FGTE6003: Final Major Project - Timeleap

Charlie Skinner 2003490
3rd May 2024

Table of contents

Introduction

For my Final Major Project, I have created a first-person shooter game using the Unity Game Engine. This game revolves around parkour-based movement mechanics. Throughout the development phase of this project, I have been able to produce a vertical slice of my game, Timeleap. Had I had more time to do so, I would have aimed to create a full game. This Games Design Document (GDD) aims to give an overview of my project, by covering a range of things, such as the game narrative, key mechanics and the development timeline.

Game Overview

Game Inspiration

For my Final Project, I initially took heavy inspiration from the following games:

- Call of Duty Black Ops 3
- Titan Fall 2
- Mirrors Edge
- Rollerdrome

I made this choice as reference for inspiration because these games all feature gameplay that revolve around the movement mechanics that have been implemented. The movement mechanics featured in these games are also crucial to the gameplay experience of their target audience(s).

I mostly took inspiration from these games regarding environment design; they are all set in a futuristic timeframe, however some levels within these games appear to use different time-periods as inspiration. This has led to my games environment revolve around a destroyed world; which explains why there are floating islands - the destroyed environment also allows the player to use the movement mechanics I have implemented in this game to traverse through the levels.

Genre

I have designed this game with three genres in mind:

- FPS
- Action
- Parkour

Now that the game has been made, I believe that the work I have produced allows for my game to fall under these genres, though in some areas it may be a loose-fit. I chose these genres as inspiration because the action and parkour genres tend to accompany each other rather well, especially showcased in other popular games under these genres. I also targeted

the FPS genre because it is what I am personally most experience with regarding the production of games.

Narrative

Timeleap has a very basic narrative featured in the game. This is mostly due to me perceiving my projects from an almost truly technical perspective, which makes it difficult to come up with game elements such as a narrative or story.

The narrative featured in Timeleap purely revolves around the tutorial scene; as this is the whole premise / set-up for the narrative within my game. The tutorial scene teaches the player the basic controls and an introduction to the movement featured. The player then comes across a pistol, that the game then tells the player to pick-up and then shoot a non-hostile NPC, who is shown to be stood next to a campfire. However, upon shooting the NPC, a futuristic looking platform / island appears, carrying more enemies - this time they're carrying weapons. This acts as a way to indicate to the player that whoever the NPC was that they killed, was not the best person to choose to kill out of impulse. The player then defeats the hostile enemies, and the game tells the player to interact with the giant satellite at the end of the platform, which is highlighted using a red outline. Upon interacting with the computer, a portal appears and the player goes through it. The player then ends up on another floating island, but there is a different time-setting in this stage. Accompanied by this, there are also even more hostile enemies that are displeased with the player.

Unique Selling Points (USPs)

Unique Selling Point 1: Movement Mechanics

One of the main selling points for my game is that the games functionality revolves around the movement mechanics that are implemented within. This is because a lot of games that are advertised to be parkour-shooter games tend to focus primarily on the other mechanics in a game, making the games movement feeling like it was added as an afterthought. This is the main aim that I'm trying to prevent with my game.

Unique Selling Point 2: Cross-platform compatibilities

Another main selling point for my game is that it will be available on two desktop platforms; both Windows and Linux. This was designed mostly in mind with the increasing popularity of hand-held gaming devices running Linux, such as the Steam Deck. According to the Steam Hardware Survey (https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam), the percentage of gamers using Linux is steadily increasing, alongside the percentage of gamers using Windows slowly decreasing. By making this game

both Linux and Windows native, it increases the availability of my game while making gamers on each platform more likely to consider purchasing this game, if it was ever officially released. This has been further achieved by using the Vulkan Graphics API, with the same graphics front-end for both platforms it removes the limits of only having certain features only available on a single platform.

Unique Selling Point 3: Game Optimisation

The targeted optimisation of my game can be another unique selling point for my game. With upcoming newly releasing games requiring a more and more high-end computer specification to run the games smoothly, it's becoming increasingly difficult for gamers on older hardware to run newly released games at a smooth frame-rate. The optimisations in my game aim to alleviate this; I want anyone who is interested in playing my game, to be able to play my game at an enjoyable experience.

Target Audience

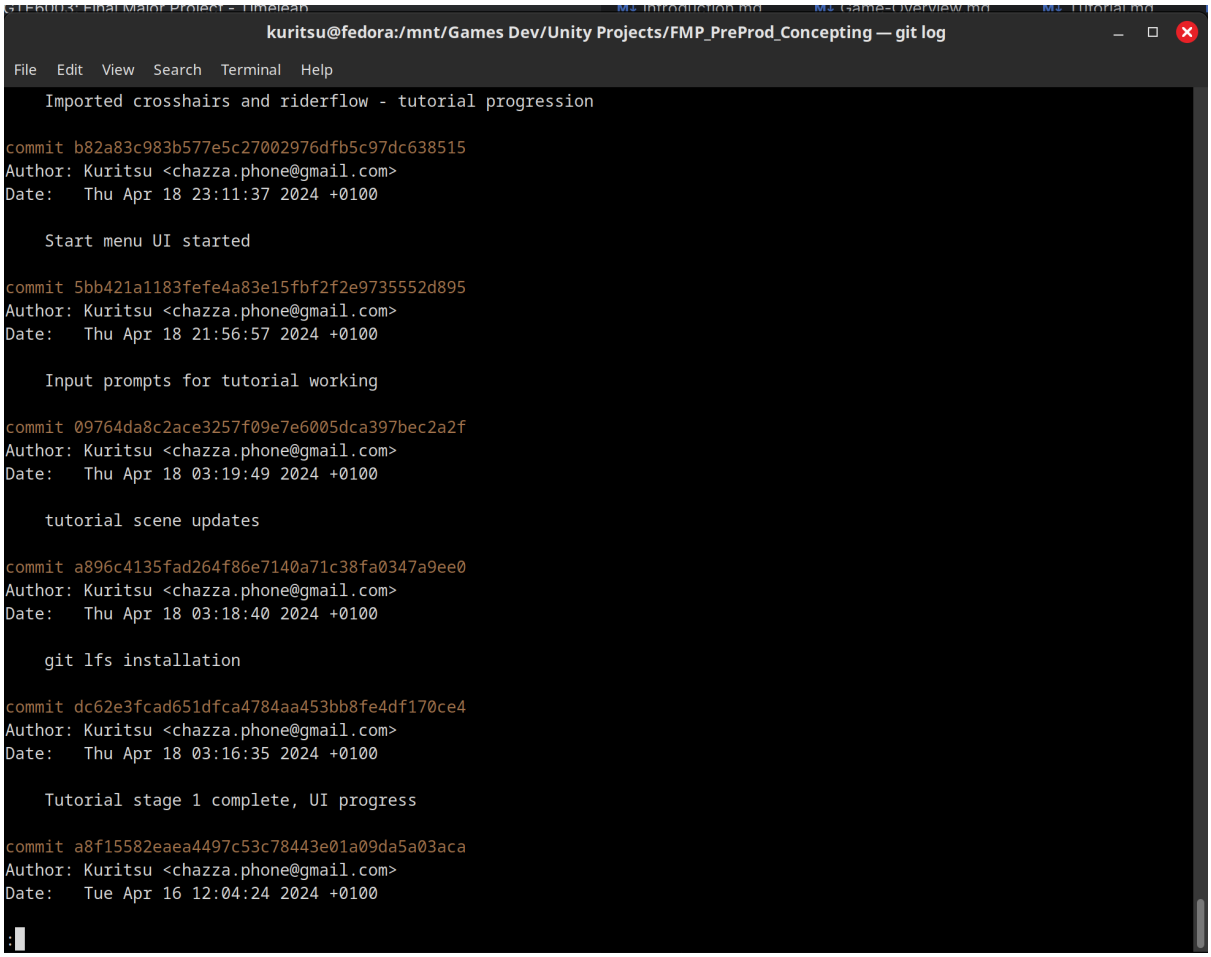
I have designed this game with a wide target audience for this game. The ideal target audience for my game would be gamers, ideally on the desktop platform, who have an interest in fast-paced movement mechanics within shooter games. As mentioned in my Unique Selling Points, I want to have as wide of a target audience as possible, as a way to extend the games outreach to potential players of this game.

Project Management

Throughout both the pre-production phase and production phase of this project, I have used a variety of software to store and use reference of what files have been worked on and when.

GitHub / Git CLI

This was the main use of external services being used by my project. I've also used this as a method to keep track with my project development and progression - the ability to look through my commit history, track the changes to files I've made is extremely helpful.



Git CLI Screenshot

Discord

I've used my own personal discord server as a method to store reference to certain areas of my project that I need to develop further, as well as sometimes a last-minute method to send files between devices. I have also used this to communicate with my fellow course-mates, to

receive feedback and help on certain areas I had become stuck with throughout the project development.

Technicalities

Platform

I have developed Timeleap for both the Windows and Linux desktop platform. This is due to the increasing market of people using Linux for gaming - this can be due to the increasing number of users owning a Steam Deck by Valve, but could also be due to the increasing number of desktop gamers who tend to use Linux over Windows. There are a few challenges accompanied by this, but throughout development I have been testing consistently on both Linux and Windows systems.

Target Hardware

I have optimised my game to run on a large variety of hardware. This prevents the potential audience from not needing an insanely high specification system to play my game with an optimal experience. I have done this as a counter to the number of modern games which hve been releasing year by year, with increasingly more and more powerful hardware requirements for their games to run smoothly. To be more specific, here is the baseline minimum spec to run my game smoothly:

- **CPU:** Intel i5 6400 / AMD Ryzen 5 1600
- **Memory:** 8GiB Minimum, 16GiB Recommended
- **Storage:** 10GiB Available
- **OS:** Windows 10 64-Bit / Linux distro with Kernel 4.19 LTS or newer
- **Display:** 1920x1080 Resolution recommended

Software & Hardware Technicalities

- **Game Engine:** Unity Engine 2022.3.0f1
- **Graphics Pipeline:** Universal Render Pipeline
- **IDE:** JetBrains Rider
- **VCS:** GitHub via Git Bash w/ ZSH

- **Graphics API Frontend:** Vulkan
- **Operating System:** Windows 11 Pro 22H2 / Linux Mint 21.2 "Victoria" / Fedora Linux Release 39

Timeleap has been tested consistently on the following hardware:

System 1 - Windows 11 Pro 22H2 & Fedora Linux Release 39

- **CPU:** AMD Ryzen 9 3900X
- **Memory:** 48GiB DDR4 @ 3600Mhz
- **GPU:** Nvidia GeForce RTX 2080 Super

System 2 - Windows 10 Pro & Linux Mint 21.2

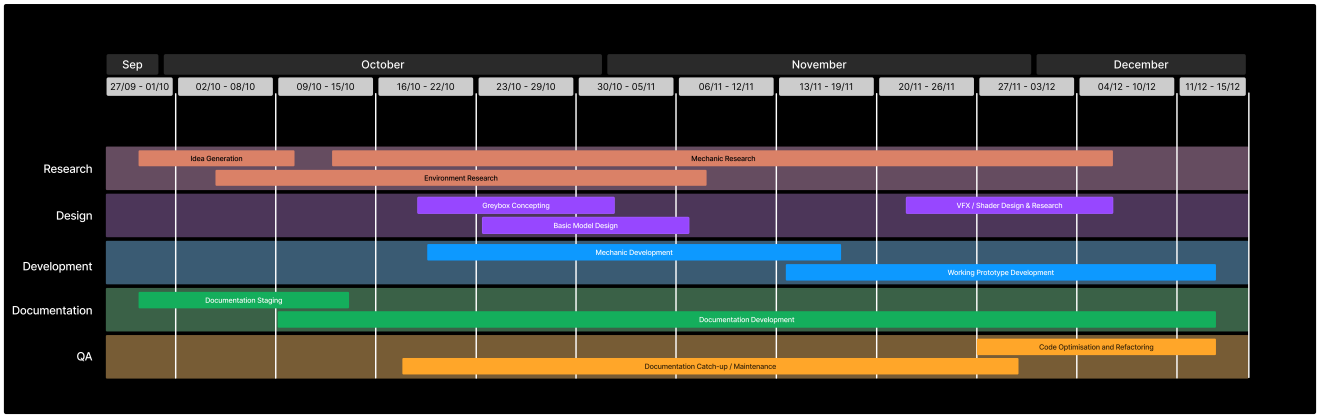
- **CPU:** Intel Core i7 6700K
- **Memory:** 16GiB DDR4 @ 3600Mhz
- **GPU:** AMD Radeon Vega 64

System 3 - Linux Mint 21.2

- **CPU:** Intel Core i7 7700HQ
- **Memory:** 16GiB DDR4 @ 2400Mhz
- **GPU:** Nvidia GeForce GTX 1060 3GB Mobile

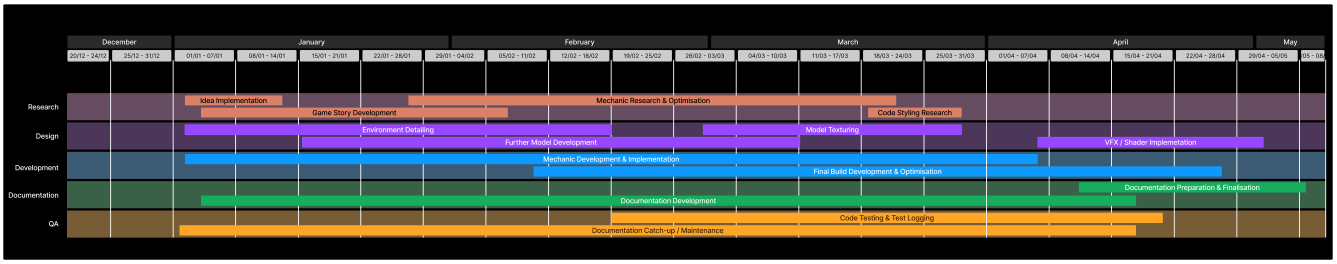
Development Timelines

Term 1 - Pre-Production Timeline



This was my initial proposed development timeline for the pre-production period of my game. I tried to stick with this plan as much as possible throughout the pre-production phase, however I'd say that I fell behind in certain areas, causing me to have to do some catching up during the production phase.

Term 2 - Production Timeline



This was my initial development timeline for the production period of my game. I would have liked to say I stuck with it, however the only thing I truly followed through with this was work towards the mechanics, and then aimed on getting everything else complete towards the end of the production phase.

Gameplay & Mechanics

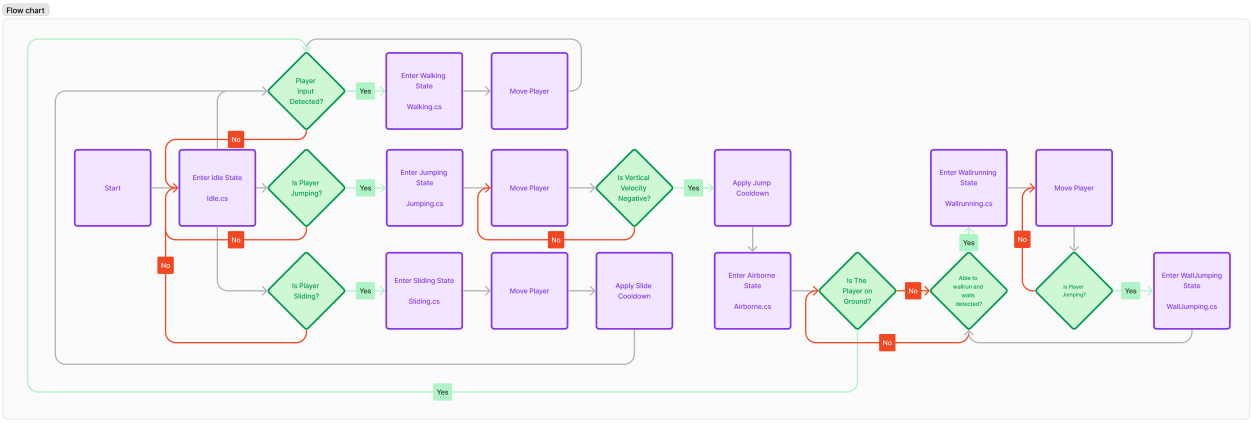
Movement

For my players movement, I implemented a Finite State System. This allows for each movement state to be isolated in their own script. This prevents the chance of one script become too large to easily maintain, and also makes mechanic debugging far easier. This aids my efficiency in implementing player movement mechanics, if there is an issue I can dial the issue down to one single state, identifying one single script that the issue will be happening within. I have implemented the following movement states into my player:

- Idle
- Walking
- Sprinting
- Jumping
- Airborne
- Wall Running
- Wall Jumping
- Sliding

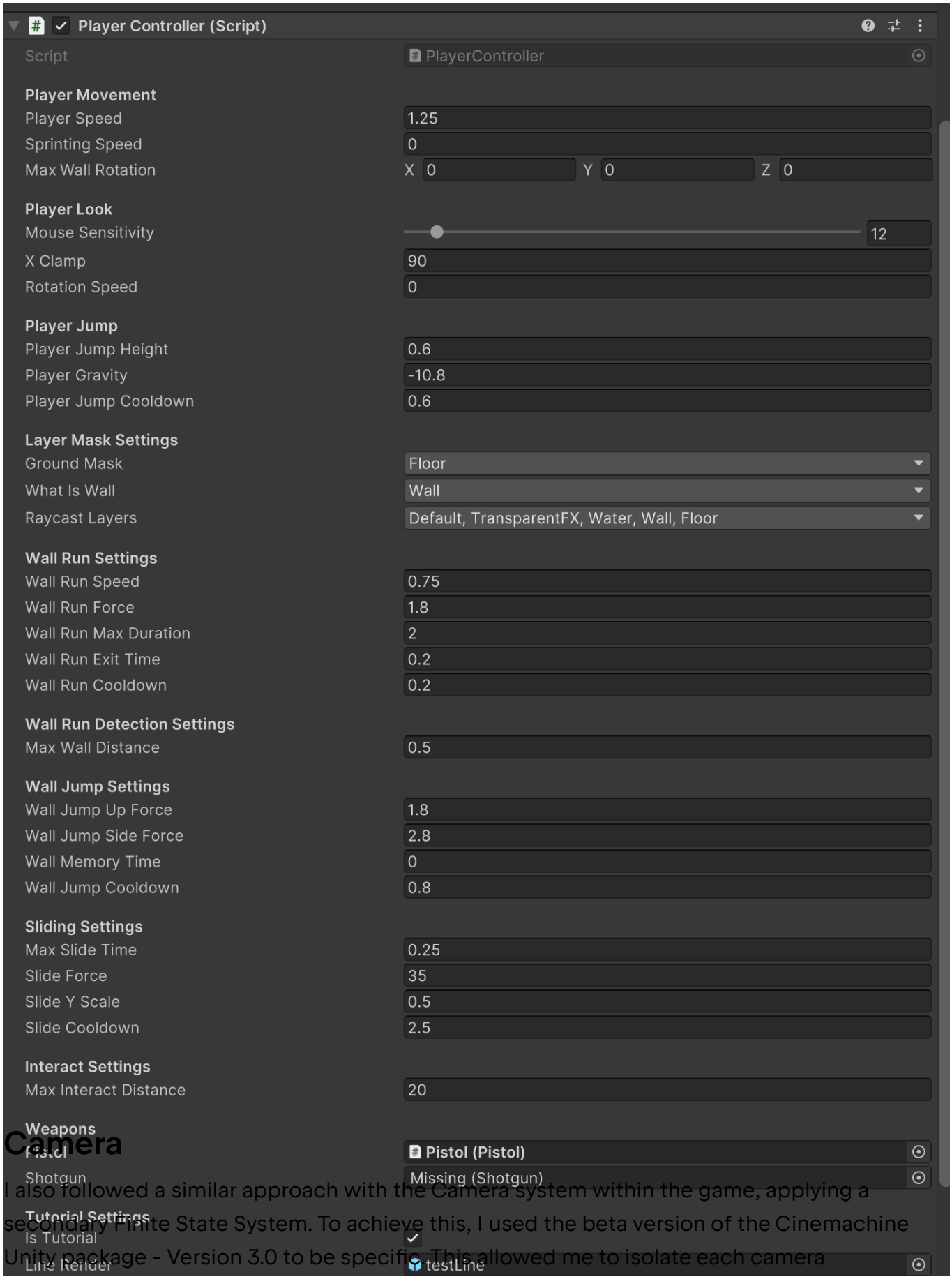
Player State Machine Flowchart

To explain how this Finite State Machine works, I have produced a flowchart that explains the basic workflow:



This is also being done to make the players input feel as responsive as possible. I have set this finite state machine up to be completely configurable, allowing these implementation to give the player a fluid feeling of movement. This also allows for a deeper level of input feedback alongside hopefully increasing the potential to become immersed in my game.

Player Settings Showcase

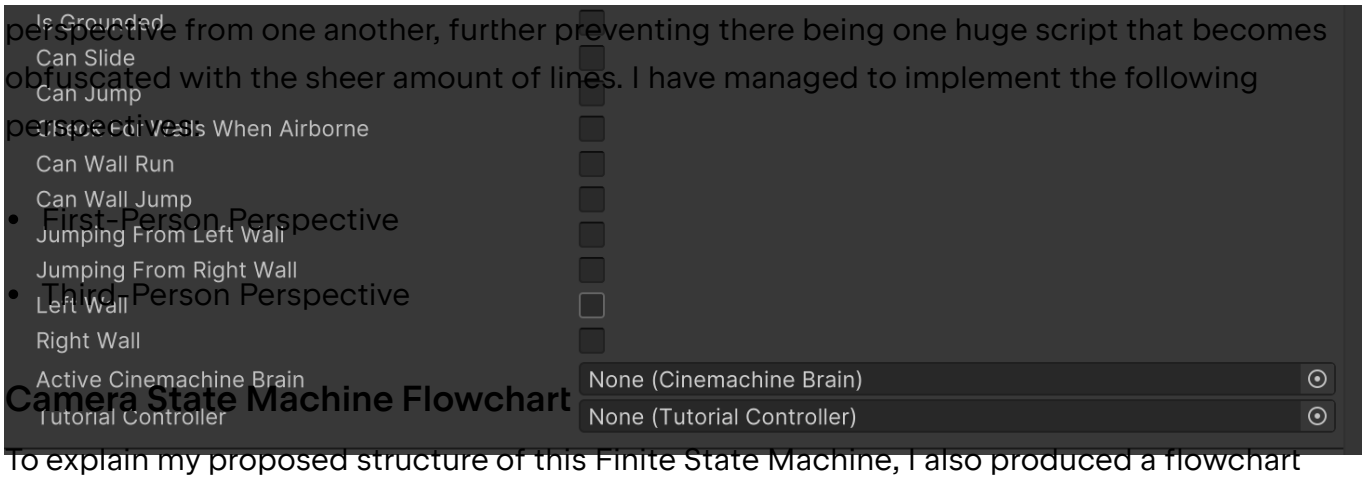


Camera

I also followed a similar approach with the Camera system within the game, applying a secondary Finite State System. To achieve this, I used the beta version of the Cinemachine Unity package - Version 3.0 to be specific. This allowed me to isolate each camera

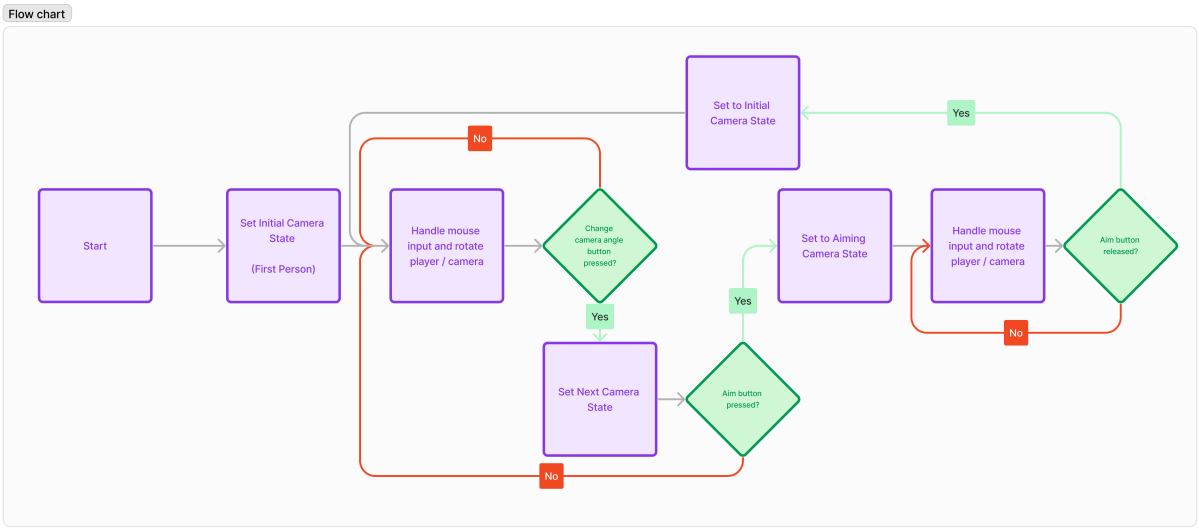
perspective from one another, further preventing there being one huge script that becomes obfuscated with the sheer amount of lines. I have managed to implement the following perspectives:

- First-Person Perspective
- Third-Person Perspective



Camera State Machine Flowchart

To explain my proposed structure of this Finite State Machine, I also produced a flowchart explaining the workflow:



I was initially intending on implementing more varieties of camera perspectives within my game, however the amount of time it took to learn the new Cinemachine 3.0 API and integrate that with the state machine was taking too much time. I decided to cut my losses and stick to only two perspectives, which can be showcased throughout play-through of my game. It is not the most perfect implementation as I ran out of time to finish of the third person mechanics within the game, but it can still be shown working transitioning between each state.

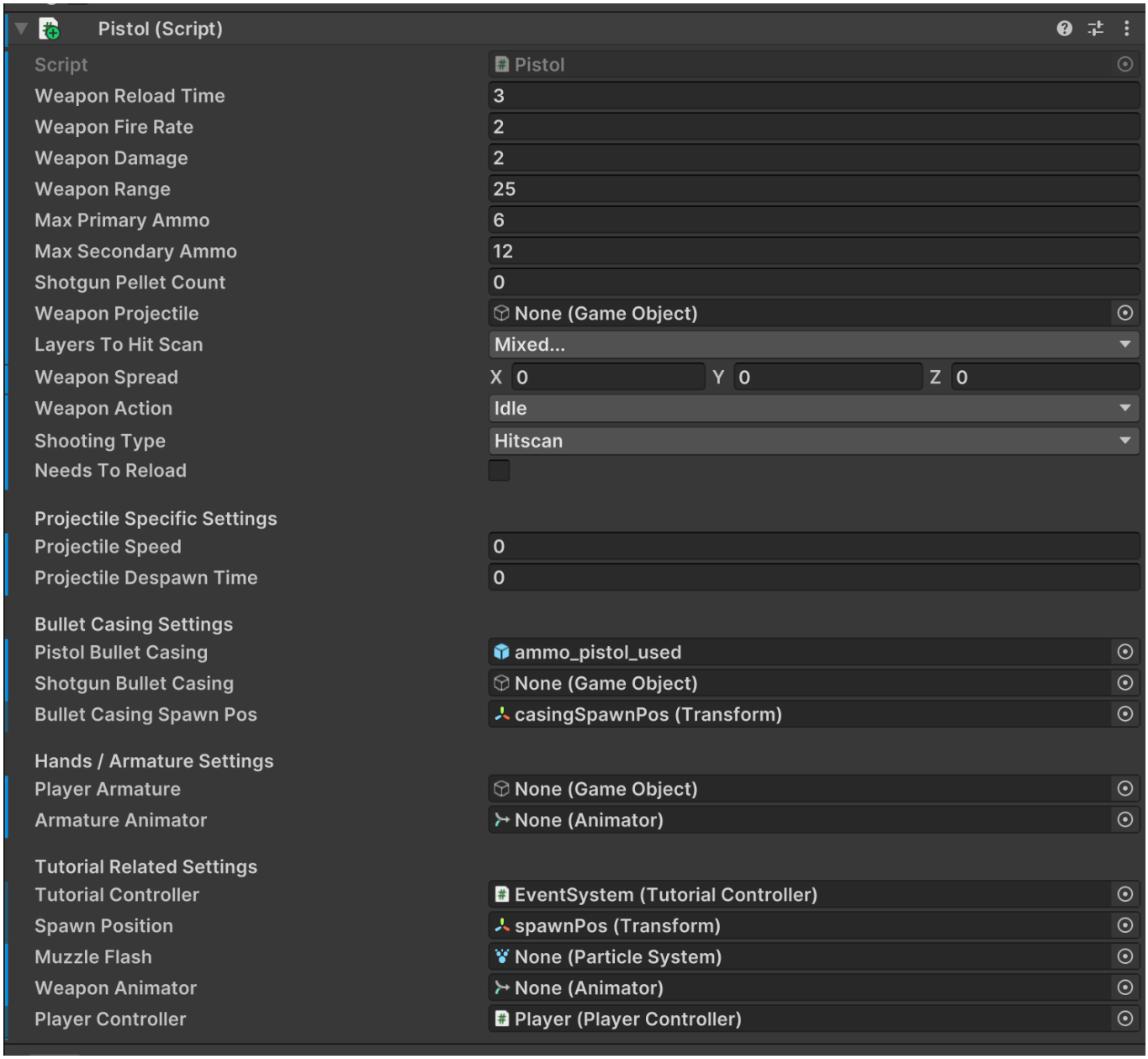
Weapons

I designed the weapon mechanics in my game around a single base class - this allows for me to easily implement any further weapons, using the configurability that I have designed in the base class. Throughout development, I have been able to implement two weapons:

- Pistol
- Shotgun

I would have ideally liked to have more implemented towards the end of development, but I feel like these two weapons are enough to showcase in my vertical slice. The base class has been designed to have the options to switch between projectiles, or raycast, and also allows for the option to have object pooling, increasing the efficiency / optimisation of my game.

Weapon Settings Showcase

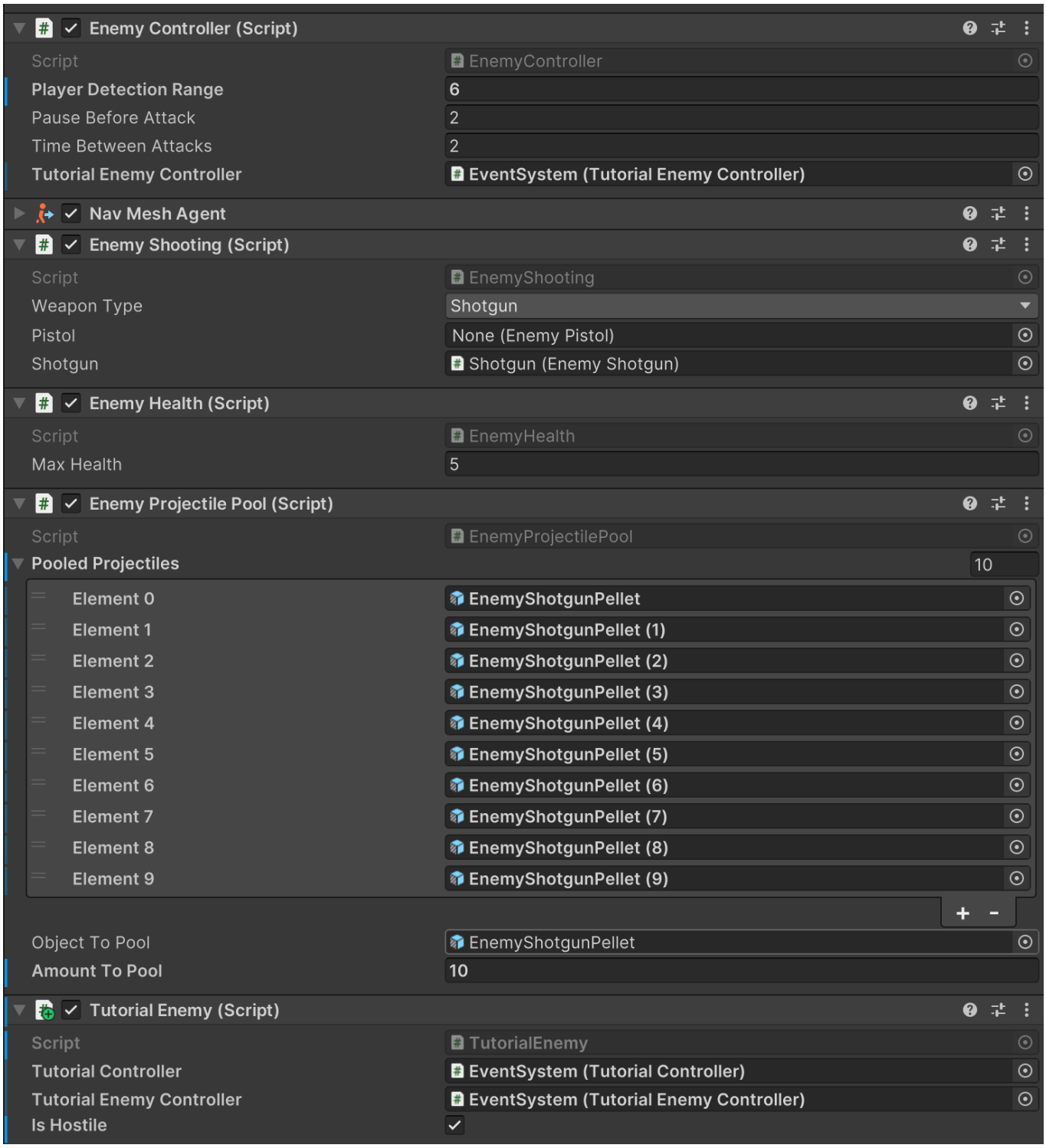


Enemy AI

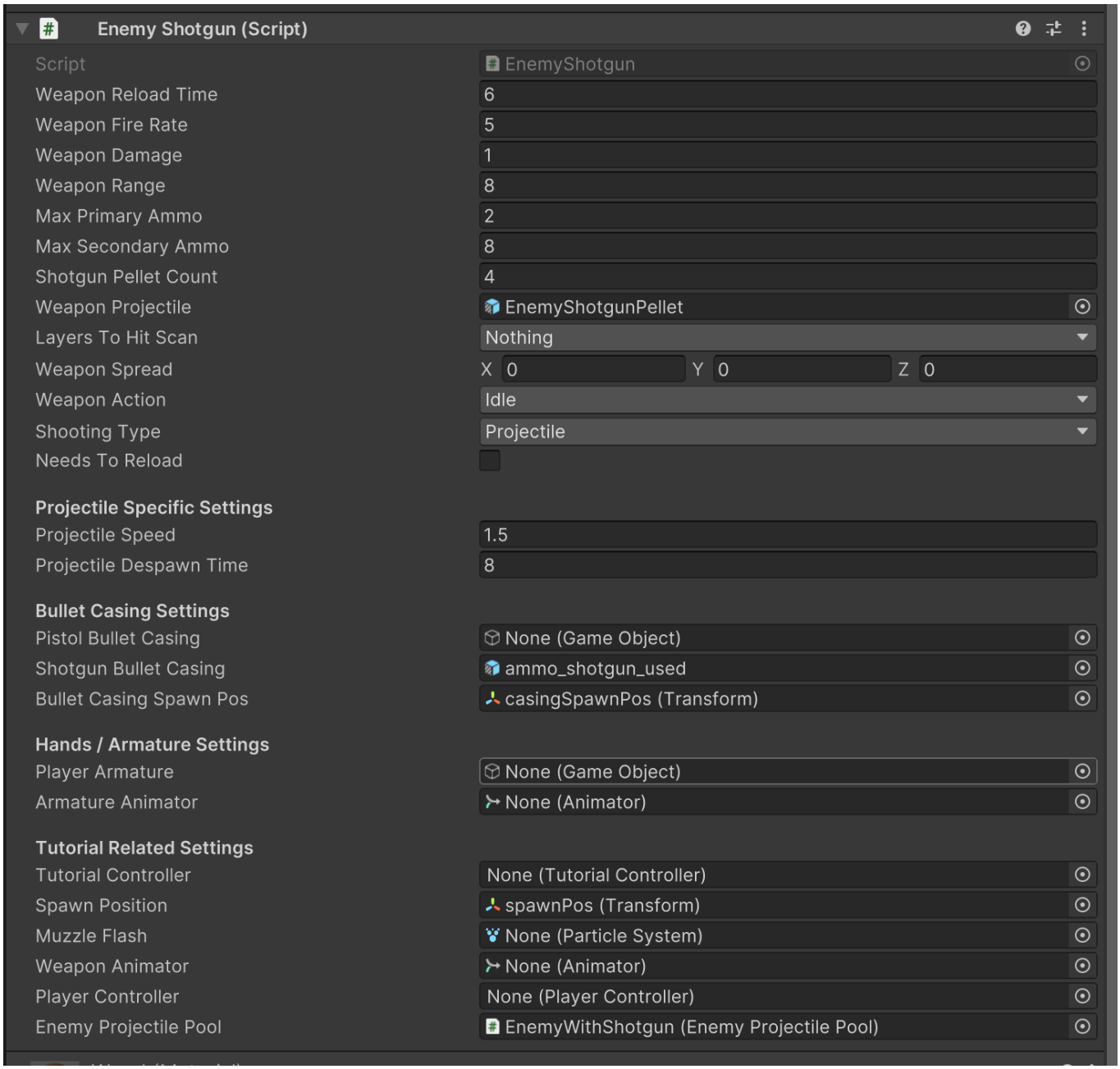
I initially intended for my enemies AI to be far more advanced than it actually turned out to be. I initially intended for my enemies to have the ability to wall run, just like the player could. However, due to several attempts to implement this within unity, it turned out to be far far far more difficult than I had originally expected it to be, causing my to remove the idea and stick with a basic enemy implementation.

I set up an enemies weapon mechanic, deriving from the base class for the weapons I mentioned in the section beforehand. This then allows for further configuration of the enemy AI and its complexity; allowing it to scale with the difficulty as the player progresses through the game.

Enemy Settings Showcase

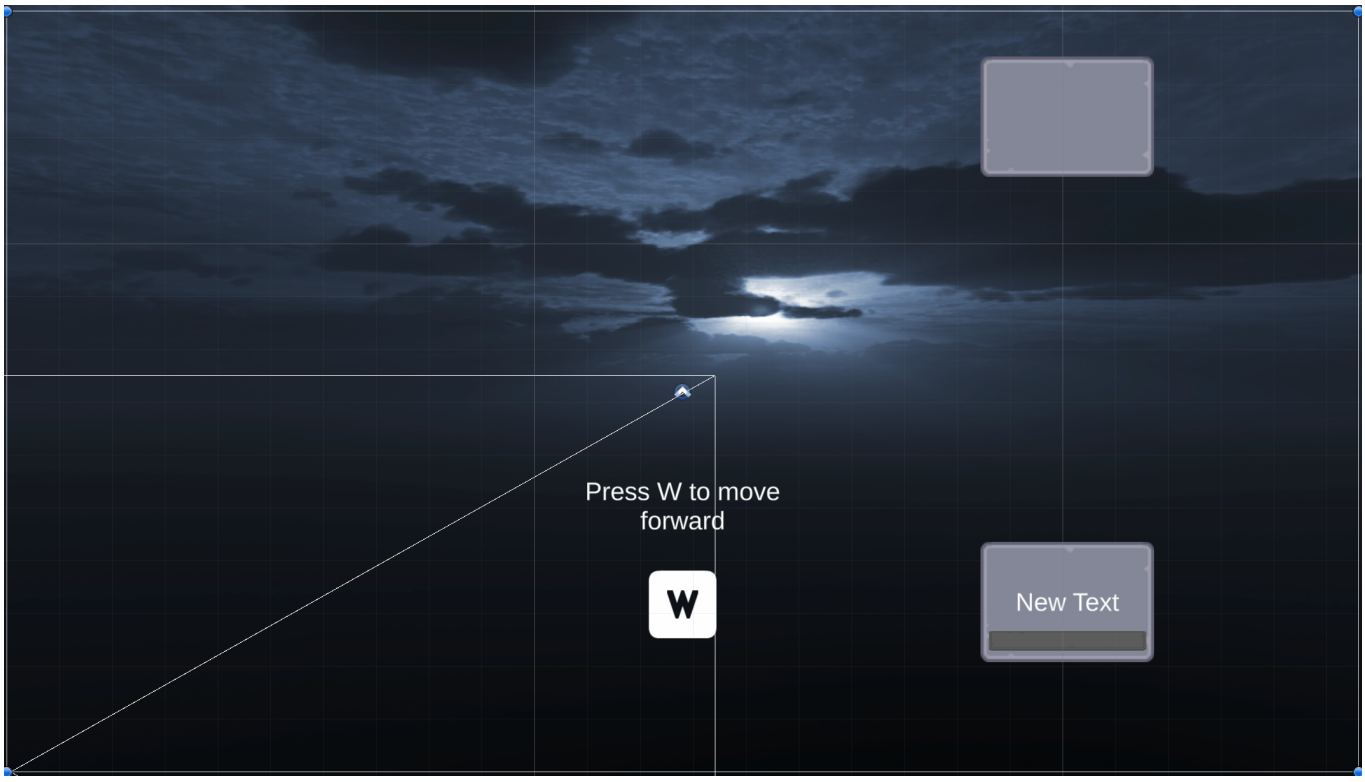


Enemy Weapon Settings Showcase



UI

UI within my game is very basic - it is rather minimal, whilst still being enough to explain what is going on within the game. If I had more time to work on this project, I would have extended the variety of UI assets that would have been implemented within my game.



Environment

I have a few scripts that control the environment within my game. This ranges from a light flickering script, to a move walls script, to some out of bound sensors. These scripts are crucial to making my environment feel interactive throughout play-through.

Input & Keybinds

Keyboard and Mouse Input

I intend for this game to mostly be played with Keyboard and Mouse input. The default control scheme is the following:

- **W**: Forward
- **A**: Left
- **S**: Right
- **D**: Backwards
- **V**: Camera Perspective
- **C**: Player Slide
- **Spacebar**: Player Jump
- **Mouse Movement / Delta**: Player Look
- **Mouse 1** (Left Click): Player Shoot
- **Mouse 2** (Right Click): Player Aim (**Not Implemented**)

Controller Input

I also initially intended to implement controller input, alongside keyboard and mouse input. Due to leaving this too late in the project, I have not had the time to implement this. If this were to be integrated into the final game, the default control map would be the following (using Xbox One Controller Scheme as reference):

- **Left Joy Axis**: Player Movement
- **Right Joy Axis**: Player Look
- **A**: Jump
- **Left Joy Button**: Player Slide

- **Left Trigger**: Player Aim (**Not Implemented**)
- **Right Trigger**: Player Shoot
- **Select**: Camera Perspective

Game Tutorial

The first level that the player plays of this game is the game tutorial. This is also the setup for the games narrative and plot. I have done this because it is a good way of introducing the game to the player; they get to learn the games controls and the basic game flow.

Stage 1: Starting Island

The starting island is where the player spawns in. At first, they are prompted to press the basic key inputs to move around in-game. Once the player has completed this, some walls will move up and stay floating, in the direction towards the next island. The player is then told the basics of how to wall run, and how to jump between walls.

Input Tutorial Showcase

Wall Running Tutorial Showcase

Stage 2: Weapon Island

The named "weapon island" is where the player first picks up a pistol, and is told by the tutorial to shoot a non-hostile NPC. This introduces the player to the weapon mechanics implemented within the game, and also sets up the narrative for the remainder of the game.

Weapon Island Tutorial Showcase

Stage 3: Enemy Island

The enemy island appears once the player kills the non-hostile NPC. This gives the player the challenge of defeating three hostile enemies, giving them an introduction to what the enemies they'll be fighting throughout the game are like. This is configured to be an easy challenge, so that the player does not get a sense of defeat this early on in a game. Upon defeating the enemies, the player is then told to interact with a giant computer, which is highlighted with a red outline. Upon interacting, a portal opens up and the player is told to jump through it.

Game Source

This section of the GDD is rough documentation regarding each script that is in my project. There are 60+ scripts in my project so this is very rough documentation, however I feel like it'd be worth covering in a style that is done the same way that professional code documentation is listed as.

Player Scripts

The Scripts that belong to or relate to the Player.

PlayerProjectilePool.cs

Description

This script stores a list of GameObjects that become enabled and disabled when the player shoots a weapon set to projectile mode.

Script

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

namespace Player
{
    public class PlayerProjectilePool : MonoBehaviour
    {
        public static PlayerProjectilePool SharedInstance;
        public List<GameObject> pooledProjectiles;

        [SerializeField] private GameObject objectToPool;
        [SerializeField] private int amountToPool;

        private GameObject projParent;

        private void Awake()
        {
            SharedInstance = this;
        }

        private void Start()
        {
            projParent = GameObject.FindGameObjectWithTag("ProjectilePool");
            pooledProjectiles = new List<GameObject>();
            for (int i = 0; i < amountToPool; i++)
            {
                var tmp = Instantiate(objectToPool, projParent.transform);
                tmp.SetActive(false);
```

```
                pooledProjectiles.Add(tmp);
            }
        }

        public GameObject GetPooledProjectile()
        {
            for (int i = 0; i < amountToPool; i++)
            {
                if (!pooledProjectiles[i].activeInHierarchy)
                    return pooledProjectiles[i];
            }

            return null;
        }
    }
}
```

Public Methods

GetPooledProjectile()

Returns the next non-active projectile in the scene hierarchy.

Parameters

pooledProjectiles

List of Game Objects - stores the projectiles to be used by the player

objectToPool

The Game Object being pooled i.e. The Projectile Prefab

amountToPool

An integer defining the size of the Game Object list - how many projectiles to pool

projParent

The parent game object that will contain all the projectiles to be pooled.