaviour is not being used in the final game. In the BehaviourAlwaysChase method, we only do something when the zombie's health is greater than 0. We check the distance between the zombie and the player, and if this distance is smaller than the zombie's attack range, then that means that the player can be attacked, and the variable targetInRange is set to true. If the target is in range the method Attack will be called, and if not, the method ChasePlayer will be called

```
1 void ChasePlayer()
2 {
3     agent.SetDestination(target.position);
4     animator.SetBool("CHASING", true);
5     animator.SetBool("ATTACKING", false);
6     StartCoroutine(GetComponent<ZombieSounds>().footstepSound());
7     attackStarted = false;
8 }
```

Snippet 5.28: ZombieController's ChasePlayer method

In the ChasePlayer method, we set the NavMesh agent's destination to be the location of the player character. The NavMesh Agent component will then compute the best route towards the destination by taking into acount the game map's NavMesh that was baked (precomputed) using the scene editor. Thus, the zombie game object will start to move towards the player at the speed and acceleration set in the Start method. We set the CHASING parameter of the animator to true, and set the ATTACKING parameter to false to insure that any attacking animation is canceled and the running animation is played. We start a coroutine of the footstepSound method from the ZombieSounds script that is attached to the zombie game object. This will make the zombie generate footstep sounds when chasing after the player. We set the attackStarted boolean field to false.

```
void Attack()
{
    if (attackStarted == false)
    {
        attackStarted = true;
        animator.SetBool("ATTACKING", true);
        attackEndTime = Time.time + enemyData.attackDuration;
    }
    else
    {
        if (Time.time > attackEndTime)
        {
            target.GetComponent<PlayerData>().takeDamage(enemyData.
                damage);
            attackStarted=false;
        }
    }
    FaceTarget();
}
void FaceTarget()
{
    if(health > 0)       //won't face target while dead
    {
        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 * 5f);
    }
}
```

Snippet 5.29: ZombieController's Attack method

As we've seen, the attack method is called when the player is within attacking range of the zombie. We first check whether an attack is already started, and if it is not, then we start it now by setting attackStarted to true and setting the ATTACKING parameter of the animator to true. We compute the attackEndTime by adding the attackDuration of this zombie type to the current time.

If an attack has already been initiated, we check if the current time has passed the attackEndTime, and if this is true then the attack was successful and we call the PlayerData script's takeDamage method with the enemyData.damage parameter to deal the correct amount of damage to the player. If at any point during an attack the player manages to distance himself from the zombie enough to exit his attack range, this will cause the ChasePlayer method to be called before the current time passes the attackEndTime, and thus the player will escape the attack before he is hit and dealt damage. The last step of the Attack method is to call FaceTarget, which will make the zombie always face towards the player while attacking. This is done by generating a 3D vector that represents the direction towards the

player, and make the zombie gradually turn in that direction using the Slerp method.

```
public void TakeDamage(float damage)
{
    health -= damage;
    gameObject.GetComponent<ZombieSounds>().playZombieHitSound();
    if (health <= 0)
    {
        if (isAlive)    //the zombie is only deleted a few seconds after
             health reaches 0, time in which the player can still shoot
            the zombie, so we need isAlive to make sure we only call
            Death() once
        {
            Death();
            isAlive = false;
        }
    }
}
void Death()
{
    agent.isStopped = true;
    /*BoxCollider[] colliders = gameObject.GetComponentsInChildren<
        BoxCollider>();
    foreach (BoxCollider collider in colliders)     //disable collider
        so that player can't shoot the zombie while dead. Optional
    {
        collider.enabled = false;
    }*/
    if (Random.value > 0.5f)    //randomly fall forward or backwards
        animator.SetTrigger("DEATHFORWARD");
    else
        animator.SetTrigger("DEATHBACKWARD");
    Destroy(gameObject, 6f);    //destroy the zombie after 6 seconds.
    target.GetComponent<PlayerData>().increaseZombieKilled();
}
```

Snippet 5.30: ZombieController's TakeDamage and Death methods

The method TakeDamage is called when the player manages to hit a zombie, as we've seen in the GunScript's HandleShot method. The health field is decreased by the value of the damage parameter, and the method playZombieHitSound is called from the ZombieSounds script. If the new health is less than or equal to zero, this means the zombie will die, but the Death method (which will delete the game object) is only called if isAlive was previously set to true. This check is required because the Death method will only delete the zombie game object after the death animation is played, which means after a few seconds. This would mean that without the mentioned check, if the player were to again shoot the now dead zombie, the Death method would be called again, which is not the desired functionality. Thus, we use isAlive to insure that the Death method is called only once no matter what.

In the Death method, we first disable the zombie's movement. There is a commented piece of code which when uncommented would delete the zombie's colliders, and so the player would not be able to still shoot a dead zombie. This feature is disabled because I considered it is more fun this way. The next step is to randomly play one of two available animations, one in which the zombie falls forward, and one in which he falls backwards. This adds an element of diversity and unpredictability to the gameplay. The last steps are to remove the zombie from the game after 6 seconds, and increment the player's zombiesKilled stat.

5.4.2 ZOMBIESOUNDS SCRIPT

The ZombieSounds scripts is responsible for playing the sounds produced by the zombies during the gameplay. There are three sound categories: footstep sounds, growl sounds, and sounds that are produced when a bullet hits a zombie.

```
1 AudioSource audioSource;
2 public AudioClip[] zombieHitSound;
3 public AudioClip[] zombieFootstepSound;
4 public AudioClip[] growlSound;
5 public float timeBetweenGrowls;
6 public float growlProbability;
7 float timer;        //used to count the time between growls
8 bool footstepSoundPlaying = false;      // to make sure we don't overlap
      footstep sounds
```

Snippet 5.31: ZombieSounds's fields

The class contains a reference to the AudioSource component of the zombie that the script is attached to. There are three AudioClip arrays, one for each of the sound categories that were mentioned above. The arrays store a selection of sound clips that represent different variations of that sound. The float field timeBetweenGrowls is used for controlling how often the zombies will produce a growl sound. The float growlProbability is used to enable an element of randomness when the zombies growl. The float timer is used to count the time between growls, and the boolean footstepSoundPlaying will be true is a footstep sound is currently being played, and false otherwise.

```
void Update()
{
    timer += Time.deltaTime;
    if(timer >= timeBetweenGrowls)
    {
        timer -= timeBetweenGrowls;
        if(Random.value < growlProbability)      // to make zombie growls
            more random
            playGrowlSound();
    }
}
void playGrowlSound()
{
    AudioClip growl = growlSound[Random.Range(0, growlSound.Length)];
        //pick a random growl sound
    audioSource.pitch = Random.Range(0.7f, 1f);      //pick a random
        pitch
    audioSource.PlayOneShot(growl);
}
```

Snippet 5.32: ZombieSounds's Update and playGrowlSound methods

In the Update method, we always increment the timer, and if the timer has passed value of timeBetweenGrowls, then we first reset the timer, and a random value is generated. If the random value is smaller than the growlProbability field, then the method playGrowlSound is called. This check is done to make sure the zombie does not always produce a growl sound at set time intervals, as this would make for an annoying and repetitive experience. The growlProbability adds an element of randomness and realism to the process.

In the playGrowlSound method, we pick a random growl clip from the growlSound array, apply a random pitch to the clip to add further randomness to the process, and finally we play the growl sound.

```
public void playZombieHitSound()
{
    audioSource.PlayOneShot(zombieHitSound[Random.Range(0,
        zombieHitSound.Length)]);        //play a random zombie hit sound
         from the selection
}
public IEnumerator footstepSound()
{
    if (!footstepSoundPlaying)      //if we are not already playing
        footstep sound
    {
        footstepSoundPlaying = true;
        AudioClip footstepSound = zombieFootstepSound[Random.Range(0,
            zombieFootstepSound.Length)];
        audioSource.volume = .5f;
        audioSource.pitch = Random.Range(0.8f, 1f);
        audioSource.PlayOneShot(footstepSound);
        yield return new WaitForSeconds(footstepSound.length);      //
            wait for the footstep sound to finish playing then set
            footstepSoundPlaying to false
        footstepSoundPlaying = false;
    }
    else        //if we are already playing a footstep sound do nothing
    {
        yield return null;
    }
}
```

Snippet 5.33: ZombieSounds's playZombieHitSound and footstepSound methods

As we've seen in the ZombieController's takeDamage method, the playZombieHitSound method is called when a bullet hits the zombie. This method simply picks a random audio clip from the zombieHitSound array and plays it.

The footstepSound method which is called by the ChasePlayer method in the ZombieController script. If a footstep sound is already being played as indicated by the footstepSoundPlaying field, then the method will do nothing. Otherwise, it will first set footstepSoundPlaying to true, and pick a random footstep audio clip from the zombieFootstepSound array, lower it's volume, apply a random pitch to the clip and play it. The yield return statement is used to pause the execution of the method for a time equal to the length of the footstep sound clip. After this time has passed, the footstep sound will have finished playing, and we can safely set footstepSoundPlaying to false to allow for a new footstep sound to be played.

5.4.3 ZOMBIE ANIMATOR

The zombie animator contains five states, with each representing a particular animation

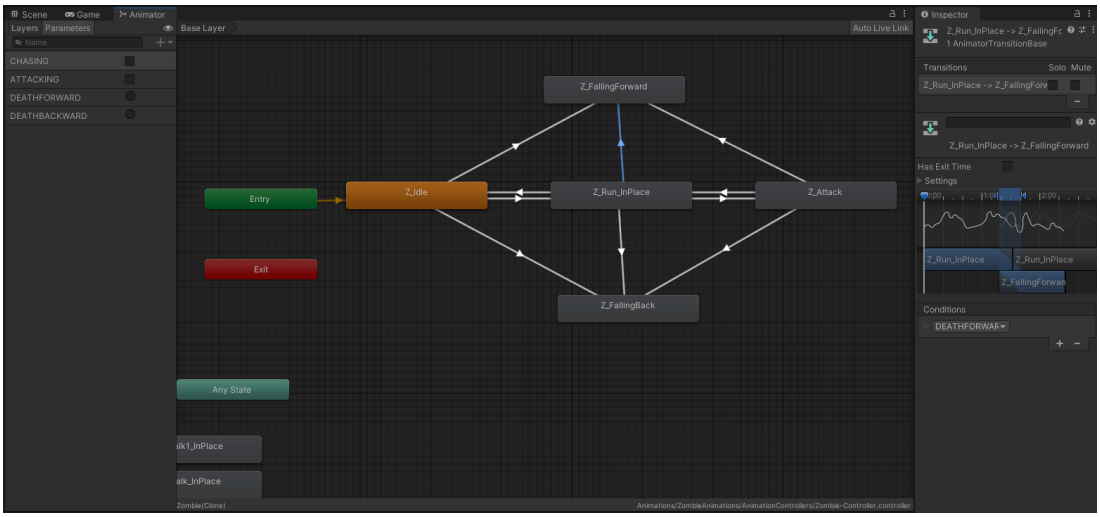

Figure 5.8: Zombie controller

clip. The first state is the idle state, in which the zombie does not move. This state will only be active for a brief moment when the zombie is first loaded into the scene, as immediately afterwards the ChasePlayer method will be called which results in the transition to the running animation, called Z_Run_InPLace in the animator.

The transition to the attack animation is performed when the boolean parameter ATTACKING is set to true. From either the running animation or the attack animation, a transition is possible to the falling forward or falling backwards animation clips, depending on the activation of the triggers DEATHFORWARD or DEATHBACKWARD respectively. This activation is performed randomly in the Death method of the ZombieController script, as discussed.

It is important to mention that unlike the animation clips for the weapons, which I recorded myself, all zombie animation clips were not created by me and have been imported from the site Mixamo.

5.5 WAVE SPAWNER

The wave spawner is an important gameplay element because it is a way to control the difficulty of the game by simulating arcade games' "levels". A wave will contain a specific number of enemies of each type, which will create the experience that I as a developer intended. If enemies would simply spawn continuously, there would be no way to control this experience. Additionally, the small time intervals between waves allows the player a moment of breather, during which he can reload his weapons, and relocate to a better position on the game map, to prepare for the incoming wave. Without this small breaks, the experience would quickly become tiring and bothersome.

The wave system is implemented inside the script WaveSpawner, with the help of the Wave class and the ZombieCounter script.

5.5.1 ZOMBIECOUNTER SCRIPT

The ZombieCounter script is both used by the WaveSpawner script as well as the UI related script ZombiesLeft. ZombieCounter will perform a count of the number of zombies left in the scene every 0.3 seconds. It contains two fields, zombiesLeft, which is an integer storing the number of zombies left in the scene at the time of the last performed count, and completedCount, a boolean value that will tell us whether a count has been completed in the last 0.3 seconds.

```
IEnumerator CountZombies()      //count the zombies every .3 seconds
{
    int count = 0;
    ZombieController[] zombieControllers = GetComponentsInChildren<
        ZombieController>();  //get only objects which have a zombie
        controller script
    foreach (ZombieController zombieController in zombieControllers)
    {
        if (zombieController.GetHealth() > 0)   //zombie game objects
            still exist a few seconds after reaching 0 health, to allow
            for the death animation to play.
        {
            count++;                            //thus we only count
                zombies who have more than 0 health as alive
        }
    }
    zombiesLeft = count;
    completedCount = true;
    yield return new WaitForSeconds(.3f);       //will only count again
        after .3 seconds, as performing the count every frame would be
        unoptimal.
    completedCount = false;
}
```

Snippet 5.34: ZombieCounter's CountZombies method

The method CountZombies will be called as a coroutine. We first create an array which stores all game objects which contain a zombieController script. We search for these objects among the children of the game object that runs the ZombieCounter script, which is simply called Zombies, and contains all the zombie game objects in the game. Once we've created the array, we iterate through it and count only the zombies that have a positive number of health points. This is necessary because as we've seen in the zombieController script, zombie game objects will still exist in the scene for a few seconds after their health has reached zero. After the count is complete, we update the zombiesLeft field and set completedCount to true. We wait for 0.3 seconds before setting completedCount back to false to allow for the next count to begin.

```
void Update()
{
    if (completedCount)
        return;
    else
    {
        StartCoroutine(CountZombies());
        return;
    }
}
```

Snippet 5.35: ZombieCounter's Update method

As we know, the Update method is called once every in-game frame, but performing this count would be an excessive amount of redundant calculation, which is why we used the boolean completedCount to make sure we only perform the count every 0.3 seconds, which leads to a much more reasonable number of counts performed while still keeping the feedback immediate. If completedCount is true, the Update method will do nothing, and if not, this means that more than 0.3 seconds have passed since the last count, and we should call CountZombies again.

5.5.2 WAVE CLASS

The wave class is a data structure that will store information about a particular wave.

```
public class Wave
{
    public string waveName;
    public Transform[] zombieTypes;
    public int[] numberOfZombies;
    public float spawnRate;
}
```

Snippet 5.36: The Wave class

We use the string waveName to store a name for the wave, which creates an easier workflow from the editor, as we can simply use names like "Wave1", "Wave2", etc. to know the correct order in which to add the waves from the scene editor. The array zombieTypes will simply store the types of zombies that will be present in this wave. Each entry will store a reference to the zombie prefab (prefabricated object) that we wish to include in this wave. The integer array numberOfZombies will store the number of zombies that will appear for each zombie type in this wave. The length of numberOfZombies will always be equal to the length of zombieTypes, and an element from a certain position inside the numberOfZombies array will indicate the number of zombies of the type from the same position in the zombieTypes array. The last field is spawnRate, which represents the time in seconds that will pass between two

zombies being spawned.

5.5.3 WAVESPAWNER SCRIPT

The WaveSpawner works as a state machine, always finding itself in one of three states: COUNTDOWN, SPAWNING, and WAITING.

```
public enum WaveSpawnerState
{
    COUNTDOWN,      //counting down until the start of the next wave
    SPAWNING,       //spawning enemies
    WAITING         //waiting for the end of the current wave
};
public Wave[] waves;
public Transform[] spawnPoints;
public GameObject allZombies;   //used for counting the number of
    remaining zombies and is also a parent object to all the zombies that
     will spawn
int nextWave = 0;
public float timeBetweenWaves;
public float timeUntilNextWave;
WaveSpawnerState state=WaveSpawnerState.COUNTDOWN;
```

Snippet 5.37: WaveSpawner's fields

When the COUNTDOWN state is active, the script will wait a given number of seconds before proceeding to spawn enemies. This waiting time is the timeframe between waves that we've discussed earlier, which allows the player a moment to recompose himself and prepare for the next attack. When the countdown is complete, the script enters the SPAWNING state, in which zombies are spawned. When the spawner has finished generating the enemies, it enters the WAITING state. In this state, the script simply waits for the player to shoot down all the zombies in the scene, before the next wave begins. When all zombies have died, the COUNTDOWN state is activated again and the cycle restarts.

The script keeps an array of waves of the Wave class, which is populated from the editor. The array spawnPoints stores all the designated locations within the game map where zombies may spawn. The field allZombies is simply a reference to the Zombies game object. As we've mentioned in the WaveCounter script, the Zombies game object will be the parent of all zombies in the scene, which is why we need the reference.

The field nextWave stores the array index of the wave that will be started next. The float timeBetweenWaves will represent the time in seconds that will ellapse during the COUNTDOWN state, and timeUntilNextWave is how much of that time remains. Lastly, the field state is used to keep track of the currently active state, and it is set to COUNTDOWN when the script is initiated.

```
void Update()
{
    if(state == WaveSpawnerState.WAITING)
    {
        if(AreEnemiesAlive())
        {
            return;     //wait for the player to kill all zombies before
                starting next wave
        }
        else
        {
            WaveCompleted();
        }
    }
    if (timeUntilNextWave <= 0)
    {
        if(state != WaveSpawnerState.SPAWNING)
        {
            StartCoroutine(SpawnWave(waves[nextWave]));
        }
    }
    else
    {
        timeUntilNextWave -= Time.deltaTime;
    }
}
```

Snippet 5.38: WaveSpawner's Update method

In the Update method, if the script is currently in the WAITING state, that means we are waiting for the player to kill all zombies in the scene. We call the method AreEnemiesAlive which returns true if there are zombies with positive health points in the game scene, and false otherwise. If the method returns true, we do nothing because the player hasn't finished defeating the current wave. If there are no zombies left alive, we call the method WaveCompleted, which sets the current state to COUNTDOWN and prepares the start of the next wave.

If the timeUntilNextWave is less than or equal to zero, that means the current countdown has finished and we can enter the SPAWNING state (if this state is not already active). We start the coroutine SpawnWave with the next wave from the waves array as a parameter. If timeUntilNextWave is larger than zero, then we simply update the timeUntilNextWave.

```
void WaveCompleted() {
    Debug.Log("Wave completed!");
    nextWave++;
    state = WaveSpawnerState.COUNTDOWN;
    timeUntilNextWave = timeBetweenWaves;
    if (nextWave > waves.Length - 1)
    {
        StartCoroutine(FindAnyObjectByType<GameManager>().GameWon());
    }
}
```

Snippet 5.39: WaveSpawner's WaveCompleted method

In the WaveCompleted method, we increment the value of nextWave, set the current state to COUNTDOWN, and set the timeUntilNextWave to be equal to the desired timeBetweenWaves. Lastly, if the next wave number is bigger than the number of waves inside the array, this means that the player has beaten all the waves, and the GameWon method is called to change the scene to the GameWon scene.

```
IEnumerator SpawnWave(Wave wave)
{
    Debug.Log("Starting wave " + (nextWave+1));
    state = WaveSpawnerState.SPAWNING;
    for(int i=0; i<wave.zombieTypes.Length;i++)      //for each zombie
        type
    {
        for (int j = 0; j < wave.numberOfZombies[i]; j++)       //spawn
            the corresponding zombie
        {
            SpawnZombie(wave.zombieTypes[i]);
            yield return new WaitForSeconds(1f / wave.spawnRate);
        }
    }
    state = WaveSpawnerState.WAITING;
    yield break;
}
void SpawnZombie(Transform zombie)
{
    Debug.Log("Spawing zombie...");
    Transform spawnpoint = spawnPoints[Random.Range(0,spawnPoints.Length
        )];
    Instantiate(zombie, spawnpoint.transform.position,spawnpoint.
        transform.rotation,allZombies.transform);
}
```

Snippet 5.40: WaveSpawner's SpawnWave and SpawnZombie methods

As we've seen, the SpawnWave method is called when the timeUntilNextWave

reaches zero. The fist step is to set the active state to SPAWNING. Then we enter a nested for loop, with the outer loop iterating through the zombieTypes array of the wave parameter, and the inner loop generating as many zombies as indicated by the numberOfZombies array for that zombie type. We spawn the zombie using the method SpawnZombie, and passing as a parameter the desired zombie type, and then waiting for an amount of time as per the spawnRate field of the wave.

After all zombies have been spawned, the current state is set to WAITING, to make the script wait for the player to kill all enemies before starting the next wave.

In the SpawnZombie method, we pick a random spawn location from the spawnPoints array, and instantiate the zombie at that location, as a child of the allZombies game object.

5.6 USER INTERFACE AND MENUS

A good user interface is important because it impacts the player's experience and overall enjoyment. In a first-person shooter (where the UI is also called Heads-Up Display, or HUD) , the player needs to have easy access to vital information such as health and ammo count, all the while not being obstructed or impaired by the display of this information.

5.6.1 HUD

Figure 5.9: Screenshot of the Heads-Up Display

As we can see in figure 5.9, the Heads-Up Display contains six main elements:

1. The Health bar, which shows a visual representation of the amount of health points the player has left.

2. The amount of ammunition that is currently loaded in the gun, out of the total amount

possible from a full magazine.

3. The current wave display.
4. The number of zombies left alive in the current wave.
5. The total number of zombies that the player has killed over the course of this current game.
6. The crosshair, which indicates the direction where the player is aiming.

Each of these elements is implemented through one script (sometimes two), and because their implementation is similar for some, or trivial for others, I will only discuss in detail one of these scripts, namely the BulletsText script, to avoid repeating the same information.

5.6.2 BULLETSTEXT SCRIPT

This purpose of this script is to display to the player the number of bullets loaded inside his currently equipped weapon.

```
public TextMeshProUGUI bulletsText;
public GameObject weaponHolder;
public weaponsEnum weapon;  //currently equiped weapon

//data about each gun
GunData pistolData;
GunData AKData;
GunData ShotgunData;
GunData SniperData;
void Start()
{
    getGunsData();
}
```

Snippet 5.41: BulletText's fields and Start method

The first field, called bulletsText of type TextMeshProGUI keeps a reference to the Text component of the game object which represents the available ammunition UI element. The weaponHolder script will store a reference to the WeaponHolder game object, which as we've discussed is where all four guns' game objects are contained. The field weapon will simply store the name of the currently equipped weapon, whose bullets will be displayed. The last four fields are references to the four scriptable objects that store information about each gun. This is how the script will be able to know the number of remaining bullets and the size of the magazine.

In the Start method we call the method getGunsData which simply makes each GunData field reference the correct weapon.

```
void Update()
{
    int currentAmmo;
    int magazineSize;
    weapon = weaponHolder.GetComponent<WeaponSwitching>().
        getSelectedWeapon();
    switch (weapon)
    {
        case weaponsEnum.PISTOL:
            currentAmmo = pistolData.currentAmmo;
            magazineSize = pistolData.magazineSize;
            break;
        case weaponsEnum.AK47:
            currentAmmo = AKData.currentAmmo;
            magazineSize = AKData.magazineSize;
            break;
        case weaponsEnum.SHOTGUN:
            currentAmmo = ShotgunData.currentAmmo;
            magazineSize = ShotgunData.magazineSize;
            break;
        case weaponsEnum.SNIPER:
            currentAmmo = SniperData.currentAmmo;
            magazineSize = SniperData.magazineSize;
            break;
        default:
            currentAmmo = 0;
            magazineSize = 0;
            break;
    }
    DisplayText(currentAmmo, magazineSize);
}
void DisplayText(int currentAmmo, int magazineSize) {
    bulletsText.text = ":" + currentAmmo + "/" + magazineSize;
}
```

Snippet 5.42: BulletText's Update and DisplayText methods

In the update method, we set the currently selected weapon using WeaponSwitching script's getSelectedWeapon method. Based on the value returned, we set the values of the currentAmmo and magazineSize variables to be equal to the currentAmmo and magazineSize fields of the GunData object corresponding to the currently selected weapon. Finally, we call the method DisplayText, with currentAmmo and magazineSize as parameters.

In the DisplayText method, we create a string of the format: ":currentaAmmo/magazineSize" and display it in the HUD by setting this string as the text field of the bulletsText component.

5.6.3 PAUSE SCREEN

Figure 5.10: Enter Caption

The Pause screen is a "Quality of Life" (or QoL) feature, because it offers the player an easy and convenient way to stop the game, and offers the option to continue playing without losing progress. When the player presses the "P" key, the game will freeze and the player will be presented with the Pause Screen, where he can choose to either resume the game, return to the main menu, or exit the game completely.

This functionality is implemented by the PauseMenu script.

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.P))
    {
        if(isGamePaused)
        {
            ResumeGame();
        }
        else
        {
            PauseGame();
        }
    }
}
void PauseGame()
{
    isGamePaused = true;
    pauseMenu.SetActive(true);
    Time.timeScale = 0;
    Cursor.lockState = CursorLockMode.Confined;
    Cursor.visible = true;
}
```

Snippet 5.43: PauseMenu's Update and PauseGame methods

In the Update method, we check if the player has pressed the "P" key. If this has happened, we pause the game by calling the method PauseGame, or, if the game was already pause when the key was pressed, this means that the player wants to unpause the game, and thus we call the method ResumeGame.

In the PauseGame method, we set the field isGamePaused to true to let the script know that the game is paused, activate the game object pauseMenu, which will make the pause menu visible, set the time scale to zero, which will effectively freeze the game, and lastly we enable the mouse cursor.

```
public void ResumeGame()
{
    pauseMenu.SetActive(false);
    isGamePaused = false;
    Time.timeScale = 1;
    Cursor.lockState = CursorLockMode.Locked;
    Cursor.visible = false;
}
public void MainMenu()
{
    SceneManager.LoadScene("MainMenu");
    isGamePaused = false;
    Time.timeScale = 1f;
}

public void QuitGame()
{
    Application.Quit();
    Debug.Log("Quitting Game");
}
```

Snippet 5.44: PauseMenu's ResumeGame, MainMenu and QuitGame methods

The ResumeGame, MainMenu and QuitGame methods are called through the editor when the player presses one of the corresponding buttons from the pause screen. In ResumeGame, we set the pauseMenu game object to inactive, which hides it during the gameplay, set the time scale back to one to unfreeze the game, and finally lock the cursor to the center of the screen and then hide it.

The MainMenu method will switch the scene to the "MainMenu" scene, and make sure the timescale is set back to normal and the game is set to unpaused.

The QuitGame method will end the execution of the game by calling the method Application.Quit.

5.6.4 MAIN MENU AND ENDINGS SCREENS

Figure 5.11: Screenshot from the Main Menu

The main menu is the first thing the player will be presented with when he runs the game. Clicking the Play button will start the game, by switching to a new scene using the command SceneManager.LoadScene, similar to how we've seen in the MainMenu method of the PauseMenu script. Clicking on Options will present the player with an options screen, which contains a slider that allows the player to select the desired audio volume (or disable it completely by dragging it all the way to the left).

Figure 5.12: "Game Over" and "Game Won" screens

The "Game Over" and "Game Won" screens are presented to the player depending on the outcome of the game, with each representing a different scene in the game files. The scene is changed inside the GameOver and GameWon methods of the GameManager script, as discussed during the "General" section of this chapter. With each of these two ending screens, there is a script which implements the correct display of the number of waves survived and zombies killed, by reading the static values zombiesKilled and wavesSurvived.

6. CONCLUSIONS

It had always wanted to develop a video game from scratch, and after my exchange semester abroad, where I had the opportunity to study game development and developed a few small projects as practice, I decided this diploma thesis project would be a good opportunity to put all my programming skills to the test. The objective of this project has been the creation of a fully-fledged first-person shooter video game, which includes mechanics like the gun system, enemy AI, user interface, difficulty scaling, animations, sounds, effects, all integrated into a complete final product, that is also programmed for efficiency and designed in a way that allows for the further addition of other gameplay mechanics.

6.1 FUTURE WORK

I plan to continue working on "Zombie Blitz", as it is a project that I am very proud of, and there are a few gameplay features that I wish to add to the game, such as:

- The addition of an "Endless" game mode, where the waves never end.
- The addition of pickups, which could take the form of health pickups, bullet pickups, or power-ups which would give the player different advantages such as increased speed, damage, armour, etc.
- The addition of an upgrade system, which would allow the player to spend his score points to increase his stats.
- The addition of more guns to the player’s arsenal

7. BIBLIOGRAPHY

[1] *How many gamers are there? (2024 statistics)*, https://whatsthebigdata.com/number-of-gamers/, Accessed: 30.05.2024, 2024.

[2] *Video games - worldwide*, https://www.statista.com/outlook/dmo/digital-media/video-games/worldwide, Accessed: 31.05.2024, 2024.

[3] D. Talevski, *How much is the gaming industry worth in 2024?* https://techjury.net/blog/gaming-industry-worth/, Accessed: 31.05.2024, 2024.

[4] M. Rouse, *What does first person shooter mean?* https://www.techopedia.com/definition/241/first-person-shooter-fps, Accessed: 04.06.2024, 2011.

[5] M. Toftedahl, *Which are the most commonly used game engines?* https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-, Accessed: 30.05.2024, 2019.

[6] *Left4dead steam page*, https://store.steampowered.com/app/500/Left_4_Dead/, Accessed: 07.06.2024.

[7] *No more room in hell official website*, https://www.nomoreroominhell.com/about/, Accessed: 07.06.2024.

[8] *Zombie steam page*, https://store.steampowered.com/app/898690/Zombie/, Accessed: 07.06.2024.

[9] *What is unity? – a top game engine for video games*, https://gamedevacademy.org/what-is-unity/, Accessed: 29.05.2024, 2023.

[10] *Unity user manual 2022.3*, https://docs.unity3d.com/Manual/index.html, Accessed: 01.06.2024.

[11] S. Golodetz, "Automatic navigation mesh generation in configuration space," 2013.

[12] D. Brewer, "Tactical pathfinding on a navmesh," 2013.

[13] X. Xu, "Automatic generated navigation mesh algorithm on 3d game scene," *Procedia Engineering*, 2011.

[14] *Build vr: Using unity navmesh for first person movement in vr*, https://medium.com/@davijoh73/build-vr-using-unity-navmesh-for-first-person-movement-in-vr-64efe0909d84, Accessed: 05.06.2024.

[15] *What can you do with visual studio?* https://visualstudio.microsoft.com/vs/features/, Accessed: 03.06.2024.

[16] *Visual studio 2022 version 17.9 release notes*, https://learn.microsoft.com/en-us/visualstudio/releases/2022/release-notes-v17.9#1793--visual-studio-2022-version-1793, Accessed: 03.06.2024.

[17] *C# programming: What it is, how it's used + how to learn it*, https://www.coursera.org/articles/c-sharp, Accessed: 02.06.2024.

[18] S. Chacon, *Pro Git*. 2009.