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

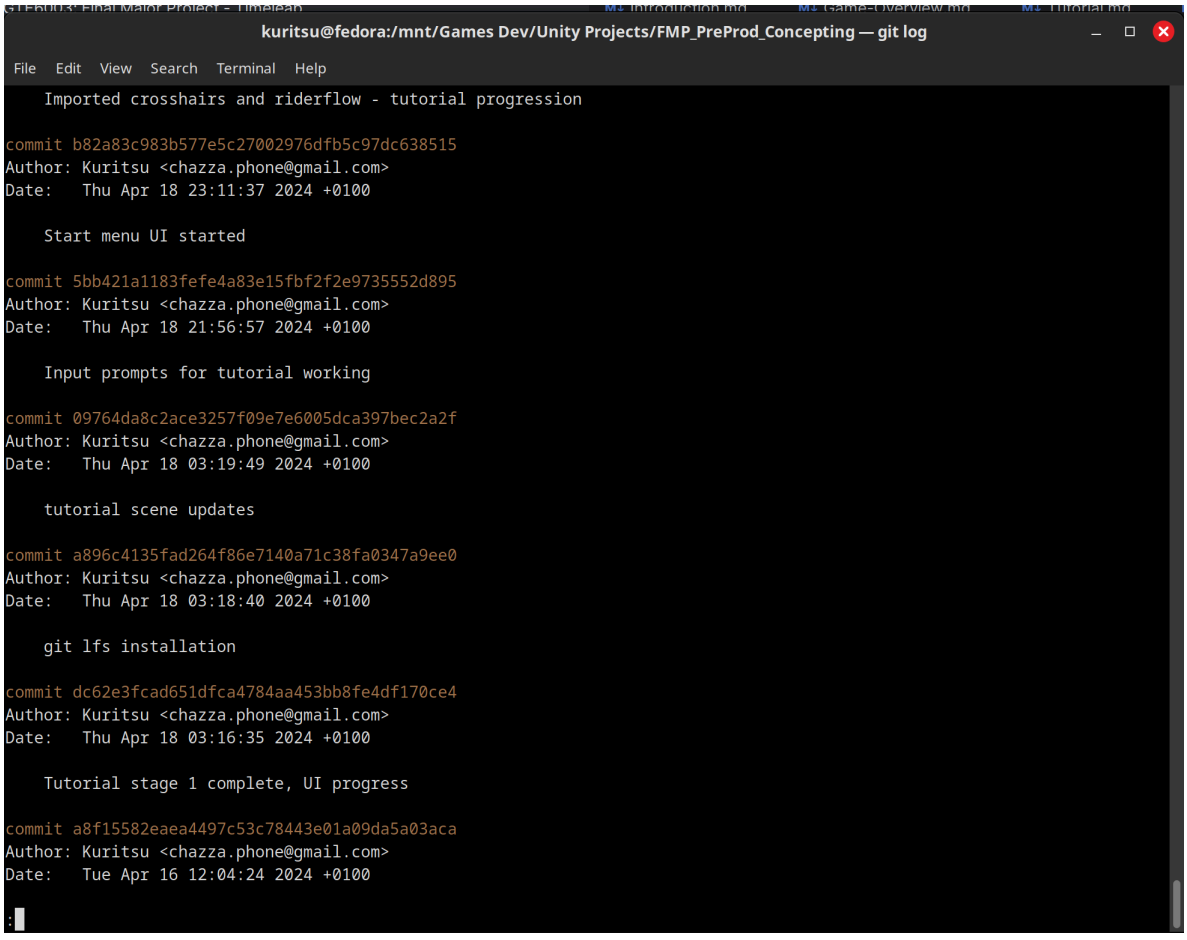
I have designed this game with a wide target audience for this game. The ideal target audience for my game would be gamers, ideally on the desktop platform, who have an interest in fast-paced movement mechanics within shooter games. As mentioned in my Unique Selling Points, I want to have as wide of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

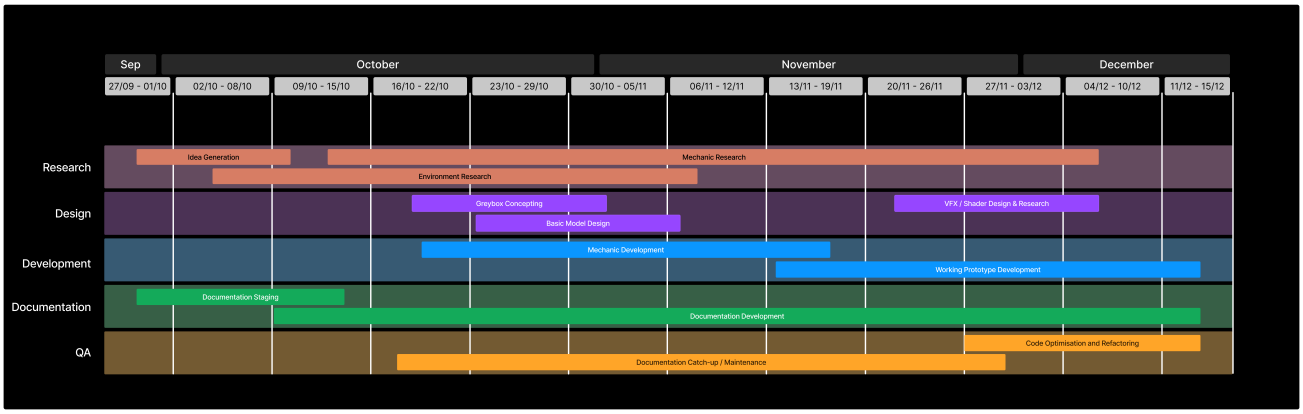
- **CPU:** Intel Core i7 6700K
- **Memory:** 16GiB DDR4 @ 3600Mhz
- **GPU:** AMD Radeon Vega 64

System 3 - Linux Mint 21.2

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

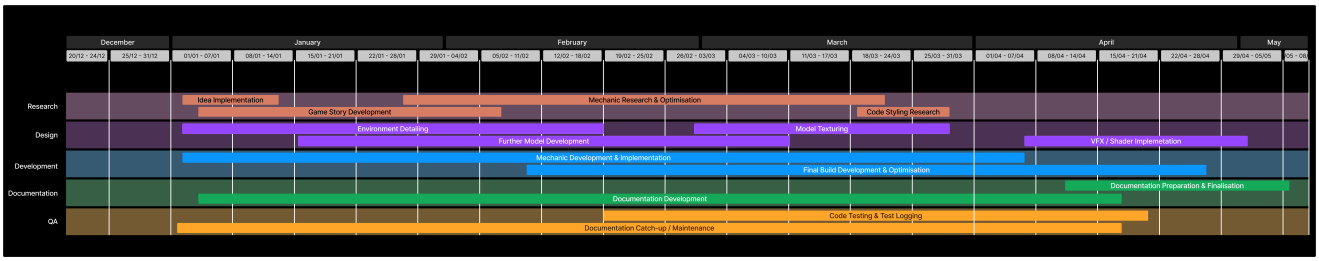
Development Timelines

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

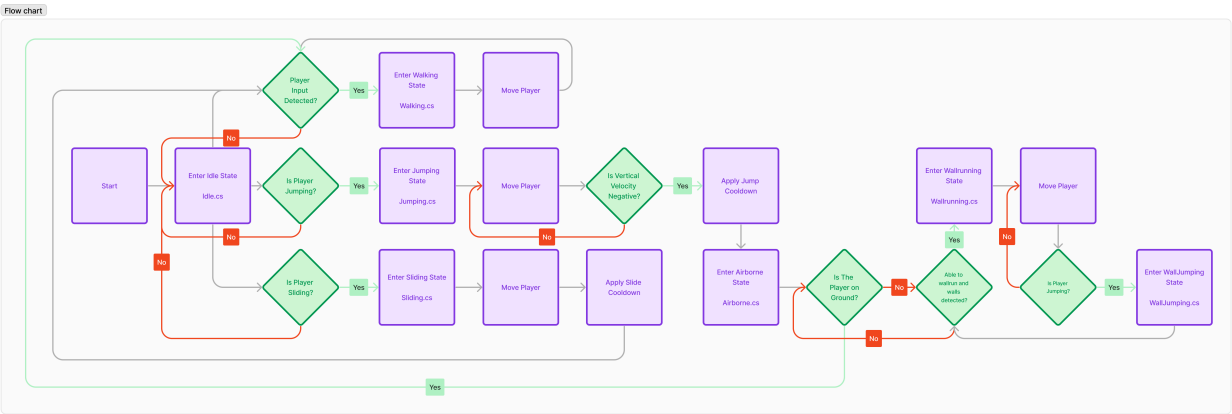
Movement

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

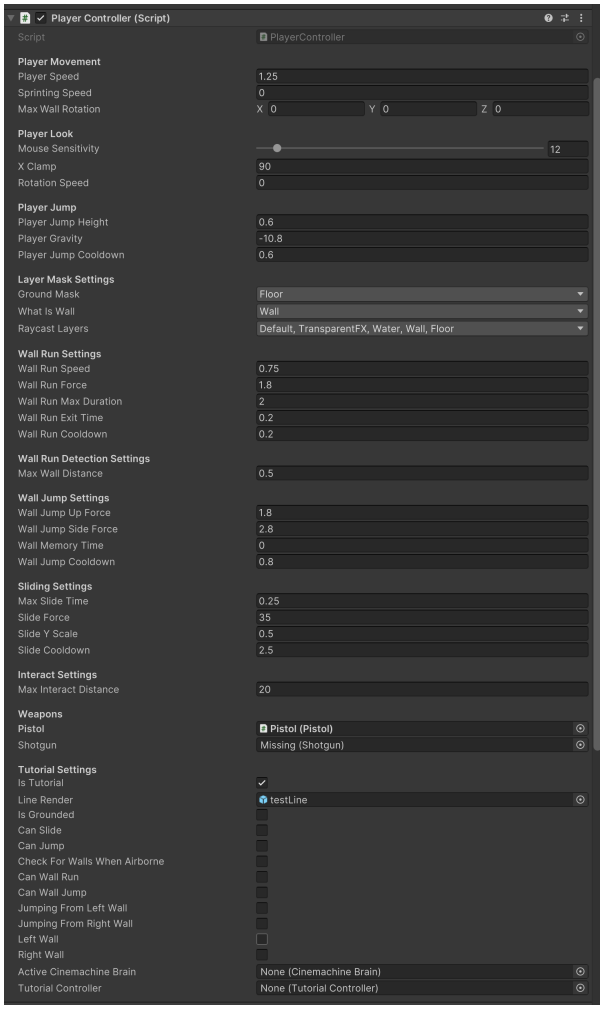
Player State Machine Flowchart

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



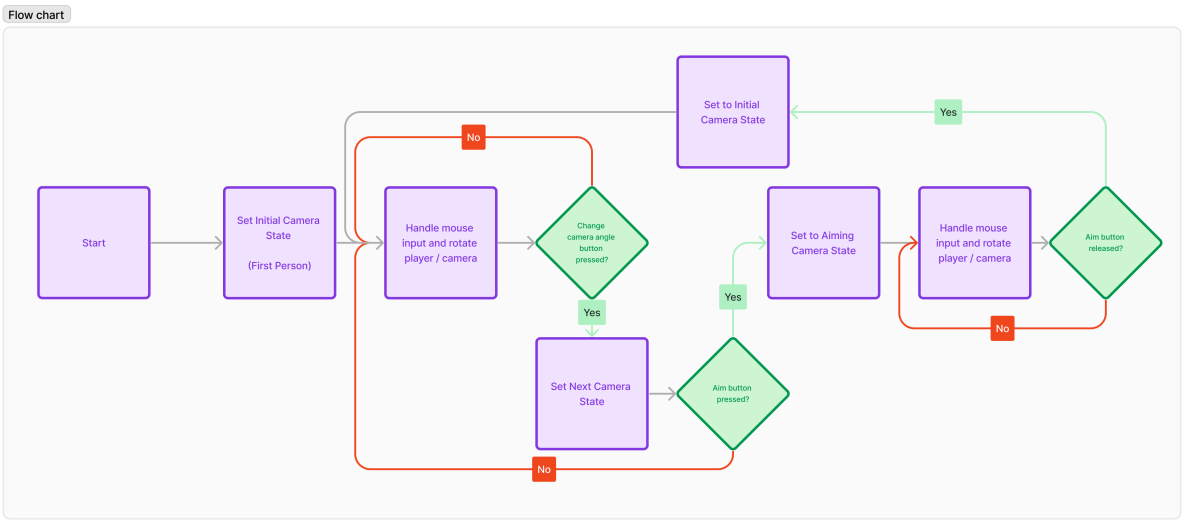
Camera

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.

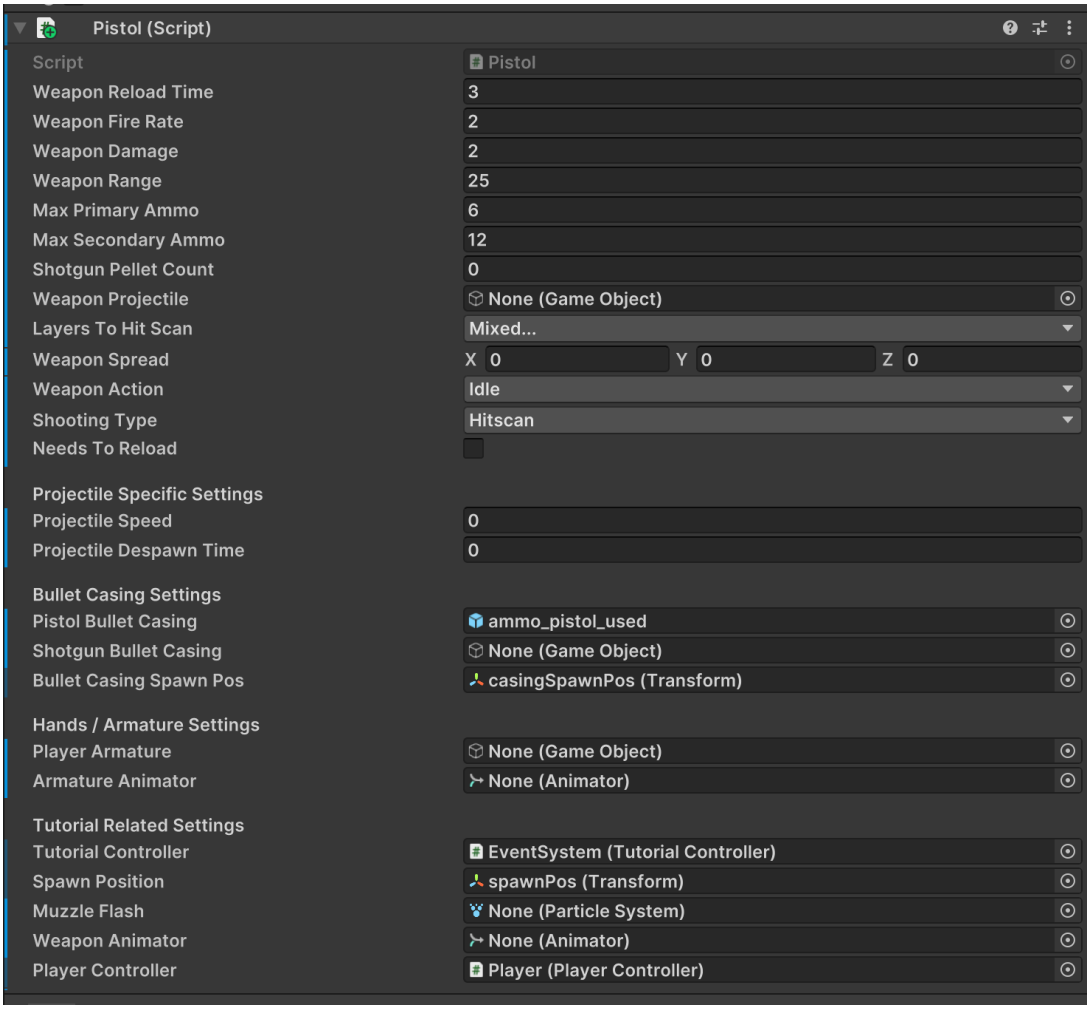
Weapons

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

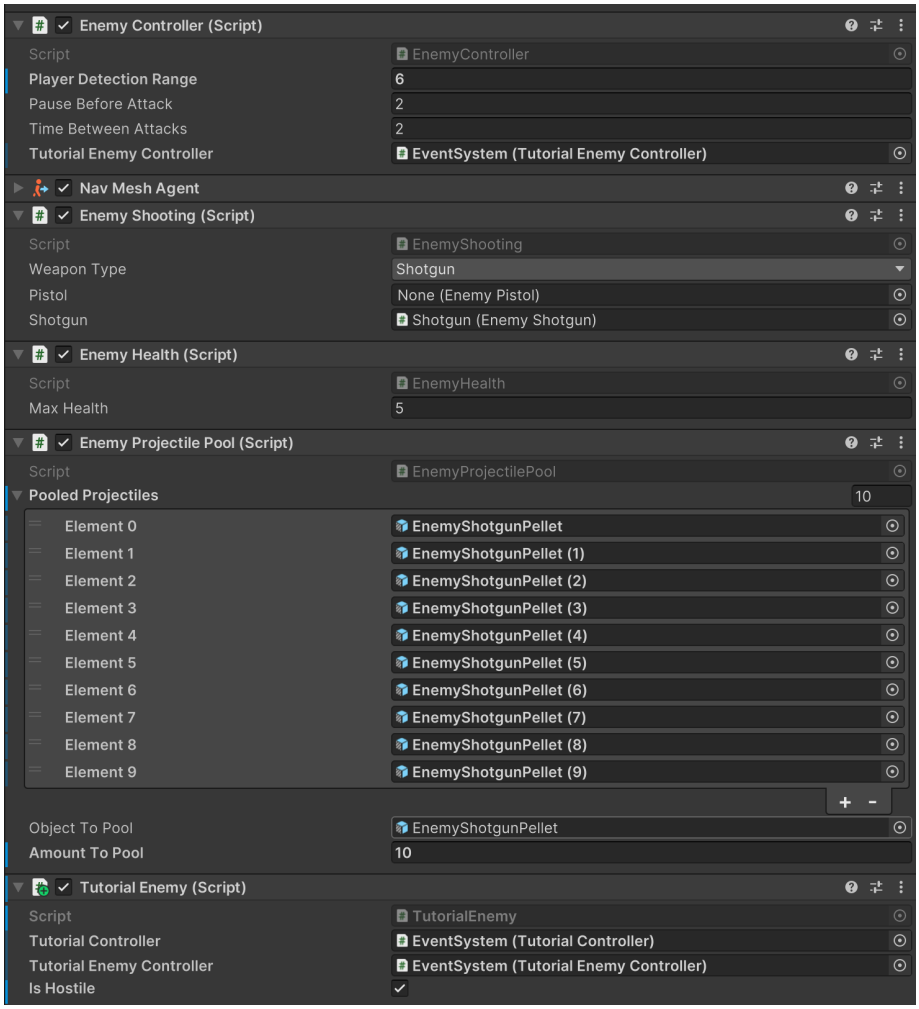


Enemy AI

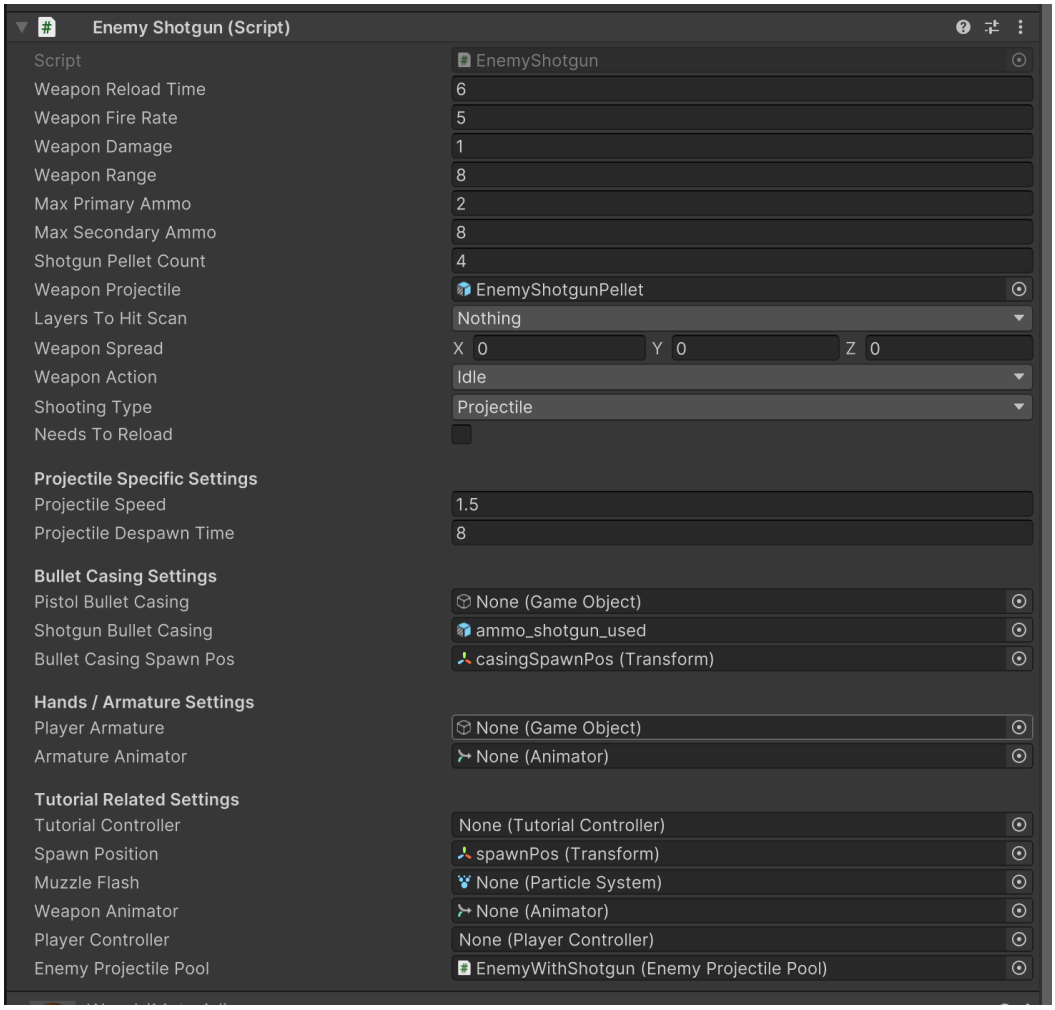
I initially intended for my enemies AI to be far more advanced than it actually turned out to be. I initially intended for my enemies to have the ability to wall run, just like the player could. However, due to several attempts to implement this within unity, it turned out to be far far far more difficult 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 Script

This is the script I have used for my tutorial scene. This is shown via text dialogue on the screen, acting as the game telling the player what to do. I implemented this into my TutorialController.cs Script (TutorialController.cs) via the use of dictionaries. Each section / segment of dialogue is in its own dictionary, allowing me to understand what sections are being displayed on the screen at a certain point of time during gameplay.

Dialogue

These are sorted by their index value, to show the rough order that this segments of dialogue appear as. In some segments i.e. the dialogue displayed when the player arrives at the larger island, is dependent on the actions the player takes when the game tells the player to shoot the non-hostile NPC.

Introduction

Index	Text
0	Welcome to this tutorial!
1	I'll be your teacher today.
2	First, lets familiarize ourselves with this Games Controls.

Input Prompts - Movement Tutorial

Index	Text
0	Press W to move Forward
1	Press S to move Backwards
2	Press A to move Left
3	Press D to move Right
4	Press Space to Jump
5	Movement Tutorial Complete!

Floating Walls Appear

Index	Text
0	Huh, moving, floating walls. Didn't expect that.
1	Try wall running to the next island.
2	Jump between the walls by pressing Space.
3	You did it! Nice work.
4	Time to explore this island.

Player Arrives at Larger Island

Index	Text
0	Oh, a free gun!
1	Press F to pickup the gun.
2	Look nearby the campfire, there's a person.
3	Shoot the person by pressing Mouse1.
4	SHOOT. THEM.
5	It'd help if you actually aimed at the person.
6	Good Job.
7	So uhhh, what now...
8	Come here often?

Enemy Island Appears

Index	Text
0	Ah. Shit.
1	These guys don't seem too happy.
2	Time to kill them I guess.

Upon Killing the New Enemies

Index	Text
0	That's those guys taken care of.
1	Huh, what's that device over there?
2	I should press this button.
3	Hmm. It's doing nothing.
4	Nevermind, spoke too soon.
5	A giant portal! Lets go through it. Nothing bad ever happens with portals.

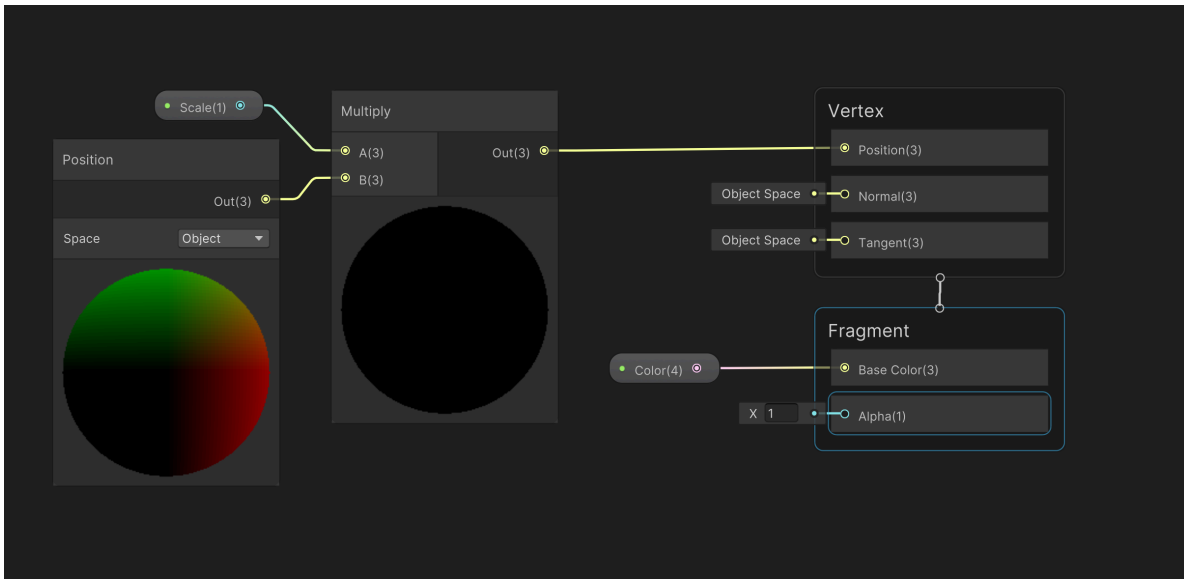
Shader Development

Throughout my project, I experimented slightly with shaders. I have no prior knowledge / experience using Unity's built-in shader graph feature, so I thought that it would be a nice introduction to doing so.

Cel-Shading

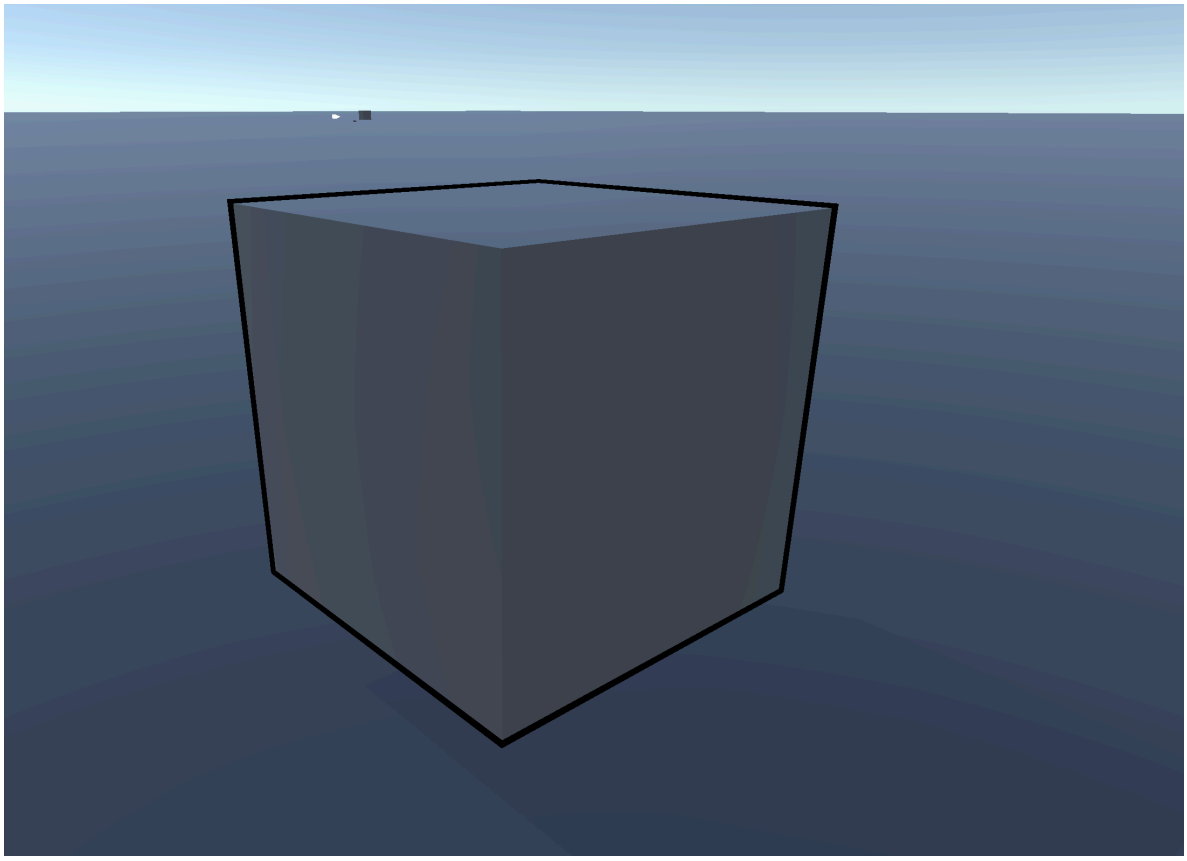
Outline - Initial Testing

I was going to experiment with a cel-shaded look in my project, as a way to compensate for the low-mid poly environment and character models, and to give the scene a further sense of personality. In doing so, I created this shader:



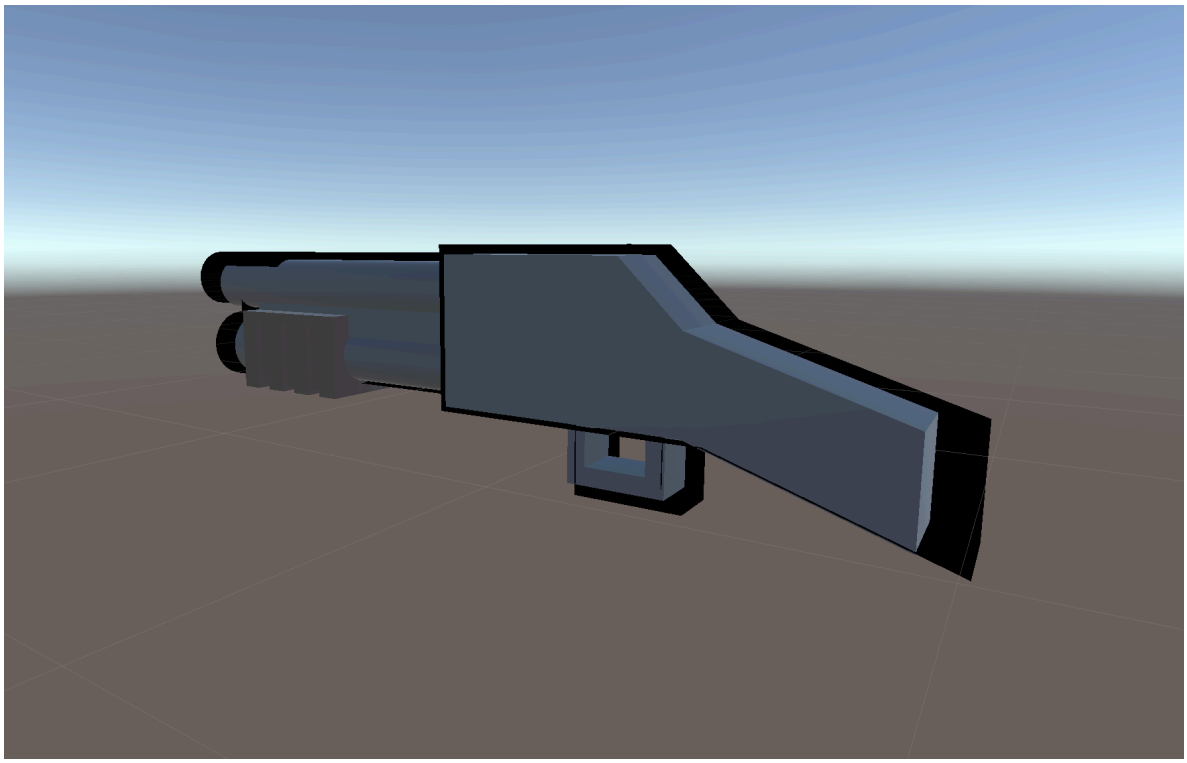
Shader Graph Displaying the Basic Outline Shader

Which resulted in once the material being applied, looked like this:



Unity Inspector Showing a Basic Cube with Outline Applied

This initially looked pretty good, however once being applied to more convex shapes issues would appear:



Unity Inspector Showing issues with the Outline on more Complex Meshes

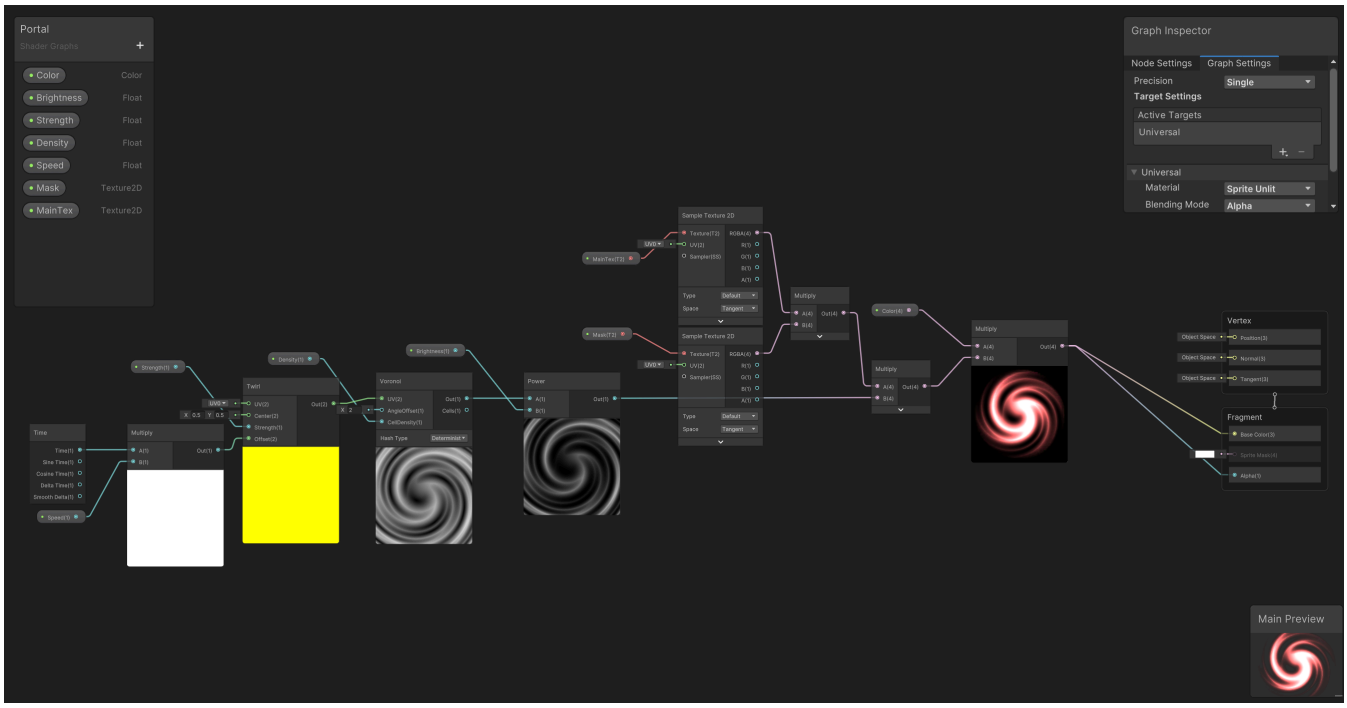
Outline - Conclusion

Due to these issues and my lack of knowledge regarding Unity Shaders (both in HLSL and via Shader Graph), I scrapped this idea early on - the amount of time I would have eventually spent on this was used to further develop my mechanic concepts.

Portal

Portal - Initial Testing

I had the plan to make a portal shader, to simulate a portal appearing throughout my game scenes, as a signifier to the end of each level / scene. Following this tutorial on YouTube (https://www.youtube.com/watch?v=GeeKVPwM5Xw), I ended up making this shader via shader graph, with the end result looking like this:



Once applying this material to a sprite renderer within my scene, the end result looked like this:

Portal - Conclusion

This shader was definitely worth learning how to make, and was a nice further introduction to Unity's Shader Graph system.

User Interface

Start typing here...

Sound Effects

Description

To make my game feel more complete, I added a few sound effects to my game. I would have ideally wanted to implement even more, however due to time constraints I was unable to do so. I sourced all of my sounds from FreeSound (https://freesound.org) - every sound published on this website is published under the Creative Commons licenses that allow the re-use of these sounds within any projects.

Weapon Sounds

Shotgun

I only managed to add one sound to my shotgun, played as a one shot whenever it is fired in-game.

Environment Sounds

Campfire

I added a simple looping campfire sound effect, added to the campfire(s) in my scenes.

Tools Used

Start typing here...

Documentation Development

Start typing here...

Documentation Deployment

Start typing here...

External Assets

Description

These are all the external assets I used throughout the production of my project. This includes software, 3D models, 2D art, sounds and tutorials I followed.

Asset Table

Asset Type	Name	Author	Link
3D Model(s)	Low poly pack: 3D Environments	Andrey Graphics	Unity Asset Store (https://assetstore.unity.com/packages/3d/environments/low-poly-pack-94605)
3D Model(s)	Low Poly Storage Pack: 3D Urban	Broken Vector	Unity Asset Store (https://assetstore.unity.com/packages/p/low-poly-storage-pack-101732)
3D Model(s)	Modular Dungeons Pack	Quaternius	Quaternius.com (https://quaternius.com/packs/modulardungeon.html)
3D Model(s)	Ultimate Buildings Pack	Quaternius	Quaternius.com (https://quaternius.com/packs/ultimatetexturedbuildings.html)
3D Model(s)	Ultimate Furniture Pack	Quaternius	Quaternius.com (https://quaternius.com/packs/ultimatefurniture.html)
3D Model(s)	Survival Pack	Quaternius	Quaternius.com (https://quaternius.com/packs/survival.html)
3D Model(s)	Medieval Village Pack	Quaternius	Quaternius.com (https://quaternius.com/packs/medievalvillage.html)
3D Model(s)	Sci-Fi Modular Gun Pack	Quaternius	Quaternius.com (https://quaternius.com/packs/scifimodularguns.html)

3D Model(s)	Ultimate Modular Ruins Pack	Quaternius	Quaternius.com (https://quaternius.com/packs/ultimatemodularruins.html)
3D Model(s)	Ultimate Modular Sci-Fi Pack	Quaternius	Quaternius.com (https://quaternius.com/packs/ultimatemodularscifi.html)
3D Model(s)	Platformer Game Kit	Quaternius	Quaternius.com (https://quaternius.com/packs/ultimateplatformer.html)
3D Model(s)	Ultimate Fantasy RTS	Quaternius	Quaternius.com (https://quaternius.com/packs/ultimatefantasyrts.html)
3D Model(s)	Cyberpunk Game Kit	Quaternius	Quaternius.com (https://quaternius.com/packs/cyberpunkgamekit.html)
3D Model(s)	Toon Shooter Game Kit	Quaternius	Quaternius.com (https://quaternius.com/packs/toonshootergamekit.html)
3D Model(s)	Animated Zombie Pack	Quaternius	Quaternius.com (https://quaternius.com/packs/animatedzombie.html)
3D Model(s)	Unreal Learning Kit: Games	Unreal Online Learning	UE Marketplace (https://www.unrealengine.com/marketplace/en-US/product/unreal-learning-kit-games)
HDRi Skybo	Kloppenheim 07 (pure sky)	Jarod Guest	PolyHaven (https://polyhaven.com/a/kloppenheim_07_puresky)

x			
HDRi Skybox	Allsky Free - 10 Sky / Skybox Set: 2D Sky	Rpgwhitelock	Unity Asset Store (https://assetstore.unity.com/packages/2d/textures-materials/sky/allsky-free-10-sky-skybox-set-146014)
Scene Plugin	RiderFlow: Level Design	JetBrains	Unity Asset Store (https://polyhaven.com/a/kloppenheim_07_puresky)
Script Library	Lean Tween: Animation Tools	Dented Pixel	Unity Asset Store (https://assetstore.unity.com/packages/tools/animation/leantween-3595)
Script Library	Quick Outline: Particles / Effects	Chris Nolet	Unity Asset Store (https://assetstore.unity.com/packages/tools/particles-effects/quick-outline-115488)
Software	Unity Engine 2022.3.0f1	Unity Technologies	Unity Technologies (https://unity.com/)
Software	JetBrains Rider 2024.1.1	JetBrains	JetBrains (https://www.jetbrains.com/rider/)
Software	JetBrains Writerside 2024.1 EAP	JetBrains	JetBrains (https://www.jetbrains.com/writerside/)
Software	OBS Studio	Open Broadcaster Software	OBSProject (https://obsproject.com/)
Software	Kdenlive	KDE	Kdenlive (https://kdenlive.org/en/trademark-logo/)

Software	Blender 4.1	Blender	Blender (https://www.blender.org/)
Sound Effect	Portal_Idle.wav	couchHero	FreeSound (https://freesound.org/people/couchHero/sounds/168910/)
Sound Effect	FireBurning_v2.wav	pcaeldries	FreeSound (https://freesound.org/people/pcaeldries/sounds/30322/)
Sound Effect	Gunshot 4.wav	ShawnyBoy	FreeSound (https://freesound.org/people/ShawnyBoy/sounds/166191/)
Sound Effect	Gun-Pistol(one shot).wav	Shades	FreeSound (https://freesound.org/people/Shades/sounds/37236/)
Sound Effect	Reload.mp3	LAGtheNoggin	FreeSound (https://freesound.org/people/LAGtheNoggin/sounds/15545/)
Tutorial	How to make awesome scene transitions in unity!	Brackeys	Youtube (https://www.youtube.com/watch?v=CE9VOZivb3I)
Tutorial	Introduction to Object Pooling	Unity Technologies	Unity Learn (https://learn.unity.com/tutorial/introduction-to-object-pooling)
UI Asset	Game Icons	Kenney	Kenney.nl (https://kenney.nl/assets/game-icons)

UI Asset	UI Pack (RPG Expansion)	Kenney	Kenney.nl (https://kenney.nl/assets/ui-pack-rpg-expansion)
UI Asset	Crosshair Pack	Kenney	Kenney.nl (https://kenney.nl/assets/crosshair-pack)
UI Asset	Fantasy UI Borders	Kenney	Kenney.nl (https://kenney.nl/assets/fantasy-ui-borders)
UI Asset	Input Prompts	Kenney	Kenney.nl (https://kenney.nl/assets/input-prompts)
UI Asset	Simple Modern Crosshairs Pack 1: 2D Icons	Visyde Interactives	Unity Asset Store (https://assetstore.unity.com/packages/2d/gui/icons/simple-modern-crosshairs-pack-1-79034)
VFX Particles	Simple FX - Cartoon Particles: VFX Particles	Synty Studios	Unity Asset Store (https://assetstore.unity.com/packages/vfx/particles/simple-fx-cartoon-particles-67834)

Self Evaluation

Start typing here...

Attendance and Engagement

Start typing here...

Builds

Start typing here...

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.

Unused Scripts

Start typing here...

Weapon Scripts

Start typing here...

Enemy Weapon Scripts

Start typing here...

EnemyBaseWeapon.cs

Description

Script

```
namespace Weapons.Enemy
{

    public class EnemyBaseWeapon : BaseWeapon
    {
        public EnemyProjectilePool enemyProjectilePool;

        private void Awake()
        {
            enemyProjectilePool =
transform.root.GetComponentInChildren<EnemyProjectilePool>();
        }
    }
}
```

EnemyPistol.cs

Description

Script

```
using Player;
using UnityEngine;

namespace Weapons.Enemy
{
    public class EnemyPistol : EnemyBaseWeapon
    {
        public override void Fire()
        {
            if (weaponAction != WeaponState.Idle) return;
            if (Physics.Raycast(spawnPosition.position,
spawnPosition.forward * 10, out RaycastHit hit, weaponRange) &&
                shootingType == ShootingType.Hitscan)
            {
                switch (hit.transform.root.tag)
                {
                    case "Player":
                        var collidedPlayer =
hit.transform.root.gameObject;
                        collidedPlayer.GetComponent<PlayerHealth>
().Damage(weaponDamage);
                        break;
                }
            }
            base.Fire();
        }
```

```
    }
}
```

EnemyProjectile.cs

Description

Script

```
using Player;
using UnityEngine;

namespace Weapons.Enemy
{
    public class EnemyProjectile : Projectile
    {
        public override void OnTriggerEnter(Collider other)
        {
            switch (other.transform.tag)
            {
                case "PlayerMesh":
                case "Player":
                    Debug.LogWarning("Player hit!!!");
                    if
(other.transform.root.TryGetComponent<PlayerHealth>(out var
playerHealthScript))
                        playerHealthScript.Damage(ProjectileDamage);
                    Despawn();
                    break;
                case "enemyProjectile":
                case "Enemy":
                    Physics.IgnoreCollision(other, ProjectileCollider);
                    break;
            }
        }
    }
}
```

EnemyProjectilePool.cs

Description

Script

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

namespace Weapons.Enemy
{
    // src = https://learn.unity.com/tutorial/introduction-to-object-pooling
    public class EnemyProjectilePool : MonoBehaviour
    {
        public List<GameObject> pooledProjectiles;

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

        private GameObject _projParent;

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

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

            return null;
        }
    }
}
```

EnemyShotgun.cs

Description

Script

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

namespace Weapons.Enemy
{
    public class EnemyShotgun : EnemyBaseWeapon
    {
        public override void Fire()
        {
            if (weaponAction != WeaponState.Idle) return;
            if (weaponProjectile && shootingType ==
ShootingType.Projectile)
            {
                var pellets = new List<Quaternion>(shotgunPelletCount);
                for (var i = 0; i < shotgunPelletCount; i++)
pellets.Add(Quaternion.Euler(Vector3.zero));
                for (var h = 0; h < shotgunPelletCount; h++)
                {
                    pellets[h] = Random.rotation;
                    var pellet =
enemyProjectilePool.GetPooledProjectile();
                    var pelletScript =
pellet.GetComponent<EnemyProjectile>();
                    if (pellet != null)
                    {
                        pellet.transform.position =
spawnPosition.position;
                        pellet.transform.rotation =
spawnPosition.rotation;
                        pellet.SetActive(true);
```

```
                    }
                    pelletScript.Initialize(weaponDamage,
ProjectileSpeed, ProjectileDespawnTime,
                        spawnPosition.transform.forward +
GetWeaponSpread(spawnPosition.transform));
                }
            }
            base.Fire();
        }
    }
}
```

BaseWeapon.cs

Description

Script

```
using System.Collections;
using Player;
using Tutorial;
using UnityEngine;


namespace Weapons
{
    public class BaseWeapon : MonoBehaviour
    {
        public enum WeaponState
        {
            Firing,
            Reloading,
            Idle,
            NoAmmo
        }

        public enum ShootingType
        {
            Hitscan,
            Projectile
        }

        public float weaponReloadTime;
        public float weaponFireRate;
        public int weaponDamage;
        public int weaponRange;
        public int maxPrimaryAmmo;
        public int maxSecondaryAmmo;
```

```
        public int shotgunPelletCount;
        public GameObject weaponProjectile;
        public LayerMask layersToHitScan;
        public Vector3 weaponSpread;
        public WeaponState weaponAction;
        public ShootingType shootingType;
        public AudioClip weaponSound;
        public AudioClip reloadSound;

        [Header("Projectile Specific Settings")]
        [SerializeField] private float projectileSpeed;
        [SerializeField] private float projectileDespawnTime;

        [Header("Bullet Casing Settings")]
        public GameObject pistolBulletCasing;
        public GameObject shotgunBulletCasing;
        public Transform bulletCasingSpawnPos;

        [Header("Tutorial Related Settings")]
        public TutorialController tutorialController;
        public Transform spawnPosition;
        public PlayerController playerController;


        public int CurrentPrimaryAmmo { get; set; }

        public int CurrentSecondaryAmmo { get; set; }

        protected float ProjectileSpeed => projectileSpeed;

        protected float ProjectileDespawnTime => projectileDespawnTime;

        public virtual void Reload()
        {
            if (weaponAction == WeaponState.Reloading) return;
            if (CurrentPrimaryAmmo == maxPrimaryAmmo) return;
            weaponAction = WeaponState.Reloading;
            switch (CurrentPrimaryAmmo)
```

```
            {
                case <= 0 when CurrentSecondaryAmmo <= 0:
                    weaponAction = WeaponState.NoAmmo;
                    return;
            }

            playerController.audioSource.PlayOneShot(reloadSound);
            var newAmmo = Mathf.Clamp(CurrentPrimaryAmmo +
CurrentSecondaryAmmo, 0, maxPrimaryAmmo);
            StartCoroutine(ReloadCooldown(newAmmo));
            if (playerController)
playerController.canvasScript.Reload(weaponReloadTime);
        }

        public virtual void Fire()
        {
            if (weaponAction != WeaponState.Idle) return;
            if (CurrentPrimaryAmmo <= 0)
                return;
            if (playerController)
                playerController.audioSource.PlayOneShot(weaponSound);
            CurrentPrimaryAmmo--;

            StartCoroutine(WeaponCooldown());
        }

        private IEnumerator WeaponCooldown()
        {
            weaponAction = WeaponState.Firing;
            var cooldown = weaponFireRate / 10f;
            yield return new WaitForSeconds(cooldown);
            weaponAction = WeaponState.Idle;
        }

        private IEnumerator ReloadCooldown(int newPrimary)
        {
            yield return new WaitForSeconds(weaponReloadTime);
            CurrentSecondaryAmmo -= maxPrimaryAmmo - CurrentPrimaryAmmo;
```

```
            CurrentSecondaryAmmo = Mathf.Clamp(CurrentSecondaryAmmo, 0,
maxSecondaryAmmo);
            CurrentPrimaryAmmo = newPrimary;
            CurrentPrimaryAmmo = Mathf.Clamp(CurrentPrimaryAmmo, 0,
maxPrimaryAmmo);
            weaponAction = WeaponState.Idle;
        }

        protected Vector3 GetWeaponSpread(Transform weaponSpawnPos)
        {
            var direction = weaponSpawnPos.forward;
            direction += new Vector3(
                Random.Range(-weaponSpread.x, weaponSpread.x),
                Random.Range(-weaponSpread.y, weaponSpread.y),
                Random.Range(-weaponSpread.z, weaponSpread.z)
            );
            return direction;
        }
    }
}
```

BulletCasing.cs

Description

Script

```
using UnityEngine;

namespace Weapons
{
    [RequireComponent(typeof(Rigidbody))]
    public class BulletCasing : MonoBehaviour
    {
        private Rigidbody _rigidbody;

        private void Start()
        {
            Invoke(nameof(Despawn), 2f);
            if (TryGetComponent(out _rigidbody))
                _rigidbody.velocity =
transform.TransformDirection(Vector3.right * 5f);
        }

        private void Despawn()
        {
            Destroy(gameObject);
        }
    }
}
```

Pistol.cs

Description

Script

```
using AI;
using Tutorial;
using UnityEngine;

namespace Weapons
{
    public class Pistol : BaseWeapon
    {
        public override void Fire()
        {
            if (tutorialController &&
!tutorialController.hasFiredPistolYet)
                tutorialController.EnemyChecks["Fired"] = true;
            if (weaponAction != WeaponState.Idle) return;
            GetWeaponSpread(spawnPosition);

playerController.activeCinemachineBrain.TryGetComponent<Camera>(out var
activeCam);
            var rayOrigin = new Ray(activeCam.transform.position,
activeCam.transform.forward);
            if (Physics.Raycast(rayOrigin, out RaycastHit hit,
weaponRange, layersToHitScan) && shootingType == ShootingType.Hitscan)
            {
                switch (hit.transform.tag)
                {
                    case "EnemyMesh":
                        var collidedEnemyMesh =
hit.transform.parent.gameObject;
                        collidedEnemyMesh.GetComponent<EnemyHealth>
().Damage(weaponDamage);
```

```
                        break;
                    case "Enemy":
                        var collidedEnemy = hit.transform.gameObject;
                        collidedEnemy.GetComponent<EnemyHealth>
().Damage(weaponDamage);
                        break;
                    case "TutorialEnemy":
                        var tutorialEnemy = hit.transform.gameObject;
                        tutorialEnemy.GetComponent<TutorialEnemy>
().Die();
                        break;
                    case null when tutorialController:
                        tutorialController.ActuallyAim();
                        break;
                    default:
                        if (!tutorialController) break;
                        tutorialController.ActuallyAim();
                        break;
                }
            }
            else if (tutorialController)
                tutorialController.ActuallyAim();
            Instantiate(pistolBulletCasing,
bulletCasingSpawnPos.position, transform.rotation);
            base.Fire();
        }

    }

}
```

Projectile.cs

Description

Script

```
using AI;
using UnityEngine;

namespace Weapons
{
    public class Projectile : MonoBehaviour
    {
        protected float ProjectileDamage { get; private set; }
        protected Collider ProjectileCollider;
        private Rigidbody _projectileRigidbody;

        public void Initialize(float damage, float projSpeed, float
despawnTime, Vector3 spawnDir)
        {
            ProjectileDamage = damage;
            ProjectileCollider = GetComponent<Collider>();
            _projectileRigidbody = GetComponent<Rigidbody>();
            Invoke(nameof(Despawn), despawnTime);
            _projectileRigidbody.velocity = (spawnDir +
transform.forward) * projSpeed;
        }

        public void Despawn()
        {
            gameObject.SetActive(false);
        }

        public virtual void OnTriggerEnter(Collider other)
        {
            switch (other.transform.root.tag)
```

```
            {
                case "Player":
                    Physics.IgnoreCollision(other, ProjectileCollider);
                    break;
                case "Enemy":
                    if
(other.transform.root.TryGetComponent<EnemyHealth>(out var
enemyHealthScript))
                        enemyHealthScript.Damage(ProjectileDamage);
                    Despawn();
                    break;
            }
        }
    }
}
```

Shotgun.cs

Description

Script

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

namespace Weapons
{
    public class Shotgun : BaseWeapon
    {
        public override void Fire()
        {
            if (weaponAction != WeaponState.Idle) return;
            var direction = GetWeaponSpread(spawnPosition);
            if (Physics.Raycast(spawnPosition.position, direction, out
RaycastHit hit, weaponRange) &&
                shootingType == ShootingType.Hitscan)
            {
                switch (hit.transform.root.tag)
                {
                    case "Enemy":
                        var collidedEnemy =
hit.transform.root.gameObject;
                        collidedEnemy.GetComponent<EnemyHealth>
().Damage(weaponDamage);
                        break;
                }
            }
            else if (weaponProjectile && shootingType ==
ShootingType.Projectile)
            {
                var pellets = new List<Quaternion>(shotgunPelletCount);
```

```
                for (var i = 0; i < shotgunPelletCount; i++)
pellets.Add(Quaternion.Euler(Vector3.zero));
                for (var h = 0; h < shotgunPelletCount; h++)
                {
                    pellets[h] = Random.rotation;
                    var pellet = Instantiate(weaponProjectile,
spawnPosition.position, spawnPosition.rotation);
                    pellet.transform.rotation =

Quaternion.RotateTowards(pellet.transform.rotation, pellets[h],
weaponSpread.x);
                    var pelletScript = pellet.GetComponent<Projectile>
();
                    pelletScript.Initialize(weaponDamage,
ProjectileSpeed, ProjectileDespawnTime, spawnPosition.transform.forward
+ GetWeaponSpread(spawnPosition.transform));
                }
            }

            Instantiate(shotgunBulletCasing,
bulletCasingSpawnPos.position, transform.rotation);
            base.Fire();
        }

    }
}
```

WeaponScript.cs

Description

Script

```
using UnityEngine;

namespace Weapons
{
    public class WeaponScript : MonoBehaviour
    {

        [SerializeField] private BaseWeapon weapon;
    }
}
```

UI Scripts

Start typing here...

StartScreen.cs

Description

Script

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using TMPro;
using UnityEditor;
using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

namespace UI
{
    // src https://www.youtube.com/watch?v=CE9VOZivb3I
    public class StartScreen : MonoBehaviour
    {

        [SerializeField] private GameObject crossFadeObj;
        [SerializeField] private TextMeshProUGUI loadingText;
        [Header("Buttons")]
        [SerializeField] private Button startBtn;
        [SerializeField] private Button aboutBtn;
        [SerializeField] private Button closeBtn;
        [SerializeField] private Button settingsBtn;
        [SerializeField] private Button applyBtn;
        [SerializeField] private Button aboutCloseBtn;
        [SerializeField] private Button settingsCloseBtn;

        [Header("Settings Menu")]
        [SerializeField] private TMP_Dropdown resolutionDropdown;
```

```
        [SerializeField] private Slider volumeSlider;
        [SerializeField] private Toggle fullscreenToggle;
        [SerializeField] private AudioMixer audioMixer;
        [SerializeField] private float currentVolume;

        [Header("UI Panel Objects")]
        [SerializeField] private GameObject aboutMenu;
        [SerializeField] private GameObject settingsMenu;

        private List<Resolution> _screenResolutions;
        private static readonly int Start1 =
Animator.StringToHash("Start");
        private Animator _crossFadeAnim;
        private int _currentResIndex;
        private bool _hasFaded;
        private void Start()
        {
            loadingText.enabled = false;
            startBtn.onClick.AddListener(StartGame);
            closeBtn.onClick.AddListener(CloseGame);
            aboutBtn.onClick.AddListener(ShowAboutMenu);
            aboutCloseBtn.onClick.AddListener(CloseAboutMenu);
            settingsBtn.onClick.AddListener(ShowSettingsMenu);
            settingsCloseBtn.onClick.AddListener(CloseSettingsMenu);

resolutionDropdown.onValueChanged.AddListener(SetResolution);

fullscreenToggle.onValueChanged.AddListener(ToggleFullscreen);
            volumeSlider.onValueChanged.AddListener(SetVolume);
            applyBtn.onClick.AddListener(SaveSettings);
            if (!crossFadeObj.activeSelf) crossFadeObj.SetActive(true);
            if (aboutMenu.activeSelf) aboutMenu.SetActive(false);
            if (settingsMenu.activeSelf) settingsMenu.SetActive(false);
            InitializeResolutions();
            LoadSettings(_currentResIndex);
            _crossFadeAnim = crossFadeObj.GetComponent<Animator>();
        }
```

```
        private void StartGame()
        {
            LoadNextScene();
        }



        private void ShowAboutMenu()
        {
            ToggleButtonInteractivity(false);
            if (!CheckButtonInteractivity()) return;
            CloseAllMenus();
            aboutMenu.SetActive(true);
        }

        private void CloseAboutMenu()
        {
            ToggleButtonInteractivity(true);
            if (!aboutMenu.activeSelf) return;
            aboutMenu.SetActive(false);
        }

        private void ShowSettingsMenu()
        {
            ToggleButtonInteractivity(true);
            if (!CheckButtonInteractivity()) return;
            CloseAllMenus();
            settingsMenu.SetActive(true);
        }

        private void CloseSettingsMenu()
        {
            ToggleButtonInteractivity(true);
            if (!settingsMenu.activeSelf) return;
            settingsMenu.SetActive(false);
        }

        private bool CheckButtonInteractivity()
```

```
        {
            return !(settingsMenu.activeSelf || aboutMenu.activeSelf);
        }

        private void ToggleButtonInteractivity(bool yn)
        {
            startBtn.interactable = yn;
            aboutBtn.interactable = yn;
            closeBtn.interactable = yn;
            settingsBtn.interactable = yn;
        }


        private static void CloseGame()
        {
            switch (Application.platform)
            {
#if UNITY_EDITOR
            case RuntimePlatform.WindowsEditor:
            case RuntimePlatform.LinuxEditor:
                EditorApplication.ExitPlaymode();
                break;
#endif
            case RuntimePlatform.WindowsPlayer:
            case RuntimePlatform.LinuxPlayer:
                Application.Quit();
                break;
            }
        }

        private void LoadNextScene()
        {

StartCoroutine(LoadLevel(SceneManager.GetActiveScene().buildIndex + 1));
        }


        private void CloseAllMenus()
```

```
        {
            settingsMenu.SetActive(false);
            aboutMenu.SetActive(false);
            ToggleButtonInteractivity(true);
        }

        private void ToggleFullscreen(bool isFullscreen)
        {
            Screen.fullScreen = isFullscreen;
        }


        private void SetResolution(int resolutionIndex)
        {
            var resolution = _screenResolutions[resolutionIndex];
            Screen.SetResolution(resolution.width, resolution.height,
Screen.fullScreen);
        }

        private void SetVolume(float volume)
        {
            audioMixer.SetFloat("Volume", volume);
            currentVolume = volume;
        }

        private void SaveSettings()
        {
            PlayerPrefs.SetInt("FullScreenPreference",
Convert.ToInt32(Screen.fullScreen));
            PlayerPrefs.SetInt("ResolutionPreference",
resolutionDropdown.value);
            PlayerPrefs.SetFloat("VolumePreference", currentVolume);
        }

        private void LoadSettings(int resolutionIndex)
        {
            Screen.fullScreen =
PlayerPrefs.HasKey("FullScreenPreference") &&
```

```
Convert.ToBoolean(PlayerPrefs.GetInt("FullScreenPreference"));

            resolutionDropdown.value =
PlayerPrefs.HasKey("ResolutionPreference")
                ? PlayerPrefs.GetInt("ResolutionPreference")
                : resolutionIndex;

            volumeSlider.value = PlayerPrefs.HasKey("VolumePreference")
? PlayerPrefs.GetFloat("VolumePreference") : 1f;
        }

        private void InitializeResolutions()
        {
            resolutionDropdown.ClearOptions();
            var options = new List<string>();
            _screenResolutions = Screen.resolutions.ToList();
            foreach (var t in _screenResolutions)
            {
                var option = t.width + " x " + t.height;
                options.Add(option);
                if (t.width == Screen.currentResolution.width &
                    t.height == Screen.currentResolution.height)
                {
                    _currentResIndex = _screenResolutions.IndexOf(t);
                }
            }
            resolutionDropdown.AddOptions(options);
            resolutionDropdown.RefreshShownValue();
        }

        private IEnumerator LoadLevel(int levelIndex)
        {
            loadingText.enabled = true;
            _crossFadeAnim.SetTrigger(Start1);
            yield return new WaitForSeconds(1f);
            SceneManager.LoadSceneAsync(levelIndex);
        }
```

```
    }
}
```

DeathScreen.cs

Description

Script

```
using System.Collections;
using TMPro;
using UnityEditor;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

namespace UI
{
    public class DeathScreen : MonoBehaviour
    {

        [SerializeField] private Button retryBtn;
        [SerializeField] private Button quitBtn;
        [SerializeField] private TextMeshProUGUI deathText;
        [SerializeField] private TextMeshProUGUI loadingText;
        [SerializeField] private GameObject crossFadeObj;


        private void Start()
        {
            deathText.enabled = true;
            loadingText.enabled = false;
            retryBtn.onClick.AddListener(RetryGame);
            quitBtn.onClick.AddListener(CloseGame);

            crossFadeObj.GetComponent<Animator>();
        }

        private void RetryGame()
```

```
        {
            LoadPreviousScene();
        }

        private static void CloseGame()
        {
            switch (Application.platform)
            {
#if UNITY_EDITOR
                case RuntimePlatform.WindowsEditor:
                case RuntimePlatform.LinuxEditor:
                    EditorApplication.ExitPlaymode();
                    break;
#endif
                case RuntimePlatform.WindowsPlayer:
                case RuntimePlatform.LinuxPlayer:
                    Application.Quit();
                    break;
            }
        }

        private void LoadPreviousScene()
        {

StartCoroutine(LoadLevel(SceneManager.GetActiveScene().buildIndex - 1));
        }

        private IEnumerator LoadLevel(int levelIndex)
        {
            deathText.enabled = false;
            loadingText.enabled = true;
            retryBtn.gameObject.SetActive(false);
            quitBtn.gameObject.SetActive(false);
            yield return new WaitForSeconds(1f);
            SceneManager.LoadSceneAsync(levelIndex);
        }
```

```
    }
}
```

CanvasScript.cs

Description

Script

```
using System;
using System.Collections;
using Player;
using TMPro;
using Tutorial;
using UnityEngine;
using UnityEngine.UI;

namespace UI
{
    public class CanvasScript : MonoBehaviour
    {
        [Header("Images")]
        [SerializeField] private Image reloadBar;
        [SerializeField] private Image healthBar;

        [Header("Panels")]
        [SerializeField] private GameObject ammoPanel;
        [SerializeField] private GameObject healthPanel;
        [SerializeField] private GameObject enemiesPanel;


        [Header("Text")]
        [SerializeField] private TextMeshProUGUI ammoReporter;
        [SerializeField] private TextMeshProUGUI enemiesToKill;

        [Header("Required Components")]
        [SerializeField] private TutorialEnemyController
tutorialEnemyController;
        [SerializeField] private TutorialController tutorialController;
```

```
        private PlayerController _player;
        private PlayerShooting _playerShooting;
        private PlayerHealth _playerHealth;

        private bool _currentlyReloading;
        private bool _enemyKillChallenge;

        private int _numberOfEnemies;


        private void Start()
        {
            _player =
GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerController
>();
            _playerShooting = _player.playerShooting;
            _playerHealth = _player.playerHealth;
            reloadBar.fillAmount = 0f;
            ammoPanel.SetActive(false);
            enemiesPanel.SetActive(false);
            healthPanel.SetActive(true);
        }

        private void FixedUpdate()
        {
            if (ammoPanel.activeSelf && Math.Abs(reloadBar.fillAmount -
1f) < 0.01f && !_currentlyReloading)
                reloadBar.gameObject.SetActive(false);

            if (_playerShooting.HasWeapon() && !ammoPanel.activeSelf)
                ammoPanel.SetActive(true);

            if (healthPanel.activeSelf)
                healthBar.fillAmount = _playerHealth.CurrentHealth /
_playerHealth.MaxHealth;

            switch (_enemyKillChallenge)
```

```
            {
                case true when !enemiesPanel.activeSelf:
                    enemiesPanel.SetActive(true);
                    break;
                case true when enemiesPanel.activeSelf:
                    enemiesToKill.text =
                        $"Remaining:
\n{tutorialEnemyController.EnemiesRemaining} / {_numberOfEnemies}";
                    break;
                case false when enemiesPanel.activeSelf:
                    enemiesPanel.SetActive(false);
                    break;
            }

            if (_playerShooting.HasWeapon() && ammoPanel.activeSelf)
                ammoReporter.text =
                    $"{_playerShooting.CurrentWeapon.CurrentPrimaryAmmo}
/ {_playerShooting.CurrentWeapon.CurrentSecondaryAmmo}";
        }


        public void Reload(float reloadTime)
        {
            _currentlyReloading = true;
            reloadBar.gameObject.SetActive(true);
            LeanTween.value(reloadBar.gameObject, 0f, 1f,
reloadTime).setOnUpdate(val =>
            {
                var i = reloadBar.fillAmount;
                i = val;
                reloadBar.fillAmount = i;
            }).setOnComplete(() => { _currentlyReloading = false; });
        }

        public void ShowKillChallengeUI(int enemies)
        {
            _numberOfEnemies = enemies;
            StartCoroutine(EnemyKillChallenge());
```

```
        }

        private IEnumerator EnemyKillChallenge()
        {
            _enemyKillChallenge = true;
            yield return new WaitUntil(() =>
tutorialEnemyController.EnemiesRemaining == 0);
            _enemyKillChallenge = false;
            tutorialController.EnemyChallengeComplete();
        }
    }
}
```

Environment Scripts

Start typing here...

Lighting

Start typing here...

LightFlicker.cs

Description

Script

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

namespace Environment.Lighting
{
    public class LightFlicker : MonoBehaviour
    {
        [SerializeField] private Color lightFlickerColour;
        [SerializeField] private float minIntensity;
        [SerializeField] private float maxIntensity;
        [SerializeField] private bool enableFlicker;
        [SerializeField] private int lightSmoothing = 8;


        private float _lastSum;
        private Light _light;
        private Queue<float> _lightQueue;

        private void Start()
        {
            _light = GetComponentInChildren<Light>();

            if (enableFlicker && lightSmoothing > 0)
                _lightQueue = new Queue<float>(lightSmoothing);
        }

        private void FixedUpdate()
        {
            if (!enableFlicker && lightSmoothing > 0) return;
```

```
            while (_lightQueue.Count >= lightSmoothing)
                _lastSum -= _lightQueue.Dequeue();

            var newVal = Random.Range(minIntensity, maxIntensity);
            _lightQueue.Enqueue(newVal);
            _lastSum += newVal;
            _light.intensity = _lastSum / (float)_lightQueue.Count;
        }

        private void Reset()
        {
            _lightQueue.Clear();
            _lastSum = 0;
        }
    }
}
```

FloatingWallController.cs

Description

Script

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

namespace Environment
{
    public class FloatingWallController : MonoBehaviour
    {

        [SerializeField] private List<GameObject> childObjs;

        public void TriggerWallMovement()
        {
            StartCoroutine(MoveObjectsY());
        }

        private IEnumerator MoveObjectsY()
        {
            foreach (var childObj in childObjs)
            {
                yield return new WaitForSeconds(0.35f);
                LeanTween.moveLocalY(childObj, 0, 3);
            }
        }
    }
}
```

OutOfWorldSensor.cs

Description

Script

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace Environment
{
    public class OutOfWorldSensor : MonoBehaviour
    {
        private void OnTriggerEnter(Collider other)
        {
            if (!other.transform.parent.gameObject.CompareTag("Player"))
return;
            SceneManager.LoadSceneAsync("deathScene");

        }

        private void OnTriggerExit(Collider other)
        {
            if (!other.transform.parent.gameObject.CompareTag("Player"))
return;
            SceneManager.LoadSceneAsync("deathScene");
        }

        private void OnTriggerStay(Collider other)
        {
            if (!other.transform.parent.gameObject.CompareTag("Player"))
return;
            SceneManager.LoadSceneAsync("deathScene");
        }
```

```
    }
}
```

Portal.cs

Description

Script

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace Environment
{
    public class Portal : MonoBehaviour
    {
        private void OnTriggerEnter(Collider other)
        {
            if (!other.transform.parent.gameObject.CompareTag("Player"))
return;
            SceneManager.LoadScene("mainScene");
        }
        private void OnTriggerStay(Collider other)
        {
            if (!other.transform.parent.gameObject.CompareTag("Player"))
return;
            SceneManager.LoadScene("mainScene");
        }
    }
}
```

Debugging Scripts

Start typing here...

ShowMoveState.cs

Description

Script

```
using Player;
using TMPro;
using UnityEngine;

namespace Debugging
{
    public class ShowMoveState : MonoBehaviour
    {

        private TextMeshProUGUI _moveStateText;
        private PlayerMovement _playerMovement;

        private void Start()
        {

GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerController
>();
            _moveStateText = GetComponentInChildren<TextMeshProUGUI>();
        }

        private void FixedUpdate()
        {
            _moveStateText.text =
_playerMovement.GetPlayerMovementState().ToString();
        }
    }
}
```

StateReporter.cs

Description

Script

```
using UnityEngine;
using Cameras;
using Player;
using TMPro;

namespace Debugging
{
    public class StateReporter : MonoBehaviour
    {
        [SerializeField] private CameraController cameraController;
        [SerializeField] private PlayerController playerController;

        private TextMeshProUGUI _text;

        private void Start()
        {
            _text = GetComponent<TextMeshProUGUI>();
        }

        private void FixedUpdate()
        {
            _text.text = $"Player State:
{playerController.PlayerFsm.CurrentState}\n" +
                         $"Camera State:
{cameraController.CameraFsm.CurrentState}";
        }
    }
}
```

Camera Scripts

Start typing here...

Camera FSM

Start typing here...

Camera States

Start typing here...

FirstPersonState.cs

Description

Script

```
using Unity.Cinemachine;

namespace Cameras.FSM.States
{
    public class FirstPersonState : CameraState
    {
        private bool _isChangingPerspective;
        public FirstPersonState(string stateName, CameraStateMachine
stateMachine, CameraController cameraController, CinemachineCamera
stateCamera) : base(stateMachine, cameraController, stateCamera)
        {
            StateMachine = stateMachine;
            CameraController = cameraController;
            StateCamera = stateCamera;
        }
        public override void LogicUpdate()
        {
            base.LogicUpdate();
            if (StateMachine.CurrentState.IsChangingPerspective)

StateMachine.ChangeState(CameraController.ThirdPersonState);
        }
    }
}
```

ThirdPersonState.cs

Description

Script

```
using Unity.Cinemachine;

namespace Cameras.FSM.States
{
    public class ThirdPersonState : CameraState
    {

        public ThirdPersonState(string stateName, CameraStateMachine
stateMachine, CameraController cameraController, CinemachineCamera
stateCamera) : base(stateMachine, cameraController, stateCamera)
        {
            StateMachine = stateMachine;
            CameraController = cameraController;
            StateCamera = stateCamera;
        }
        public override void LogicUpdate()
        {
            base.LogicUpdate();

            if (StateMachine.CurrentState.IsChangingPerspective)

StateMachine.ChangeState(CameraController.FirstPersonState);
        }

    }
}
```

CameraStateMachine.cs

Description

Script

```
using Cameras.FSM.States;
using UnityEngine;

namespace Cameras.FSM
{
    public abstract class CameraStateMachine
    {
        private CameraState _initialState;

        public CameraState CurrentState { get; private set; }

        public CameraState PreviousState { get; set; }

        public void Initialize(CameraState startingState)
        {
            CurrentState = startingState;
            startingState.Enter();
        }

        public void ChangeState(CameraState newState)
        {
            PreviousState = CurrentState;
            CurrentState?.Exit();
            CurrentState = newState;
            CurrentState?.Enter();
        }

    }

    public class CameraMachine : CameraStateMachine
```

```
    {
        [HideInInspector] public FirstPersonState FirstPersonState;
        [HideInInspector] public ThirdPersonState ThirdPersonState;
    }
}
```

CameraState.cs

Description

Script

```
using Unity.Cinemachine;

namespace Cameras.FSM
{
    public abstract class CameraState
    {
        protected CameraController CameraController;
        protected CameraStateMachine StateMachine;

        public bool IsChangingPerspective { get; set; }

        protected CinemachineCamera StateCamera { get; set; }


        protected CameraState(CameraStateMachine stateMachine,
CameraController cameraController,
            CinemachineCamera stateCamera)
        {
            StateMachine = stateMachine;
            CameraController = cameraController;
            StateCamera = stateCamera;

CameraController.playerInput.actions["Perspective"].performed += _ =>
PerspectiveChange();
        }

        public void Enter()
        {
            MainCamera.SetActiveCamera(StateCamera);
        }
```

```
        public void HandleInput()
        {
        }

        public virtual void LogicUpdate()
        {
        }

        public void PhysicsUpdate()
        {
        }

        private void PerspectiveChange()
        {
            StateMachine.CurrentState.IsChangingPerspective = true;
        }

        public void Exit()
        {
            IsChangingPerspective = false;
        }
    }
}
```

CinemachineMouseLook.cs

Description

Script

```
using System;
using Unity.Cinemachine;
using UnityEngine;
using UnityEngine.InputSystem;
using Object = UnityEngine.Object;

namespace Cameras
{
    [ExecuteAlways]
    [SaveDuringPlay]
    [AddComponentMenu("Cinemachine/Helpers/Cinemachine Mouse Look")]
    public class CinemachineMouseLook :
InputAxisControllerBase<CinemachineMouseLook.MouseReader>
    {
        // value that is read by the mouse reader class
        private static float _mouseSensitivity;
        // value visible in inspector
        [SerializeField] private float mouseSens;
        // input override
        [SerializeField] private PlayerInput playerInput;

        private void Awake()
        {
            // non-static to static var
            UpdateSensAndSmoothing(mouseSens);
            // attempt to get player input override
            if (!playerInput)
                TryGetComponent(out playerInput);
            if (!playerInput)
                Debug.LogError("Cannot find input component");
```

```
            else
            {
                playerInput.notificationBehavior =
PlayerNotifications.InvokeCSharpEvents;
                playerInput.onActionTriggered += value =>
                {
                    foreach (var controller in Controllers)
                    {
                        controller.Input.ProcessInput(value.action);
                    }
                };
            }
        }

        public void UpdateSensAndSmoothing(float newSens)
        {
            mouseSens = newSens;
            _mouseSensitivity = newSens;
            foreach (var controller in Controllers)
            {
                controller.Driver.AccelTime = (float) (0.1 * (1 /
newSens));
                controller.Driver.DecelTime = (float) (0.1 * (1 /
newSens));
            }
        }

        private void FixedUpdate()
        {
            if (Application.isPlaying)
                UpdateControllers();
        }

        [Serializable]
        public sealed class MouseReader : IInputAxisReader
        {
            public InputActionReference inputActionReference;
            private Vector2 _value;
```

```
            public void ProcessInput(InputAction action)
            {
                if (!inputActionReference ||
inputActionReference.action.id != action.id) return;
                if (action.expectedControlType == "Vector2")
                    _value = action.ReadValue<Vector2>();
                else
                    _value.x = _value.y = action.ReadValue<float>();
            }

            public float GetValue(Object context,
IInputAxisOwner.AxisDescriptor.Hints hint)
            {
                return hint == IInputAxisOwner.AxisDescriptor.Hints.Y
                    ? _value.y * -_mouseSensitivity
                    : _value.x * _mouseSensitivity;
            }
        }

    }
}
```

MainCamera.cs

Description

Script

```
using System;
using Unity.Cinemachine;
using UnityEngine;
using UnityEngine.Serialization;

namespace Cameras
{
    public class MainCamera : MonoBehaviour
    {
        [SerializeField] private CinemachineCamera firstPersonCam;
        [SerializeField] private CinemachineCamera thirdPersonCam;
        [FormerlySerializedAs("_cameraController")] [SerializeField]
private CameraController cameraController;

        [Header("Testing")]
        [SerializeField] private bool isTesting;
        [Range(0, 360)] [SerializeField] private float xRotation;
        [Range(0, 360)] [SerializeField] private float yRotation;
        [Range(0, 360)] [SerializeField] private float zRotation;
        [Range(0, 360)] [SerializeField] private float dutch;
        [Range(0, 250)] [SerializeField] private float fov;
        [SerializeField] private bool lerpFOV;

        public static CameraChanger.CameraModes ActiveCameraMode =>
CameraChanger.GetActiveCamera();

        private static CinemachineCamera _previousCam;
        private static CinemachineCamera _activeCam;

        private CinemachineMouseLook _cinemachineMouseLook;
```

```
        public MainCamera Instance { get; private set; }


        private void Awake()
        {
            if (Instance != null && Instance != this)
                Destroy(gameObject);
            else
            {
                Instance = this;
            }
        }

        public static void SetActiveCamera(CinemachineCamera newCamera)
        {
            if (!_activeCam && !_previousCam) Debug.LogWarning("No
active cam or previous cam assigned");
            else
            {
                _previousCam = _activeCam;
                _previousCam.Priority.Value = 0;
            }
            _activeCam = newCamera;
            _activeCam.Priority.Value = 10;



        }

        public static CameraChanger.CameraModes GetActiveMode()
        {
            return CameraChanger.GetActiveCamera();
        }

        public void SetSensitivity(float sensitivity)
        {
            if (!_cinemachineMouseLook)
                _cinemachineMouseLook =
firstPersonCam.GetComponent<CinemachineMouseLook>();
```

```
            if (!_cinemachineMouseLook)
                throw new Exception("Cannot find the mouse look
component!");
            _cinemachineMouseLook.UpdateSensAndSmoothing(sensitivity);

        }

        public static void DoFov(float endValue, float timeToTake)
        {
            switch (ActiveCameraMode)
            {
                case CameraChanger.CameraModes.FirstPerson:
                    CameraChanger.FirstPersonCam.LerpFirstFOV(endValue,
timeToTake);
                    break;
                case CameraChanger.CameraModes.ThirdPerson:
                    CameraChanger.ThirdPersonCam.LerpThirdFOV(endValue,
timeToTake);
                    break;
                default:
                    throw new ArgumentOutOfRangeException();
            }
        }

        public static void DoTilt(float endValue, float timeToTake)
        {
            switch (ActiveCameraMode)
            {
                case CameraChanger.CameraModes.FirstPerson:

CameraChanger.FirstPersonCam.LerpFirstDutch(endValue, timeToTake);
                    break;
                case CameraChanger.CameraModes.ThirdPerson:

CameraChanger.ThirdPersonCam.LerpThirdDutch(endValue, timeToTake);
                    break;
                default:
                    throw new ArgumentOutOfRangeException();
```

```
            }
        }


    }
}
```

Enemy Scripts

Start typing here...

EnemyController.cs

Description

Script

```
using System.Collections;
using System.Linq;
using Tutorial;
using UnityEngine;
using UnityEngine.AI;

namespace AI
{
    public static class GameObjectExtensions
    {
        public static GameObject GetChildWithTag(this GameObject parent,
string tag)
        {
            var t = parent.transform;
            return (from Transform tr in t where tr.CompareTag(tag)
select tr.gameObject).FirstOrDefault();
        }
    }

    public class EnemyController : MonoBehaviour
    {
        [SerializeField] private float playerDetectionRange;
        [SerializeField] private float pauseBeforeAttack;
        [SerializeField] private float timeBetweenAttacks;
        [SerializeField] private TutorialEnemyController
tutorialEnemyController;
        private Animator _enemyAnimator;
        private bool _canMove = true;
        private EnemyShooting _enemyShooting;
        private float _velocity;
```

```
        private GameObject _player;
        private GameObject _enemyMesh;
        private NavMeshAgent _navMeshAgent;
        private Quaternion _enemyRotation;
        private Transform _enemyTransform;
        private Vector3 _targetPoint;
        private Vector3 _previousPos;

        // ReSharper disable once NotAccessedField.Local
        private TutorialEnemy _tutorialEnemy;
        private static readonly int Velocity =
Animator.StringToHash("velocity");
        public bool IsTutorial { get; private set; }
        public TutorialEnemyController TutorialEnemyManager =>
tutorialEnemyController;


        private void Start()
        {
            _enemyMesh = gameObject.GetChildWithTag("EnemyMesh");
            _player = GameObject.FindGameObjectWithTag("PlayerMesh");
            _navMeshAgent = GetComponent<NavMeshAgent>();
            _enemyShooting = GetComponent<EnemyShooting>();
            _enemyAnimator = GetComponentInChildren<Animator>();
            IsTutorial = TryGetComponent(out _tutorialEnemy);
            if (!IsTutorial) return;
            _navMeshAgent.enabled = false;
            _enemyShooting.enabled = false;
            _canMove = false;
        }

        public void EnableEnemy()
        {
            _navMeshAgent.enabled = true;
            _enemyShooting.enabled = true;
            _canMove = true;
        }
```

```
        private void FixedUpdate()
        {
            if (!_enemyShooting.enabled) return;
            if (_enemyShooting.CurrentWeapon.CurrentPrimaryAmmo == 0)
            {
                _enemyShooting.Reload();
                return;
            }

            if (Vector3.Distance(transform.position,
_player.transform.position) > playerDetectionRange) return;
            if (IsFacingPlayer() && _enemyShooting.CanAttack)
PrepareToShoot();
            if (_targetPoint == _player.transform.position) return;
            _targetPoint = _player.transform.position;
            if (_canMove) _navMeshAgent.SetDestination(_targetPoint);
        }

        private void LateUpdate()
        {
            _velocity = (_enemyMesh.transform.position -
_previousPos).magnitude / Time.deltaTime;
            _previousPos = _enemyMesh.transform.position;
            _enemyAnimator.SetFloat(Velocity, _velocity);
        }

        private bool IsFacingPlayer()
        {
            return Vector3.Dot(transform.forward,
(_player.transform.position - transform.position).normalized) > 0.95f;
        }

        private void PrepareToShoot()
        {
            _navMeshAgent.isStopped = true;
            _canMove = false;
            transform.LookAt(_player.transform);
            StartCoroutine(WaitBeforeAttack());
```

```
        }

        private IEnumerator WaitBeforeAttack()
        {
            if (!_enemyShooting.CanAttack) yield break;
            yield return new WaitForSeconds(pauseBeforeAttack);
            _enemyShooting.Fire();
            _navMeshAgent.isStopped = false;
            _canMove = true;
            StartCoroutine(TimeBeforeAttacks());
        }

        private IEnumerator TimeBeforeAttacks()
        {
            _enemyShooting.CanAttack = false;
            yield return new WaitForSeconds(timeBetweenAttacks);
            _enemyShooting.CanAttack = true;
        }
    }
}
```

EnemyHealth.cs

Description

Script

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

namespace AI
{
    public class EnemyHealth : MonoBehaviour
    {
        [SerializeField] private float maxHealth = 5f;
        private Animator _enemyAnimator;
        private bool _hasDied;
        private Collider _enemyCollider;
        private EnemyShooting _enemyShooting;
        private EnemyController _enemyController;
        private float CurrentHealth { get; set; }
        private NavMeshAgent _navMeshAgent;
        private static readonly int IsDead =
Animator.StringToHash("isDead");

        private void Start()
        {
            CurrentHealth = maxHealth;
            _enemyController = GetComponent<EnemyController>();
            _enemyShooting = GetComponent<EnemyShooting>();
            _enemyAnimator = GetComponentInChildren<Animator>();
            _enemyCollider = GetComponentInChildren<Collider>();
            _navMeshAgent = GetComponent<NavMeshAgent>();
        }

        private void FixedUpdate()
```

```
        {
            if (CurrentHealth <= 0 && !_hasDied) StartCoroutine(Die());
        }
        public void Damage(float value)
        {
            CurrentHealth -= value;
        }

        private IEnumerator Die()
        {
            _hasDied = true;
            if (_enemyController.IsTutorial)

_enemyController.TutorialEnemyManager.EnemyKilled(_enemyController);

            _enemyAnimator.SetBool(IsDead, true);
            Destroy(_navMeshAgent);
            Destroy(_enemyShooting);
            Destroy(_enemyController);
            Destroy(_enemyCollider);
            yield return new WaitForSeconds(5f);
            Destroy(gameObject);
        }
    }
}
```

EnemyShooting.cs

Description

Script

```
using System;
using UnityEngine;
using Weapons.Enemy;

namespace AI
{
    public class EnemyShooting : MonoBehaviour
    {
        private enum WeaponType
        {
            Pistol,
            Shotgun
        }

        [SerializeField] private WeaponType weaponType;
        [SerializeField] private EnemyPistol pistol;
        [SerializeField] private EnemyShotgun shotgun;

        public EnemyBaseWeapon CurrentWeapon { get; private set; }

        public bool CanAttack { get; set; }

        public void Start()
        {
            CanAttack = true;
            switch (weaponType)
            {
                case WeaponType.Pistol:
                    EquipWeapon(pistol);
                    break;
```

```
                case WeaponType.Shotgun:
                    EquipWeapon(shotgun);
                    break;
                default:
                    throw new ArgumentOutOfRangeException();
            }
        }

        private void EquipWeapon(EnemyBaseWeapon newWeapon)
        {
            CurrentWeapon = newWeapon;
            CurrentWeapon.CurrentPrimaryAmmo =
CurrentWeapon.maxPrimaryAmmo;
            CurrentWeapon.CurrentSecondaryAmmo =
CurrentWeapon.maxSecondaryAmmo;

        }

        public void Reload()
        {
            if (!CurrentWeapon) return;
            CurrentWeapon.Reload();
        }

        public void Fire()
        {
            if (!CurrentWeapon || !CanAttack) return;
            CurrentWeapon.Fire();
        }
    }
}
```

Player Scripts

The Scripts that belong to or relate to the Player.

Player FSM

Collection of scripts used for the Players Finite State Machine.

Player States

The State Scripts to be used by the Player's Finite State Machine.

Airborne.cs

Description

This script controls the Airborne state of the player.

Script

```
using UnityEngine;

namespace Player.FSM.States
{
    public class Airborne : FsmState
    {
        private float _gravityValue;
        private float _playerSpeed;
        private bool _isGrounded;
        private bool _checkForWalls;
        private bool _leftWall;
        private bool _rightWall;
        private bool _canWallRun;
        private RaycastHit _leftWallHit;
        private RaycastHit _rightWallHit;
        private Vector2 _movementInput;
        private Vector3 _playerVelocity;
        private Vector3 _verticalVelocity;
        private LayerMask _whatIsWall;
        private float _maxWallDistance;
        private Transform PlayerTransform => Character.PlayerTransform;

        public Airborne(string stateName, PlayerController
playerController, FiniteStateMachine stateMachine) : base(stateMachine,
playerController)
        {
            Character = playerController;
            StateMachine = stateMachine;
```

```
        }

        public override void Enter()
        {
            base.Enter();
            _isGrounded = false;
            _leftWall = false;
            _rightWall = false;
            _playerSpeed = Character.PlayerSpeed;
            _gravityValue = Character.PlayerGravity;
            _verticalVelocity = Vector3.zero;
            _checkForWalls = Character.checkForWallsWhenAirborne;
            _whatIsWall = Character.WhatIsWall;
            _maxWallDistance = Character.MaxWallDistance;

        }

        public override void HandleInput()
        {
            base.HandleInput();
            _movementInput = MoveAction.ReadValue<Vector2>();
            _playerVelocity = (PlayerTransform.right * _movementInput.x
+
                               PlayerTransform.forward *
_movementInput.y) * _playerSpeed;
        }

        public override void LogicUpdate()
        {
            base.LogicUpdate();

            switch (_isGrounded)
            {
                case true when _movementInput is not {x: 0, y: 0}:
                    StateMachine.ChangeState(Character.WalkingState);
                    break;
                case true when _movementInput is {x: 0, y:0}:
                    StateMachine.ChangeState(Character.IdleState);
```

```
                    break;
                case false when (_leftWall || _rightWall) &&
_canWallRun:
                    StateMachine.ChangeState(Character.WallRunState);
                    break;
            }


        }

        public override void PhysicsUpdate()
        {
            base.PhysicsUpdate();

            _isGrounded = Character.isGrounded;
            Character.characterController.Move(_playerVelocity *
Time.deltaTime + _verticalVelocity * Time.deltaTime);
            if (!_isGrounded) _verticalVelocity.y += _gravityValue *
Time.deltaTime;
            if (!_checkForWalls) return;
            var right = PlayerTransform.right;
            var position = PlayerTransform.position;
            _rightWall = Physics.Raycast(position, right, out
_rightWallHit, _maxWallDistance, _whatIsWall);
            _leftWall = Physics.Raycast(position, -right, out
_leftWallHit, _maxWallDistance, _whatIsWall);

            if ((!_leftWall && !_rightWall) || _movementInput is {x: 0,
y: 0} || _isGrounded) return;
            if (_leftWall && !_rightWall)
            {
                Character.leftWall = true;
                Character.rightWall = false;
                Character.LeftWallHit = _leftWallHit;
            }
            else if (_rightWall && !_leftWall)
            {
                Character.rightWall = true;
```

```
                Character.leftWall = false;
                Character.RightWallHit = _rightWallHit;
            }
            _canWallRun = true;
        }

        public override void Exit()
        {
            base.Exit();
            _isGrounded = true;
            Character.canJump = false;
            Character.StartCoroutine(PlayerController.ActionCooldown(()
=> Character.canJump = true, Character.JumpCooldown));
            Character.StartCoroutine(PlayerController.ActionCooldown(()
=> { },
                Character.WallRunCooldown));
            Character.jumpingFromLeftWall = false;
            Character.jumpingFromRightWall = false;
            Character.checkForWallsWhenAirborne = false;
        }
    }
}
```

Public Methods

Airborne()

Stores a reference to the player controller script and the state machine controller.

Enter()

Sets variable values to match the values on the Player Controller script, while also zeroing variables that may have been true before entering this state.

HandleInput()

Reads for Player Movement input and updates the player's velocity accordingly.

LogicUpdate()

Checks if the player is grounded, and then updates the player state according to the movement input and if there are any walls detected.

PhysicsUpdate()

Updates if the player is grounded, moves the player, calculates vertical velocity, checks for walls nearby, and determines whether the player can wall run.

Exit()

Applies cooldowns and zeroes values.

Variables

gravityValue

Reference to the player gravity.

playerSpeed

Reference to the player's speed.

isMoving

Is the player moving.

isGrounded

Is the player grounded.

checkForWalls

Should the script check for walls while the player is airborne.

_leftWall

Is left wall detected.

_rightWall

Is right wall detected.

canWallRun

Can the player wall run.

_leftWallHit

Reference to the wall detected to the left of the player.

_rightWallHit

Reference to the wall detected to the right of the player.

movementInput

The X and Y values of the players movement input.

playerVelocity

The X, Y and Z values of the players velocity.

verticalVelocity

The X, Y and Z values of the players vertical velocity.

_whatIsWall

The layer mask determining what objects can be wall ran on.

_maxWallDistance

The max length that can be reached between a player and a wall.

PlayerTransform

A reference to the transform of the player's Character Controller component.

Idle.cs

Description

This script controls the Idle state of the player.

Script

```
using Tutorial;
using UnityEngine;

namespace Player.FSM.States
{
    public class Idle : FsmState
    {
        private bool _isJumping;
        private bool _isSliding;
        private bool _isGrounded;
        private bool _isMoving;
        private Vector2 _movementInput;
        private Vector3 _playerVelocity;
        private Vector3 _verticalVelocity;
        private Transform PlayerTransform => Character.PlayerTransform;


        public Idle(string name, PlayerController playerController,
FiniteStateMachine stateMachine) : base(stateMachine, playerController)
        {
            Character = playerController;
            StateMachine = stateMachine;
        }

        public override void Enter()
        {
            base.Enter();
            _isMoving = false;
```

```
            _isJumping = false;
            _isSliding = false;
            _isGrounded = true;
            _playerVelocity = Vector3.zero;
            _verticalVelocity = Vector3.zero;
        }

        public override void Tick(float deltaTime)
        {
            throw new System.NotImplementedException();
        }

        public override void HandleInput()
        {
            base.HandleInput();
            _isJumping = JumpAction.IsPressed();
            if (_isJumping && Character.IsTutorial &&
TutorialController.IntroComplete() &&
                TutorialController.nextKeyToPress ==
TutorialController.NextKeyPress.Jump)
                TutorialController.TutorialChecks["Jump"] = true;
            if (_movementInput is not { x: 0, y: 0 })
            {
                _isMoving = true;
                if (!Character.IsTutorial) return;
                switch (_movementInput)
                {
                    case { x: 0, y: > 0 }:
                        if (!TutorialController.IntroComplete() &&
TutorialController.nextKeyToPress ==
                            TutorialController.NextKeyPress.Forward)
break;
                        TutorialController.TutorialChecks["Forward"] =
true;
                        break;
                    case { x: 0, y: < 0 }:
                        if (!TutorialController.IntroComplete() &&
TutorialController.nextKeyToPress !=
```

```
                            TutorialController.NextKeyPress.Backwards)
break;
                        TutorialController.TutorialChecks["Backwards"] =
true;
                        break;
                    case { x: > 0, y: 0 }:
                        if (!TutorialController.IntroComplete() &&
TutorialController.nextKeyToPress !=
                            TutorialController.NextKeyPress.Right)
break;
                        TutorialController.TutorialChecks["Right"] =
true;
                        break;
                    case { x: < 0, y: 0 }:
                        if (!TutorialController.IntroComplete() &&
                            TutorialController.nextKeyToPress !=
TutorialController.NextKeyPress.Left) break;
                        TutorialController.TutorialChecks["Left"] =
true;
                        break;
                }
            }

            _movementInput = MoveAction.ReadValue<Vector2>();
            _playerVelocity = (PlayerTransform.right * _movementInput.x
+
                               PlayerTransform.forward *
_movementInput.y) * PlayerSpeed;
        }

        public override void LogicUpdate()
        {
            base.LogicUpdate();

            if (_isJumping && Character.canJump)
                StateMachine.ChangeState(Character.JumpingState);
            if (_isMoving)
                StateMachine.ChangeState(Character.WalkingState);
```

```
            if (_isSliding && Character.canSlide)
                StateMachine.ChangeState(Character.SlidingState);
        }

        public override void PhysicsUpdate()
        {
            base.PhysicsUpdate();

            _verticalVelocity.y += GravityValue * Time.deltaTime;
            _isGrounded = Character.isGrounded;
            if (_isGrounded && _verticalVelocity.y < 0)
                _verticalVelocity.y = 0f;

            Character.characterController.Move(_playerVelocity *
Time.deltaTime + _verticalVelocity * Time.deltaTime);
        }
    }
}
```

Public Methods

Idle()

Stores a reference to the player controller script and the state machine controller.

Enter()

Sets variable values to match the values on the Player Controller Script, while zeroing the players velocity and vertical velocity.

HandleInput()

Checks if the player is jumping, if the player is in the tutorial level then confirm the player has pressed jump for the input prompt. If the player is pressing any of the

movement keys, and is in tutorial, while the introduction is not complete, then set the relevant input checks to true. Calculates the player's velocity.

LogicUpdate()

If Jumping then change to jumping state, if moving then change to the walking state, if sliding then change to the sliding state.

PhysicsUpdate()

Apply gravity, update if the player is grounded, zeroes the gravity if grounded, moves the player.

Variables

isJumping

Is the player jumping.

isSliding

Is the player sliding.

isGrounded

Is the player grounded.

isMoving

Is the player moving.

movementInput

X and Y Values of the players movement input.

playerVelocity

X, Y and Z Values of the players velocity.

verticalVelocity

X, Y and Z values of the players vertical velocity.

PlayerTransform

A reference to the transform of the player's Character Controller component.

Jumping.cs

Description

This script controls the Jumping state of the player.

Script

```
using Unity.Cinemachine;
using UnityEngine;

namespace Player.FSM.States
{
    public class Jumping : FsmState
    {
        private float _gravityValue;
        private float _playerSpeed;
        private float _playerJumpHeight;
        private bool _isGrounded;
        private Vector2 _mouseInput;
        private Vector2 _movementInput;
        private Vector3 _playerVelocity;
        private Vector3 _verticalVelocity;
        private float _mouseX;
        private float _mouseY;
        private float _xRotation;
        private float _maxWallDistance;
        private LayerMask _whatIsWall;
        private Vector3 _targetRotation;
        private CinemachineCamera _thirdPersonCam;
        private CinemachineCamera _firstPersonCam;
        private RaycastHit _leftWallHit;
        private RaycastHit _rightWallHit;
        private bool _leftWall;
        private bool _rightWall;
        private bool _canWallRun;
```

```
        private Transform PlayerTransform => Character.PlayerTransform;


        public Jumping(string stateName, PlayerController
playerController, FiniteStateMachine stateMachine) : base(stateMachine,
playerController)
        {
            Character = playerController;
            StateMachine = stateMachine;
        }

        public override void Enter()
        {
            base.Enter();
            _canWallRun = false;
            _isGrounded = false;
            _playerSpeed = Character.PlayerSpeed;
            _gravityValue = Character.PlayerGravity;
            _playerJumpHeight = Character.JumpHeight;
            _verticalVelocity = Vector3.zero;
            _maxWallDistance = Character.MaxWallDistance;
            _whatIsWall = Character.WhatIsWall;
            Jump();
        }

        private void Jump()
        {
            _verticalVelocity.y = Mathf.Sqrt(-2f * _playerJumpHeight *
_gravityValue);
        }

        public override void HandleInput()
        {
            base.HandleInput();


            SlideAction.IsPressed();
            _movementInput = MoveAction.ReadValue<Vector2>();
```

```
            _playerVelocity = (PlayerTransform.right * _movementInput.x
+
                               PlayerTransform.forward *
_movementInput.y) * _playerSpeed;
        }

        public override void LogicUpdate()
        {
            base.LogicUpdate();

            switch (_isGrounded)
            {
                case false when (_leftWall || _rightWall) &&
_canWallRun:
                    StateMachine.ChangeState(Character.WallRunState);
                    break;
                case true when _movementInput is not { x: 0, y: 0 } &&
_verticalVelocity.y < 0:
                    StateMachine.ChangeState(Character.IdleState);
                    break;
                case true when _movementInput is { x: 0, y: 0 } &&
_verticalVelocity.y < 0:
                    StateMachine.ChangeState(Character.WalkingState);
                    break;
                case false when _verticalVelocity.y <= 0:
                    StateMachine.ChangeState(Character.AirborneState);
                    break;
            }
        }

        public override void PhysicsUpdate()
        {
            base.PhysicsUpdate();
            _isGrounded = Character.isGrounded;
            Character.characterController.Move(_playerVelocity *
Time.deltaTime + _verticalVelocity * Time.deltaTime);
            if (!_isGrounded) _verticalVelocity.y += _gravityValue *
Time.deltaTime;
```

```
            var right = PlayerTransform.right;
            var position = PlayerTransform.position;
            _rightWall = Physics.Raycast(position, right, out
_rightWallHit, _maxWallDistance, _whatIsWall);
            _leftWall = Physics.Raycast(position, -right, out
_leftWallHit, _maxWallDistance, _whatIsWall);

            if ((!_leftWall && !_rightWall) || _movementInput is { x: 0,
y: 0 } || _isGrounded) return;
            if (_leftWall && !_rightWall)
            {
                Character.leftWall = true;
                Character.rightWall = false;
                Character.LeftWallHit = _leftWallHit;
            }
            else if (_rightWall && !_leftWall)
            {
                Character.rightWall = true;
                Character.leftWall = false;
                Character.RightWallHit = _rightWallHit;
            }

            _canWallRun = true;
        }

        public override void Exit()
        {
            base.Exit();
            _isGrounded = true;

            if (_leftWall && !_rightWall)
            {
                Character.leftWall = true;
                Character.rightWall = false;
                Character.LeftWallHit = _leftWallHit;
            }
```

```
            else if (_rightWall && !_leftWall)
            {
                Character.rightWall = true;
                Character.leftWall = false;
                Character.RightWallHit = _rightWallHit;
            }
        }
    }
}
```

Public Methods

Jumping()

Stores a reference to the player controller script and the state machine controller.

Enter()

Sets variable values to match the values on the Player Controller script, while also resetting variable values that may have been different before entering this state, then calls the Jump function.

HandleInput()

Reads for Player Movement input and then updates the player's velocity accordingly.

LogicUpdate()

Checks if the player is grounded, and then changes the state depending on this alongside the players movement input and vertical velocity, while also making checks if the player can wall run and if either walls to the left or the right of the player has been detected.

PhysicsUpdate()

Updates if the player is grounded, moves the player, calculates the players vertical velocity, ray casts to the left and right of the player, if walls are detected, the player is moving & not grounded, then store references to the walls detected alongside the RaycastHit variable and indicate that the player is able to wall run.

Exit()

Sets the player to grounded, no longer jumping, and checks if walls are marked as true and update the values on the Player Controller script.

Private Methods

Jump()

Sets the Y axis of the vertical velocity vector to the calculated value dependant on the players jump height and gravity.

Variables

gravityValue

Reference to the player gravity.

playerSpeed

Reference to the player's speed.

playerJumpHeight

Reference to the player's jump height.

isJumping

Is the player jumping.

isSliding

Is the player sliding.

isGrounded

Is the player grounded.

isMoving

Is the player moving.

mouseInput

The X and Y values of the player's mouse delta.

movementInput

The X and Y values of the players movement input.

playerVelocity

The X, Y and Z values of the players velocity.

verticalVelocity

The X, Y and Z values of the players vertical velocity

_maxWallDistance

The max length that can be reached between a player and a wall.

_whatIsWall

The layer mask determining what objects can be wall ran on.

_leftWallHit

Reference to the wall detected to the left of the player.

_rightWallHit

Reference to the wall detected to the right of the player.

_leftWall

Is left wall detected.

_rightWall

Is right wall detected.

_canWallRun

Can the player wall run.

PlayerTransform

A reference to the transform of the player's Character Controller component.

Sliding.cs

Description

This script controls the Sliding state of the player.

Script

```
using System.Collections;
using UnityEngine;

namespace Player.FSM.States
{
    public class Sliding : FsmState
    {
        private float _maxSlideTime;
        private float _slideForce;
        private float _slideYScale;
        private Transform _playerTransform;
        private float _playerSpeed;
        private Vector2 _movementInput;
        private Vector3 _playerVelocity;
        private Vector3 _slideVelocity;
        private bool _isSliding;
        private bool _isJumping;
        private Coroutine _slideTimer;
        private float _startYScale;

        public Sliding(string stateName, PlayerController
playerController, FiniteStateMachine stateMachine) : base(stateMachine,
playerController)
        {
            Character = playerController;
            StateMachine = stateMachine;
        }
```

```
        public override void Enter()
        {
            base.Enter();

            _isSliding = true;
            _isJumping = false;
            _maxSlideTime = Character.MaxSlideTime;
            _slideForce = Character.SlideForce;
            _slideYScale = Character.SlideYScale;
            _playerTransform = Character.PlayerTransform;
            _playerSpeed = Character.PlayerSpeed;
            _slideTimer = Character.StartCoroutine(SlideTimer());
            _startYScale = _playerTransform.localScale.y;

            var localScale = _playerTransform.localScale;
            localScale = new Vector3(localScale.x, _slideYScale,
localScale.y);
            _playerTransform.localScale = localScale;
        }

        public override void HandleInput()
        {
            base.HandleInput();

            _movementInput = MoveAction.ReadValue<Vector2>();
            _playerVelocity = (_playerTransform.right * _movementInput.x
+
                               _playerTransform.forward *
_movementInput.y) * _playerSpeed;


            _isJumping = JumpAction.IsPressed();
        }

        public override void LogicUpdate()
        {
            base.LogicUpdate();
```

```
            switch (_isSliding)
            {
                case false when _movementInput is { x: 0, y: 0 }:
                    StateMachine.ChangeState(Character.IdleState);
                    break;
                case false when _movementInput is not { x: 0, y: 0 }:
                    StateMachine.ChangeState(Character.WalkingState);
                    break;
            }

            if (_isJumping)
                StateMachine.ChangeState(Character.JumpingState);
        }

        public override void PhysicsUpdate()
        {
            base.PhysicsUpdate();
            _slideVelocity = _playerVelocity.normalized * _slideForce;

            if (!_isSliding) return;
            Character.characterController.Move(_slideVelocity *
Time.deltaTime);
        }

        public override void Exit()
        {
            base.Exit();
            _isSliding = false;
            Character.StopCoroutine(_slideTimer);
            Character.canSlide = false;
            Character.StartCoroutine(PlayerController.ActionCooldown(()
=> Character.canSlide = true,
                Character.SlideCooldown));
            var localScale = _playerTransform.localScale;
            localScale = new Vector3(localScale.x, _startYScale,
localScale.z);
            _playerTransform.localScale = localScale;
```

```
        }

        private IEnumerator SlideTimer()
        {
            yield return new WaitForSeconds(_maxSlideTime);
            _isSliding = false;
        }
    }
}
```

Public Methods

Sliding()

Stores a reference to the player controller script and the state machine controller.

Enter()

Sets the variable values to match the values on the Player Controller script, while also setting the scaling the Y axis of the player by the slide Y scale.

HandleInput()

Reads for Player Movement input and updates the player's velocity accordingly. Also checks if the player is jumping.

LogicUpdate()

Checks if the player is still sliding and the movement input, changes the current state accordingly.

PhysicsUpdate()

Calculates the slide velocity of the player, them moves the player based on the value of the slide velocity.

Exit()

Zeroes values, sets the sliding cooldown, and then restores the players scale on the Y axis to the value it was set to beforehand.

SlideTimer()

Waits for the max duration the player is able to slide for. Then updates a variable to indicate that the player is no longer sliding.

Walking.cs

Description

This script controls the Walking state of the player.

```
using UnityEngine;

namespace Player.FSM.States
{
    public class Walking : FsmState
    {
        private float _gravityValue;
        private float _playerSpeed;
        private bool _isJumping;
        private bool _isSliding;
        private bool _isGrounded;
        private bool _isMoving;
        private Vector2 _movementInput;
        private Vector3 _playerVelocity;
        private Vector3 _verticalVelocity;
        private Transform PlayerTransform => Character.PlayerTransform;


        public Walking(string stateName, PlayerController
playerController, FiniteStateMachine stateMachine) : base(stateMachine,
playerController)
        {
            Character = playerController;
            StateMachine = stateMachine;
        }

        public override void Enter()
        {
            base.Enter();
```

```
            _isMoving = true;
            _isJumping = false;
            _isSliding = false;
            _isGrounded = true;
            _playerSpeed = Character.PlayerSpeed;
            _gravityValue = Character.PlayerGravity;
        }

        public override void HandleInput()
        {
            base.HandleInput();

            _isJumping = JumpAction.IsPressed();
            _isSliding = SlideAction.IsPressed();

            if (_movementInput is { x: 0, y: 0 })
                _isMoving = false;
            _movementInput = MoveAction.ReadValue<Vector2>();
            _playerVelocity = (PlayerTransform.right * _movementInput.x
+
                               PlayerTransform.forward *
_movementInput.y) * _playerSpeed;
        }

        public override void LogicUpdate()
        {
            base.LogicUpdate();


            if (_isJumping && Character.canJump)
                StateMachine.ChangeState(Character.JumpingState);
            if (!_isMoving)
                StateMachine.ChangeState(Character.IdleState);
            if (_isSliding && Character.canSlide)
                StateMachine.ChangeState(Character.SlidingState);
        }
```

```
        public override void PhysicsUpdate()
        {
            base.PhysicsUpdate();
            _verticalVelocity.y += _gravityValue * Time.deltaTime;
            _isGrounded = Character.isGrounded;

            if (_isGrounded && _verticalVelocity.y < 0)
                _verticalVelocity.y = 0f;

            Character.characterController.Move(_playerVelocity *
Time.deltaTime + _verticalVelocity * Time.deltaTime);
        }
    }
}
```

Public Methods

Walking()

Stores a reference to the player controller script and the state machine controller.

Enter()

Sets variable values to match the values on the Player Controller script, while also resetting variable values that may be different before entering this state.

HandleInput()

Reads for Player Movement input and updates the player's velocity. Also checks if the player is jumping or sliding.

LogicUpdate()

If Jumping and can jump then change to the jumping state, if not moving then change to idle state, if sliding and can slide then go to the sliding state.

PhysicsUpdate()

Applies gravity to the players vertical velocity and updates the value to check if the player is grounded. If grounded then zero the vertical velocity. Moves the player.

Variables

gravityValue

Reference to the player gravity.

playerSpeed

Reference to the player's speed.

isJumping

Is the player jumping.

isSliding

Is the player sliding.

isGrounded

Is the player grounded.

isMoving

Is the player moving.

movementInput

The X and Y values of the players movement input.

playerVelocity

The X, Y and Z values of the players velocity.

verticalVelocity

The X, Y and Z values of the players vertical velocity.

PlayerTransform

A reference to the transform of the player's Character Controller component.

WallJumping.cs

Description

This script controls the Wall Jumping state of the player.

Script

```
using UnityEngine;

namespace Player.FSM.States
{
    public class WallJumping : FsmState
    {
        private float _gravityValue;
        private float _playerSpeed;
        private float _playerJumpHeight;
        private float _wallJumpSideForce;
        private float _wallJumpUpForce;
        private bool _isJumping;
        private bool _isGrounded;
        private bool _isWallRunning;
        private Vector2 _movementInput;
        private Vector3 _playerVelocity;
        private Vector3 _verticalVelocity;
        private RaycastHit _leftWallHit;
        private RaycastHit _rightWallHit;
#pragma warning disable CS0414 // Field is assigned but its value is
never used
        private bool _leftWall;
#pragma warning restore CS0414 // Field is assigned but its value is
never used
        private bool _rightWall;
        private Transform PlayerTransform => Character.PlayerTransform;

        public WallJumping(PlayerController playerController,
```

```
FiniteStateMachine stateMachine) : base(
            stateMachine, playerController)
        {
            Character = playerController;
            StateMachine = stateMachine;
        }

        public override void Enter()
        {
            base.Enter();

            _isGrounded = false;
            _playerSpeed = Character.PlayerSpeed;
            _gravityValue = Character.PlayerGravity;
            _wallJumpUpForce = Character.WallJumpUpForce;
            _wallJumpSideForce = Character.WallJumpSideForce;


            if (Character.jumpingFromRightWall &&
!Character.jumpingFromLeftWall)
            {
                _rightWall = true;
                _leftWall = false;
                _rightWallHit = Character.JumpingRightWallHit;
            }
            else if (Character.jumpingFromLeftWall &&
!Character.jumpingFromRightWall)
            {
                _leftWall = true;
                _rightWall = false;
                _leftWallHit = Character.JumpingLeftWallHit;
            }
            WallJump();
        }

        public override void HandleInput()
        {
            base.HandleInput();
```

```
            _movementInput = MoveAction.ReadValue<Vector2>();
            _playerVelocity = (PlayerTransform.right * _movementInput.x
+
                               PlayerTransform.forward *
_movementInput.y) * _playerSpeed;
        }

        public override void PhysicsUpdate()
        {
            base.PhysicsUpdate();
            _isGrounded = Character.isGrounded;
            Character.characterController.Move(_playerVelocity *
Time.deltaTime + _verticalVelocity * Time.deltaTime);
            if (!_isGrounded) _verticalVelocity.y += _gravityValue *
Time.deltaTime;
        }

        public override void LogicUpdate()
        {
            base.LogicUpdate();

            if (_verticalVelocity.y <= 0)
                StateMachine.ChangeState(Character.AirborneState);
        }

        private void WallJump()
        {
            var wallNormal = _rightWall ? _rightWallHit.normal :
_leftWallHit.normal;
            var playerForceToApply = PlayerTransform.up *
_wallJumpUpForce + wallNormal * _wallJumpSideForce;
            _verticalVelocity = playerForceToApply;

        }

        public override void Exit()
        {
```

```
            base.Exit();
            Character.canWallJump = false;
            Character.StartCoroutine(PlayerController.ActionCooldown(()
=> Character.canWallJump = true,
                Character.WallJumpCooldown));
            Character.canJump = false;
            Character.StartCoroutine(PlayerController.ActionCooldown(()
=> Character.canJump = true,
                Character.JumpCooldown));
            Character.checkForWallsWhenAirborne = true;
            _leftWall = false;
            _rightWall = false;
        }
    }
}
```

Public Methods

WallJumping()

Stores a reference to the player controller script and the state machine controller.

Enter()

Sets variable values to match the values on the Player Controller script, whilst also checking which wall the character is jumping from. Then calls the wall jump function.

HandleInput()

Reads for Player Movement input and updates the player's velocity accordingly.

LogicUpdate()

Checks the vertical velocity, if it less than zero then change to the airborne state.

PhysicsUpdate()

Updates if the player is grounded, and moves the player. If the player is not grounded then applies the value of gravity to the Y Axis of the vertical velocity.

WallJump()

Calculates the normals of the wall based upon the normal of the raycast variable. Then calculates the players force dependent on the up force and side force values. Then updates the vertical velocity by setting it to the calculated force to apply.

Exit()

Starts the jump and wall jump cooldowns, and resets the relevant values on the Player Controller.

WallRunning.cs

Description

This script controls the wall running state of the player.

Script

```
using Unity.Cinemachine;
using UnityEngine;

namespace Player.FSM.States
{
    public class WallRunning : FsmState
    {
        private bool _isGrounded;
        private bool _isJumping;
        private Vector2 _mouseInput;
        private Vector2 _movementInput;
        private Vector3 _verticalVelocity;
        private float _mouseX;
        private float _mouseY;
        private float _xRotation;
        private Vector3 _targetRotation;
        private CinemachineCamera _thirdPersonCam;
        private CinemachineCamera _firstPersonCam;
        private LayerMask _whatIsWall;
        private float _wallRunForce;
        private float _wallRunMaxDuration;
        private float _wallRunExitTime;
        private float _wallRunSpeed;
        private RaycastHit _leftWallHit;
        private RaycastHit _rightWallHit;
        private bool _leftWall;
        private bool _rightWall;
        private bool _isExitingWall;
```

```
        private bool _exitWallTimerExceeded;
        private bool _exitWallTimerActive;
        private float _maxWallDistance;

        private Transform PlayerTransform => Character.PlayerTransform;


        public WallRunning(PlayerController playerController,
FiniteStateMachine stateMachine) : base(stateMachine,
            playerController)
        {
            Character = playerController;
            StateMachine = stateMachine;
        }

        public override void Enter()
        {
            base.Enter();

            _isGrounded = false;
            _wallRunForce = Character.WallRunForce;
            _maxWallDistance = Character.MaxWallDistance;
            _whatIsWall = Character.WhatIsWall;

            if (Character.leftWall)
            {
                _leftWallHit = Character.LeftWallHit;
                _leftWall = true;
            }
            else if (Character.rightWall)
            {
                _rightWallHit = Character.RightWallHit;
                _rightWall = true;
            }
        }

        public override void HandleInput()
        {
```

```
            base.HandleInput();

            _isJumping = JumpAction.IsPressed();
            _movementInput = MoveAction.ReadValue<Vector2>();
        }

        public override void LogicUpdate()
        {
            base.LogicUpdate();

            if (!_leftWall && !_rightWall && !_isGrounded)
                StateMachine.ChangeState(Character.AirborneState);
            if (_isJumping && Character.canWallJump)
                StateMachine.ChangeState(Character.WallJumpingState);
        }

        public override void PhysicsUpdate()
        {
            base.PhysicsUpdate();
            _isGrounded = Character.isGrounded;
            var right = PlayerTransform.right;
            var position = PlayerTransform.position;
            _rightWall = Physics.Raycast(position, right, out
_rightWallHit, _maxWallDistance, _whatIsWall);
            _leftWall = Physics.Raycast(position, -right, out
_leftWallHit, _maxWallDistance, _whatIsWall);

            if (_rightWall)
                _leftWall = false;
            else if (_leftWall)
                _rightWall = false;


            if (_rightWall && _leftWall)
                Debug.LogError("Both walls have been detected. This is
logically not meant to happen.");

            if (!_leftWall && !_rightWall) return;
```

```
            var wallNormal = _rightWall ? _rightWallHit.normal :
_leftWallHit.normal;
            var wallForward = Vector3.Cross(wallNormal,
PlayerTransform.up);
            if ((PlayerTransform.forward - wallForward).magnitude >
(PlayerTransform.forward - -wallForward).magnitude)
                wallForward = -wallForward;
            Character.characterController.Move(wallForward *
(_wallRunForce * Time.deltaTime));

            switch (_leftWall)
            {
                case true or true when _movementInput is not { x: 0, y:
0 } && !_isGrounded:
                {
                    if (_leftWall)
                    {
                        if (Character.IsTutorial &&
!TutorialController.WallRunChecks["FirstWall"])

TutorialController.WallRunChecks["FirstWall"] = true;
                            Character.leftWall = true;
                            Character.LeftWallHit = _leftWallHit;
                    }
                    else if (_rightWall)
                    {
                        if (Character.IsTutorial &&
TutorialController.WallRunChecks["FirstWall"] &&

!TutorialController.WallRunChecks["SecondWall"])

TutorialController.WallRunChecks["SecondWall"] = true;
                            Character.rightWall = true;
                            Character.RightWallHit = _rightWallHit;
                    }

                    break;
```

```
                }
                case false when _rightWall && _movementInput is not { x:
0, y: 0 }:
                    if (Character.IsTutorial &&
TutorialController.WallRunChecks["FirstWall"] &&
                        !TutorialController.WallRunChecks["SecondWall"])
                        TutorialController.WallRunChecks["SecondWall"] =
true;
                    Character.characterController.Move(-wallNormal *
(100 * Time.deltaTime));
                    break;
            }
        }

        public override void Exit()
        {
            base.Exit();
            if (!_isJumping) return;
            if (_leftWall && !_rightWall)
            {
                Character.jumpingFromLeftWall = true;
                Character.jumpingFromRightWall = false;
                Character.rightWall = false;
                Character.leftWall = false;
                Character.JumpingLeftWallHit = _leftWallHit;
            }
            else if (_rightWall && !_leftWall)
            {
                Character.jumpingFromRightWall = true;
                Character.jumpingFromLeftWall = false;
                Character.leftWall = false;
                Character.rightWall = false;
                Character.JumpingRightWallHit = _rightWallHit;
            }
        }
```

```
    }
}
```

Public Methods

WallRunning()

Stores a reference to the player controller script and the state machine controller.

Enter()

Sets variable values to match the values on the Player Controller script, while also zeroing variables that may have been true before entering this state. Also checks which side of the player that the wall they are running on is.

HandleInput()

Reads for Player Movement input and updates the player's velocity accordingly.

LogicUpdate()

If no walls on either side of the player are detected then change to the airborne state. If the player is jumping and can wall jump then change to the wall jumping state.

PhysicsUpdate()

Updates if the player is grounded, and calculates the correct forces to apply on the play depending on the walls detected and their magnitudes. Then moves the play based upon the force and forward values of the walls. If the player is in the tutorial level then it updates the wall run checks to indicate that the player has passed certain walls.

Exit()

Indicates that the player is no longer wall running, if the player is jumping whilst wall running then it sets the values appropriate to what side of the player the wall they are jumping from is.

FsmState.cs

Description

The base class for all the states used within the Player's Finite State Machine.

Script

```
using UnityEngine;
using UnityEngine.InputSystem;
using Cameras;
using Tutorial;
using Unity.Cinemachine;

namespace Player.FSM
{
    public abstract class FsmState
    {
        protected PlayerController Character;
        protected readonly TutorialController TutorialController;
        protected FiniteStateMachine StateMachine;

        protected readonly InputAction MoveAction;
        protected readonly InputAction JumpAction;
        protected readonly InputAction SlideAction;
        protected readonly float PlayerSpeed;
        protected readonly float GravityValue;
        private Vector2 _mouseInput;
        private float _mouseX;
        private float _mouseY;
        // ReSharper disable once NotAccessedField.Local
        private CinemachineCamera _thirdPersonCam;
        // ReSharper disable once NotAccessedField.Local
        private CinemachineCamera _firstPersonCam;
        private float _xRotation;
        private Vector3 _targetRotation;
```

```
        protected FsmState(FiniteStateMachine stateMachine,
PlayerController playerController)
        {
            StateMachine = stateMachine;
            Character = playerController;


            MoveAction =
playerController.playerInput.actions["Movement"];
            JumpAction = playerController.playerInput.actions["Jump"];
            SlideAction = playerController.playerInput.actions["Slide"];
            PlayerSpeed = Character.PlayerSpeed;
            GravityValue = Character.PlayerGravity;

            if (Character.IsTutorial) TutorialController =
Character.tutorialController;
        }


        public virtual void Enter()
        {
        }

        public virtual void HandleInput()
        {
        }

        public virtual void LogicUpdate()
        {

        }

        public virtual void PhysicsUpdate()
        {
            CameraChanger.GetActiveCams(out _thirdPersonCam, out
_firstPersonCam);
            switch (MainCamera.ActiveCameraMode)
```

```
            {
                case CameraChanger.CameraModes.FirstPerson:
                    var playerLocalRotation =
Character.PlayerTransform.localRotation;
                    playerLocalRotation.y =
Character.activeCinemachineBrain.transform.rotation.y;
                    Character.PlayerTransform.localRotation =
playerLocalRotation;
                    break;
            }
        }


        public virtual void Tick(float deltaTime)
        {

        }

        public virtual void Exit()
        {
        }

    }
}
```

Protected Methods

FsmState()

Sets the values required for each state to be used.

Public Methods

Enter()

Base function for the logic that happens when a state is entered.

HandleInput()

Base function for handling the player's input.

LogicUpdate()

Base function for handling the player logic.

PhysicsUpdate()

Base function for handling the player physics.

FiniteStateMachine.cs

Description

This script defines the Player's Finite State Machine system, and handles the initialisation & changing of the active state.

Script

```
namespace Player.FSM
{
    public abstract class FiniteStateMachine
    {
        private FsmState _initialState;

        public FsmState CurrentState { get; set; }
        public FsmState PreviousState { get; set; }



        public void Initialize(FsmState startingState)
        {
            CurrentState = startingState;
            CurrentState.Enter();

        }


        public void ChangeState(FsmState newState)
        {
            PreviousState = CurrentState;
            CurrentState?.Exit();
            CurrentState = newState;
            CurrentState?.Enter();
        }
```

```
    }
}
```

Public Methods

Initialize()

Sets the current state of the Finite State Machine to the state passed through as an input, and then enters that state.

ChangeState()

Changes the current state of the Finite State Machine. Stores a reference to the previous state, exits it, updates the current state and then enters the new state.

Variables

CurrentState

The current state of the Finite State Machine.

PreviousState

The previous state of the Finite State Machine.

PlayerController.cs

Description

This script is used for controlling the player and handling all the player-relevant logic.

Script

```
using System;
using System.Collections;
using Cameras;
using Unity.Cinemachine;
using input;
using Player.FSM.States;
using Tutorial;
using UI;
using UnityEngine;
using UnityEngine.InputSystem;
using Weapons;

namespace Player
{
    public static class TransformExtensions
    {
        public static GameObject FindGameObjectInChildWithTag(this
Transform parent, string tag)
        {
            GameObject foundChild = null;
            for (var i = 0; i < parent.childCount; i++)
            {
                var child = parent.GetChild(i);
                if (child.CompareTag(tag))
                    foundChild = child.gameObject;
            }

            if (foundChild)
```

```
            {
                return foundChild;
            }

            throw new Exception("No child object with tag!");
        }
    }

    public class PlayerController : MonoBehaviour
    {
        #region Required Components

        [HideInInspector] public GameObject eventSystem;
        [HideInInspector] public inputSystem inputSystem;
        [HideInInspector] public PlayerInput playerInput;
        [HideInInspector] public MainCamera mainCamera;
        [HideInInspector] public CharacterController
characterController;
        [HideInInspector] public GameObject playerMesh;
        [HideInInspector] public PlayerShooting playerShooting;
        [HideInInspector] public PlayerHealth playerHealth;
        [HideInInspector] public CanvasScript canvasScript;
        [HideInInspector] public AudioSource audioSource;

        #endregion

        #region Player States

        private PlayerStateMachine _playerStateMachine;
        [HideInInspector] public Idle IdleState;
        [HideInInspector] public Walking WalkingState;
        [HideInInspector] public Jumping JumpingState;
        [HideInInspector] public Airborne AirborneState;
        [HideInInspector] public WallJumping WallJumpingState;
        [HideInInspector] public WallRunning WallRunState;
        [HideInInspector] public Sliding SlidingState;

        #endregion
```

```
#region Configurable Settings

[Header("Player Movement")]
[SerializeField] private float playerSpeed;
[SerializeField] private float sprintingSpeed;
[SerializeField] private Quaternion maxWallRotation;

[Header("Player Look")] [Range(0, 200)]
[SerializeField] private float mouseSensitivity;
[SerializeField] private float xClamp;
[SerializeField] private float rotationSpeed;



[Header("Player Jump")]
[SerializeField] private float playerJumpHeight;
[SerializeField] private float playerGravity;
[SerializeField] private float playerJumpCooldown;

[Header("Layer Mask Settings")]
[SerializeField] private LayerMask groundMask;
[SerializeField] private LayerMask whatIsWall;
[SerializeField] private LayerMask raycastLayers;

[Header("Wall Run Settings")]
[SerializeField] private float wallRunSpeed;
[SerializeField] private float wallRunForce;
[SerializeField] private float wallRunMaxDuration;
[SerializeField] private float wallRunExitTime;
[SerializeField] private float wallRunCooldown;

[Header("Wall Run Detection Settings")]
[SerializeField] private float maxWallDistance;

[Header("Wall Jump Settings")]
[SerializeField] private float wallJumpUpForce;
[SerializeField] private float wallJumpSideForce;
[SerializeField] private float wallMemoryTime;
```

```
        [SerializeField] private float wallJumpCooldown;

        [Header("Sliding Settings")]
        [SerializeField] private float maxSlideTime;
        [SerializeField] private float slideForce;
        [SerializeField] private float slideYScale;
        [SerializeField] private float slideCooldown;

        [Header("Interact Settings")]
        [SerializeField] private float maxInteractDistance;

        [Header("Weapons")]
        [SerializeField] private Pistol pistol;
        [SerializeField] private Shotgun shotgun;

        [Header("Tutorial Settings")]
        [SerializeField] private bool isTutorial;
        [SerializeField] private GameObject lineRender;

        #endregion

        #region Public References to private vars

        public Transform PlayerTransform =>
characterController.transform;
        public float JumpHeight => playerJumpHeight;
        public float PlayerSpeed => playerSpeed;
        public float SlideCooldown => slideCooldown;
        public float PlayerGravity => playerGravity;
        public float WallRunForce => wallRunForce;
        public LayerMask WhatIsWall => whatIsWall;
        public float MaxSlideTime => maxSlideTime;
        public float SlideForce => slideForce;
        public float SlideYScale => slideYScale;
        public float JumpCooldown => playerJumpCooldown;
        public float WallRunCooldown => wallRunCooldown;
        public float WallJumpCooldown => wallJumpCooldown;
        public float WallJumpUpForce => wallJumpUpForce;
```

```
        public float WallJumpSideForce => wallJumpSideForce;
        public float MaxWallDistance => maxWallDistance;
        public bool IsTutorial => isTutorial;
        public PlayerStateMachine PlayerFsm => _playerStateMachine;

        #endregion

        #region Public Vars

        public bool isGrounded;
        public bool canSlide;
        public bool canJump;
        public bool checkForWallsWhenAirborne;
        public bool canWallJump;
        public bool jumpingFromLeftWall;
        public bool jumpingFromRightWall;
        public bool leftWall;
        public bool rightWall;

        public RaycastHit JumpingLeftWallHit;
        public RaycastHit JumpingRightWallHit;
        public RaycastHit LeftWallHit;
        public RaycastHit RightWallHit;

        public CinemachineBrain activeCinemachineBrain;
        public TutorialController tutorialController;

        #endregion


        public void Awake()
        {
            audioSource = GetComponent<AudioSource>();
            characterController =
GetComponentInChildren<CharacterController>();
            eventSystem =
GameObject.FindGameObjectWithTag("EventSystem");
            mainCamera = eventSystem.GetComponent<MainCamera>();
```

```
            inputSystem = eventSystem.GetComponent<inputSystem>();
            playerMesh =
transform.FindGameObjectInChildWithTag("PlayerMesh");
            playerInput = GetComponent<PlayerInput>();
            playerHealth = GetComponent<PlayerHealth>();
            canvasScript =
GameObject.FindGameObjectWithTag("Canvas").GetComponent<CanvasScript>();
            _playerStateMachine = new PlayerStateMachine();
            activeCinemachineBrain =
GetComponentInChildren<CinemachineBrain>();
            playerShooting = GetComponent<PlayerShooting>();
            if (isTutorial) tutorialController =
eventSystem.GetComponent<TutorialController>();
            IdleState = new Idle("Idle", this, _playerStateMachine);
            WalkingState = new Walking("Walking", this,
_playerStateMachine);
            JumpingState = new Jumping("Jumping", this,
_playerStateMachine);
            WallRunState = new WallRunning(this, _playerStateMachine);
            AirborneState = new Airborne("Airborne", this,
_playerStateMachine);
            SlidingState = new Sliding("Sliding", this,
_playerStateMachine);
            WallJumpingState = new WallJumping(this,
_playerStateMachine);
            playerInput.actions["Shoot"].performed += _ =>
playerShooting.Fire();
            playerInput.actions["Interact"].performed += _ =>
Interact();
            playerInput.actions["Reload"].performed += _ =>
playerShooting.Reload();
            canSlide = true;
            canJump = true;
            canWallJump = true;
            _playerStateMachine.Initialize(IdleState);
            Cursor.lockState = CursorLockMode.Locked;
            SetMouseSensitivity();
        }
```

```
        private void Update()
        {
            _playerStateMachine.CurrentState.HandleInput();
            _playerStateMachine.CurrentState.LogicUpdate();
        }


        private void SetMouseSensitivity()
        {
            mainCamera.SetSensitivity(mouseSensitivity);
        }

        private void FixedUpdate()
        {
            isGrounded = characterController.isGrounded;
            _playerStateMachine.CurrentState.PhysicsUpdate();
        }

        public static IEnumerator ActionCooldown(Action
cooldownComplete, float timeToTake)
        {
            yield return new WaitForSeconds(timeToTake);
            cooldownComplete?.Invoke();
        }

        private void Interact()
        {
            activeCinemachineBrain.gameObject.TryGetComponent<Camera>
(out var activeCam);
            var rayOrigin = new Ray(activeCam.transform.position,
activeCam.transform.forward);
            if (!Physics.Raycast(rayOrigin, out var hit,
maxInteractDistance)) return;
            switch (hit.transform.tag)
            {
                case "Pistol":
                    var collidedPistol = hit.transform.gameObject;
```

```
                    playerShooting.EquipWeapon(pistol);
                    pistol.gameObject.SetActive(true);
                    if (collidedPistol.TryGetComponent<Outline>(out _))
                        Destroy(collidedPistol.GetComponent<Outline>());
                    Destroy(collidedPistol);
                    if (isTutorial)
tutorialController.PistolCollected();
                    break;
                case "Shotgun":
                    var collidedShotgun = hit.transform.gameObject;
                    playerShooting.EquipWeapon(shotgun);
                    shotgun.gameObject.SetActive(true);
                    Destroy(collidedShotgun);
                    break;
                case "Computer":
                    tutorialController.ComputerInteracted();
                    break;
            }
        }
    }
}
```

Public Methods

ActionCooldown()

Waits for a specified amount of time, then invokes the cooldown action passed into the function.

Private Methods

Awake()

Gets all the required components for the script to function. Also creates the states for the Player State Machine, and creates the bindings linked in with the Input System.

Update()

Calls the current state within the Player State Machine to handle Player Input and Logic Updates.

SetMouseSensitivity()

Calls the Main Camera within the scene and sets the mouse sensitivity.

FixedUpdate()

Checks if the player is grounded, and calls the current state within the Player State Machine to handle Physics Updates.

Interact()

Handles the player interacting with objects within the scene. Raycasts, then checks the tag of the hit object, and does the relevant logic.

FindGameObjectInChildWithTag()

Static function that finds a child object of a specified Game Object with a matching tag.

Variables

Components

eventSystem

The event system Game Object that handles logic within the scene.

inputSystem

The component that handles Player Input.

playerInput

The input map for the Player Controls.

mainCamera

The Main Camera script that controls the Games camera components.

characterController

The character controller component attached to the player Game Object.

playerMesh

The Player's Mesh / visual model.

playerShooting

The script that controls the players weapon system.

playerHealth

The script that manages the players health system.

canvasScript

The script with is used to handle UI elements within the scene.

audioSource

The Audio Source component attached to the player.

Player States

_playerStateMachine

The script that controls the current player state.

IdleState

The Idle state script for the player.

WalkingState

The Walking state script for the player.

SprintingState

The Sprinting state script for the player.

JumpingState

The Jumping state script for the player.

AirborneState

The Airborne state script for the player.

WallJumpingState

The Wall Jumping state script for the player.

WallRunState

The Wall Running state script for the player.

SlidingState

The Sliding state script for the player.

Player Settings

Player Movement

playerSpeed

The value of the player's movement speed.

sprintingSpeed

The value of the player's sprinting speed.

Player Look

mouseSensitivity

The mouse sensitivity of the cameras in-game.

xClamp

The limits of the X axis for the camera.

Player Jump

playerJumpHeight

How high the player can jump.

playerGravity

The strength of gravity for the player.

playerJumpCooldown

The time determining the length when the player can jump again.

Layer Masks

whatIsWall

A LayerMask to determine what walls can be wall ran on.

raycastLayers

The Layers in-game that can receive raycasts.

Wall Running

wallRunSpeed

The speed that the player can run walls at.

wallRunForce

The force applied onto the side of the player during wall running.

wallRunMaxDuration

The length of how long the player can wall run for.

wallRunCooldown

The time determining when the player can wall run again.

maxWallDistance

The max distance between the player and the walls that can be detected during ray casting.

Wall Jumping

wallJumpUpForce

The amount of force applied to the player's Y Axis when wall jumping.

wallJumpSideForce

The amount of force applied to the player's X Axis when wall jumping.

wallJumpCooldown

The time determining when the player can wall jump again.

Sliding

maxSlideTime

The max time the player is able to slide for.

slideForce

The force applied onto the player when sliding.

slideYScale

The scale applied to the Y Axis of the player when sliding.

slideCooldown

The time determining when the player can slide again.

Interaction

maxInteractDistance

The max distance for raycasts.

Weapons

pistol

The pistol object childed to the player.

shotgun

The shotgun object childed to the player.

Tutorial

isTutorial

Determines if the player is in the tutorial stage or not.

Encapsulated Public Variables

Player

PlayerTransform

Reference to the Character Controllers transform component.

PlayerGravity

Reference to the player's gravity.

PlayerSpeed

Reference to the player's speed.

Wall Run Settings

WallRunForce

Reference to the player's wall run force.

WallRunSpeed

Reference to the player's wall run speed.

WhatIsWall

Reference to the layer mask determining what can be wall ran on.

Slide Settings

SlideCooldown

Reference to the player's slide cooldown.

MaxSlideTime

Reference to the player's max slide time.

SlideForce

Reference to the player's slide force.

SlideYScale

Reference to the player's Y Scale when sliding.

Jump Settings

JumpHeight

Reference to the player's jump height.

JumpCooldown

Reference to the player's jump cooldown length.

WallRunCooldown

Reference to the player's wall run cooldown length.

WallJumpCooldown

Reference to the player's wall jump cooldown length.

WallJumpUpForce

Reference to the player's wall jump Y Axis force.

WallJumpSideForce

Reference to the player's wall jump X Axis force.

Interaction Settings

MaxWallDistance

Reference to the max wall distance from the player.

Tutorial Settings

IsTutorial

Reference to the check determining if the player is in a tutorial.

Player State Machine

PlayerFsm

Reference to the player's state machine.

Public Variables

isGrounded

Reference to if the player is grounded.

canSlide

Reference to if the player can slide.

canJump

Reference to if the player can jump.

checkForWallsWhenAirborne

Reference to if the script should detect walls when the player is airborne.

canWallRun

Reference to if the player can wall run.

canWallJump

Reference to if the player can wall jump.

jumpingFromLeftWall

Reference to if the player is jumping from a wall on the left.

jumpingFromRightWall

Reference to if the player is jumping from a wall on the right.

leftWall

Reference to if a wall is detected on the left of the player.

rightWall

Reference to if a wall is detected on the right of the player.

JumpingLeftWallHit

The RaycastHit value of the wall detected to the left of the player while wall jumping.

JumpingRightWallHit

The RaycastHit value of the wall detected to the right of the player while wall jumping.

LeftWallHit

The RaycastHit value of the wall detected to the left of the player.

RightWallHit

The RaycastHit value of the wall detected to the right of the player.

activeCinemachineBrain

The active Cinemachine Brain component in the scene.

tutorialController

The tutorial controller within the tutorial scene.

PlayerHealth.cs

Description

This script is used to manage the Player's health in-game. It calls the required functions for the Player Death to happen.

Script

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace Player
{
    public class PlayerHealth : MonoBehaviour
    {
        [SerializeField] private float maxHealth = 10;
        public float CurrentHealth { get; private set; }

        public float MaxHealth => maxHealth;


        public void Damage(float amount)
        {
            CurrentHealth -= amount;
            if (CurrentHealth <= 0) Die();
        }

        private void Start()
        {
            CurrentHealth = maxHealth;
        }

        private static void Die()
        {
            LoadNextScene();
```

```
        }

        private static void LoadNextScene()
        {
            Cursor.lockState = CursorLockMode.None;
            SceneManager.LoadScene(2);
        }
    }
}
```

Public Methods

Damage()

Lowers the Player's health by the value of the float as an input.

Private Methods

Start()

Called on game start, sets the Players Current Health to the value of the Max Health.

Die()

Calls the function to load the Death Scene

LoadNextScene()

Enables the cursor and loads the death scene.

LoadLevel() - Not Implemented

Originally was going to be used to fade the scene to black before loading the Death Scene.

Variables

maxHealth

The value that the player's max health will be upon game start.

CurrentHealth

Encapsulated value that stores the player's current health.

MaxHealth

Encapsulated and public variable for the maxHealth variable.

PlayerAnimation.cs

Description

This script controls the animations for the player character model.

Script

```
using UnityEngine;

namespace Player
{
    public class PlayerAnimation : MonoBehaviour
    {
        private PlayerController _playerController;
        private Animator _playerAnimator;
        private float _velocity;
        private Vector3 _previousPos;
        private GameObject _playerMesh;
        private static readonly int Velocity =
Animator.StringToHash("Velocity");
        private static readonly int HasWeapon =
Animator.StringToHash("hasWeapon");

        private void Start()
        {
            _playerController = GetComponent<PlayerController>();
            _playerMesh = _playerController.playerMesh;
            _playerAnimator = _playerMesh.GetComponent<Animator>();
        }


        private void LateUpdate()
        {
            _velocity = (_playerMesh.transform.position -
_previousPos).magnitude / Time.deltaTime;
```

```
            _previousPos = _playerMesh.transform.position;
            _playerAnimator.SetBool(HasWeapon,
_playerController.playerShooting.CurrentWeapon);
            _playerAnimator.SetFloat(Velocity, _velocity);
        }
    }
}
```

Private Methods

Start()

Gets the relevant components required for the script to function

LateUpdate()

Calculates the players velocity and updates the Animator components variables accordingly

Variables

_playerController

A variable reference to the Player Controller script.

_playerAnimator

A variable reference to the Player Animator component.

_velocity

A variable reference to the speed of the Player.

_previousPos

A variable storing the previous position of the Player at the previous frame.

_playerMesh

A variable reference to the Player Mesh GameObject.

Velocity

a hashed reference to the Velocity variable within the Player Animator component.

HasWeapon

a hashed reference to the hasWeapon variable within the Player Animator component.

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 List<GameObject> pooledProjectiles;

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

        private GameObject _projParent;


        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.

Variables

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.

PlayerShooting.cs

Description

Script

```
using TMPro;
using UnityEngine;
using UnityEngine.UI;
using Weapons;

namespace Player
{
    public class PlayerShooting : MonoBehaviour
    {
        [Header("Fists Class for game start or when no weapon")]
        [SerializeField] private BaseWeapon fists;
        [SerializeField] private TextMeshProUGUI ammoReporter;
        [SerializeField] private GameObject ammoPanel;
        [SerializeField] private Image reloadBar;
        private BaseWeapon _previousWeapon;
        public BaseWeapon CurrentWeapon { get; private set; }
        public bool HasWeapon()
        {
            return CurrentWeapon;
        }


        public void EquipWeapon(BaseWeapon newWeapon)
        {
            CurrentWeapon = newWeapon;
            CurrentWeapon.CurrentPrimaryAmmo =
CurrentWeapon.maxPrimaryAmmo;
            CurrentWeapon.CurrentSecondaryAmmo =
CurrentWeapon.maxSecondaryAmmo;
        }
```

```
        public void Reload()
        {
            if (!CurrentWeapon) return;
            CurrentWeapon.Reload();
        }

        public void Fire()
        {
            if (!CurrentWeapon || CurrentWeapon.CurrentPrimaryAmmo <= 0)
return;
            CurrentWeapon.Fire();
        }

    }
}
```

Input Scripts

Start typing here...

InputSystem.cs

Description

Script

```
using UnityEngine;

namespace input
{
    public class inputSystem : MonoBehaviour
    {
        public inputSystem Instance { get; private set; }

        public float HorizontalInput => _movementInput.x;
        public float VerticalInput => _movementInput.y;

        public float MouseX => _mouseInput.x;
        public float MouseY => _mouseInput.y;


        private Vector2 _movementInput;
        private Vector2 _mouseInput;

        private void Awake()
        {
            if (Instance != null && Instance != this)
                Destroy(gameObject);
            else
            {
                Instance = this;
            }
        }


        private void OnEnable()
```

```
        {
            Cursor.lockState = CursorLockMode.Locked;
        }

    }
}
```

Tutorial Scripts

Start typing here...

HighlightWeapon.cs

Description

Script

```
using System.Collections;
using UnityEngine;

namespace Tutorial
{
    public class HighlightWeapon : MonoBehaviour
    {
        [SerializeField] private float targetOutlineWidth;
        [SerializeField] private Color outlineColor;
        [SerializeField] private float timeToTake;
        [SerializeField] private float pulseTime;


        private Outline _weaponOutline;


        private void Start()
        {
            _weaponOutline = GetComponent<Outline>();
            _weaponOutline.OutlineWidth = 0;
            _weaponOutline.enabled = false;
            _weaponOutline.OutlineColor = outlineColor;
        }

        public void OutlineWeapon()
        {
            _weaponOutline.enabled = true;
            LeanTween.value(_weaponOutline.gameObject,
_weaponOutline.OutlineWidth, targetOutlineWidth, 2f)
                .setOnUpdate(
```

```
                    f =>
                    {
                        _weaponOutline.OutlineWidth = f;
                    })
                .setLoopPingPong();
        }

        private IEnumerator LerpHighlight()
        {
            var elapsedTime = 0f;
            while (_weaponOutline.OutlineWidth < targetOutlineWidth)
            {
                _weaponOutline.OutlineWidth = Mathf.Lerp(2f,
targetOutlineWidth, elapsedTime / timeToTake);
                elapsedTime += Time.deltaTime;
                yield return null;
            }

            StartCoroutine(PulseOutline());
            yield return null;
        }

        private IEnumerator PulseOutline()
        {
            var timeTaken = 0f;
            while (_weaponOutline.OutlineWidth > 0f)
            {
                _weaponOutline.OutlineWidth =
Mathf.Lerp(_weaponOutline.OutlineWidth, 0f, timeTaken / pulseTime);
                timeTaken += Time.deltaTime;
                yield return null;
            }
            StartCoroutine(LerpHighlight());
            yield return null;

        }
```

```
    }
}
```

HighlightComputer.cs

Description

Script

```
using UnityEngine;

namespace Tutorial
{
    public class HighlightComputer : MonoBehaviour
    {
        [SerializeField] private float targetOutlineWidth;
        [SerializeField] private Color outLineColor;
        [SerializeField] private float timeToTake;

        private Outline _computerOutline;

        private void Start()
        {
            _computerOutline = GetComponent<Outline>();
            _computerOutline.OutlineWidth = 0;
            _computerOutline.enabled = false;
            _computerOutline.OutlineColor = outLineColor;
        }



        public void OutlineComputer()
        {
            _computerOutline.enabled = true;
            LeanTween.value(_computerOutline.gameObject,
_computerOutline.OutlineWidth, targetOutlineWidth, timeToTake)
                .setOnUpdate(
                    f =>
                    {
```

```
                        _computerOutline.OutlineWidth = f;
                    })
                .setLoopPingPong();
        }

        public void StopOutline()
        {
            LeanTween.cancel(_computerOutline.gameObject);
            _computerOutline.enabled = false;
        }

    }
}
```

LargeIslandSensor.cs

Description

Script

```
using UnityEngine;

namespace Tutorial
{
    public class LargeIslandSensor : MonoBehaviour
    {
        [SerializeField] private TutorialController tutorialController;

        private void OnTriggerEnter(Collider other)
        {
            if (!other.transform.parent.gameObject.CompareTag("Player"))
return;
            tutorialController.OtherIslandReached();
            Destroy(gameObject);

        }

        private void OnTriggerStay(Collider other)
        {
            if (!other.transform.parent.gameObject.CompareTag("Player"))
return;
            tutorialController.OtherIslandReached();
            Destroy(gameObject);
        }
    }
}
```

TutorialController.cs

Description

Script

```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using Environment;
using Player;
using TMPro;
using UnityEngine;
using UnityEngine.Serialization;
using UnityEngine.UI;

namespace Tutorial
{
    /** sources used:
     * https://stackoverflow.com/questions/70073128/how-to-check-if-all-values-of-a-c-sharp-dictionary-are-true
     * https://imran-momin.medium.com/dictionaries-unity-c-69b48448445f
     * https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.keyvaluepair-2?view=netframework-4.8
     */
    public static class ImageTweening
    {
        public static void ClearAlpha(ref Image img, bool loop)
        {
            var tempImg = img;
            if (loop)
            {
                LeanTween.value(img.gameObject, img.color.a, 0f, 3f).setOnUpdate(val =>
                {
```

```
                    Color c = tempImg.color;
                    c.a = val;
                    tempImg.color = c;
                }).setLoopPingPong();
            }
            else
            {
                LeanTween.value(img.gameObject, img.color.a, 0f,
3f).setOnUpdate(val =>
                {
                    Color c = tempImg.color;
                    c.a = val;
                    tempImg.color = c;
                });
            }

            img = tempImg;
        }

        public static void ClearTextAlpha(ref TextMeshProUGUI text, bool
loop)
        {
            var tempTxt = text;

            if (loop)
            {
                LeanTween.value(text.gameObject, 1, 0f,
1.5f).setOnUpdate(val =>
                {
                    Color c = tempTxt.color;
                    c.a = val;
                    tempTxt.color = c;
                }).setLoopPingPong();
            }
            else
            {
                LeanTween.value(text.gameObject, 1, 0f,
1.5f).setOnUpdate(val =>
```

```
                {
                    Color c = tempTxt.color;
                    c.a = val;
                    tempTxt.color = c;
                });
            }

            text = tempTxt;
        }


        public static void FillAlpha(ref Image img, bool loop)
        {
            var tempImg = img;
            if (loop)
            {
                LeanTween.value(img.gameObject, img.color.a, 1f,
3f).setOnUpdate(val =>
                {
                    Color c = tempImg.color;
                    c.a = val;
                    tempImg.color = c;
                }).setLoopPingPong();
            }
            else
            {
                LeanTween.value(img.gameObject, img.color.a, 1f,
3f).setOnUpdate(val =>
                {
                    Color c = tempImg.color;
                    c.a = val;
                    tempImg.color = c;
                });
            }

            img = tempImg;
        }
```

```
        public static void AlphaPrompt(ref TextMeshProUGUI text, ref
Image imgMain, ref Image imgAlt, bool loop)
        {
            ClearTextAlpha(ref text, loop);
            ClearAlpha(ref imgMain, loop);
            FillAlpha(ref imgAlt, loop);
        }


        public static void ChangePrompt(ref TextMeshProUGUI text, ref
GameObject oldPrompt, ref GameObject newPrompt,
            ref Image newImg, ref Image newImgAlt, ref Image oldImg, ref
Image oldImgAlt,
            ref Dictionary<int, string> textList, int textIndex)
        {
            var tempTxt = text;
            ClearAlpha(ref oldImg, false);
            ClearAlpha(ref oldImgAlt, false);
            ClearTextAlpha(ref text, false);
            LeanTween.cancel(text.gameObject);
            LeanTween.cancel(oldImg.gameObject);
            LeanTween.cancel(oldImgAlt.gameObject);
            LeanTween.value(text.gameObject, 1, 1f, 1.5f).setOnUpdate(f
=>
            {
                Color c = tempTxt.color;
                c.a = f;
                tempTxt.color = c;
            });

            text.text = textList[textIndex];
            oldPrompt.SetActive(false);
            newPrompt.SetActive(true);

            ClearTextAlpha(ref text, true);
            ClearAlpha(ref newImg, true);
            FillAlpha(ref newImgAlt, true);
        }
```

```
        public static void ChangeTextPromptOnly(ref TextMeshProUGUI
text, ref Dictionary<int, string> textList,
            int textIndex)
        {
            var tempTxt = text;
            ClearTextAlpha(ref text, false);
            LeanTween.value(text.gameObject, 1f, 1f, 1.5f).setOnUpdate(f
=>
            {
                Color c = tempTxt.color;
                c.a = f;
                tempTxt.color = c;
            });
            text.text = textList[textIndex];
            ClearTextAlpha(ref text, true);
        }
    }


    public class TutorialController : MonoBehaviour
    {
        [SerializeField] private Canvas canvas;
        [SerializeField] private PlayerController playerController;
        [SerializeField] private FloatingWallController
floatingWallController;
        [SerializeField] private GameObject tutorialPistol;
        [SerializeField] private TutorialEnemyController
tutorialEnemyController;
        [SerializeField] private GameObject endComputer;

        [FormerlySerializedAs("Prompt_W")] [Header("Input Prompts")]
        [SerializeField] private GameObject promptW;
        [SerializeField] private GameObject promptS;
        [SerializeField] private GameObject promptA;
        [SerializeField] private GameObject promptD;
        [SerializeField] private GameObject promptJump;
        [SerializeField] private GameObject promptComplete;
```

```
        [SerializeField] private Image keyPressW;
        [SerializeField] private Image keyPressWalt;
        [SerializeField] private Image keyPressS;
        [SerializeField] private Image keyPressSalt;
        [SerializeField] private Image keyPressA;
        [SerializeField] private Image keyPressAalt;
        [SerializeField] private Image keyPressD;
        [SerializeField] private Image keyPressDalt;
        [SerializeField] private Image keyPressSpace;
        [SerializeField] private Image keyPressSpacealt;
        [SerializeField] private Image keyPressComplete;
        [SerializeField] private Image keyPressCompletealt;

        [Header("Tutorial Text Prompts")]
        [SerializeField] private TextMeshProUGUI tutorialTextHint;

        [Header("Islands")]
        [SerializeField] private GameObject enemyIsland;

        [Header("Portal")]
        [SerializeField] private GameObject portal;
        [SerializeField] private SpriteRenderer portalSpriteRenderer;


        private HighlightWeapon _pistolOutline;
        private HighlightComputer _computerOutline;
        private bool _areWallsAppearing;
        private bool _isWeaponGlowing;
        private bool _hasEnemyIslandAppeared;
        public bool hasFiredPistolYet;


        public enum NextKeyPress
        {
            Forward,
            Backwards,
            Left,
            Right,
```

```
        Jump,
        Complete
    }

    public NextKeyPress nextKeyToPress;
    public Dictionary<string, bool> TutorialChecks;
    public Dictionary<string, bool> EnemyChecks;
    public Dictionary<string, bool> WallRunChecks;
    private Dictionary<int, string> _introductionTexts;
    private Dictionary<int, string> _inputPromptTexts;
    private Dictionary<int, string> _wallRunPromptTexts;
    private Dictionary<int, string> _weaponPromptTexts;
    private Dictionary<int, string> _enemyIslandTexts;
    private Dictionary<int, string> _challengeCompleteTexts;

    private void Start()
    {
        promptW.SetActive(false);
        promptS.SetActive(false);
        promptA.SetActive(false);
        promptD.SetActive(false);
        promptJump.SetActive(false);
        promptComplete.SetActive(false);
        enemyIsland.SetActive(false);
        portal.SetActive(false);


        TutorialChecks = new Dictionary<string, bool>
        {
            { "IntroductionComplete", false },
            { "Forward", false },
            { "Backwards", false },
            { "Left", false },
            { "Right", false },
            { "Jump", false }
        };

        WallRunChecks = new Dictionary<string, bool>
```

```
            {
                { "FirstWall", false },
                { "SecondWall", false },
                { "IslandReached", false }
            };

            EnemyChecks = new Dictionary<string, bool>
            {
                { "Equipped", false },
                { "Fired", false },
                { "Missed", false },
                { "Killed", false }
            };

            _introductionTexts = new Dictionary<int, string>
            {
                { 0, "Welcome to this tutorial!" },
                { 1, "I'll be your teacher today." },
                { 2, "First, lets familiarize ourselves with this Games
Controls." }
            };


            _inputPromptTexts = new Dictionary<int, string>
            {
                { 0, "Press W to move Forward" },
                { 1, "Press S to move Backwards" },
                { 2, "Press A to move Left" },
                { 3, "Press D to move Right" },
                { 4, "Press Space to Jump" },
                { 5, "Movement Tutorial Complete!" },
            };

            _wallRunPromptTexts = new Dictionary<int, string>
            {
                { 0, "Huh, moving, floating walls. Didn't expect that."
},
                { 1, "Try wall running to the next island." },
```

```
            { 2, "Jump between the walls by pressing Space." },
            { 3, "You did it! Nice work." },
            { 4, "Time to explore this island." }
        };

        _weaponPromptTexts = new Dictionary<int, string>
        {
            { 0, "Oh, a free gun!" },
            { 1, "Press F to pickup the gun." },
            { 2, "Look nearby the campfire, there's a person." },
            { 3, "Shoot the person by pressing Mouse1." },
            { 4, "SHOOT. THEM." },
            { 5, "It'd help if you actually aimed at the person." },
            { 6, "Good Job." },
            { 7, "So uhhh, what now..." },
            { 8, "Come here often?" }
        };

        _enemyIslandTexts = new Dictionary<int, string>
        {
            { 0, "Ah. Shit." },
            { 1, "These guys don't seem too happy." },
            { 2, "Time to kill them I guess." }
        };

        _challengeCompleteTexts = new Dictionary<int, string>
        {
            { 0, "That's those guys taken care of. " },
            { 1, "Huh, what's that device over there?" },
            { 2, "I should press this button." },
            { 3, "Hmm. It's doing nothing." },
            { 4, "Nevermind, spoke too soon." },
            { 5, "A giant portal! Lets go through it. Nothing bad
ever happens with portals." }
        };


        tutorialTextHint.text = _introductionTexts[0];
```

```
            _pistolOutline =
tutorialPistol.GetComponent<HighlightWeapon>();
            _computerOutline =
endComputer.GetComponent<HighlightComputer>();
            tutorialEnemyController =
GetComponent<TutorialEnemyController>();
            ImageTweening.ClearTextAlpha(ref tutorialTextHint, true);
            ImageTweening.ClearAlpha(ref keyPressW, true);
            ImageTweening.FillAlpha(ref keyPressWalt, true);
            nextKeyToPress = NextKeyPress.Forward;
            StartCoroutine(IntroductionText());
        }


        public bool IntroComplete()
        {
            return TutorialChecks["IntroductionComplete"];
        }

        public void OtherIslandReached()
        {
            WallRunChecks["IslandReached"] = true;
            if (!_isWeaponGlowing)
                StartCoroutine(StartWeaponTutorial());
        }

        public void PistolCollected()
        {
            EnemyChecks["Equipped"] = true;
            StartCoroutine(PistolRelatedDialogue());
        }

        public void ActuallyAim()
        {
            if (tutorialTextHint.text != _weaponPromptTexts[4]) return;
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_weaponPromptTexts, 5);
        }
```

```
        public void TutorialEnemyKilled()
        {
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_weaponPromptTexts, 6);
            StartCoroutine(SpawnEnemyIsland());
        }

        public void EnemyChallengeComplete()
        {
            StartCoroutine(EnemiesAreKilled());
        }

        private IEnumerator EnemiesAreKilled()
        {
            tutorialTextHint.gameObject.SetActive(true);
            ImageTweening.ClearTextAlpha(ref tutorialTextHint, true);
            yield return new WaitForSeconds(0.8f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_challengeCompleteTexts, 0);
            yield return new WaitForSeconds(2.5f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_challengeCompleteTexts, 1);
            _computerOutline.OutlineComputer();
        }

        private IEnumerator SpawnPortal()
        {
            yield return new WaitForSeconds(1.2f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_challengeCompleteTexts, 4);
            portal.SetActive(true);
            LeanTween.value(portal, 0f, 1f, 3.5f).setOnUpdate(f =>
            {
                Color c = portalSpriteRenderer.color;
                c.a = f;
                portalSpriteRenderer.color = c;
            }).setOnComplete(() =>
```

```
            {
                endComputer.LeanMoveLocalY(3.25f, 0.4f).setOnComplete(()
=>
                {
                    endComputer.LeanRotateX(-25f, 0.5f).setOnComplete(()
=>
                    {
                        endComputer.LeanMoveLocalY(3.825f,
0.4f).setOnComplete(() =>
                        {
                            endComputer.LeanRotateX(-45f,
0.25f).setOnComplete(() =>
                            {
                                endComputer.LeanMoveLocalX(-8.5f,
0.25f).setOnComplete(() =>
                                {
                                    endComputer.SetActive(false);

ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint,
                                        ref _challengeCompleteTexts, 5);
                                });
                            });
                        });
                    });
                });
            }

        private IEnumerator SpawnEnemyIsland()
        {
            yield return new WaitForSeconds(3f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_weaponPromptTexts, 7);
            yield return new WaitForSeconds(2f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_weaponPromptTexts, 8);
            enemyIsland.SetActive(true);
            yield return new WaitForSeconds(1f);
```

```
            LeanTween.moveY(enemyIsland, -2.5f, 5f).setOnComplete(() =>
            {
                LeanTween.moveX(enemyIsland, -2f, 3f);
                _hasEnemyIslandAppeared = true;
            });
            yield return new WaitUntil(() => _hasEnemyIslandAppeared);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_enemyIslandTexts, 0);
            yield return new WaitForSeconds(2f);
            LeanTween.moveX(enemyIsland, -2f, 3f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_enemyIslandTexts, 1);
            yield return new WaitForSeconds(3f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_enemyIslandTexts, 2);
            yield return new WaitForSeconds(1.2f);
            tutorialEnemyController.StartKillChallenge();
            LeanTween.cancel(tutorialTextHint.gameObject);
            LeanTween.value(tutorialTextHint.gameObject, 1f, 0f,
1.5f).setOnUpdate(f =>
            {
                Color c = tutorialTextHint.color;
                c.a = f;
                tutorialTextHint.color = c;
            }).setOnComplete(() =>
tutorialTextHint.gameObject.SetActive(false));
        }


        public void ComputerInteracted()
        {
            _computerOutline.StopOutline();
            StartCoroutine(FinalPrompts());
        }

        private IEnumerator FinalPrompts()
        {
            yield return new WaitForSeconds(1f);
```

```
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_challengeCompleteTexts, 3);
            yield return new WaitForSeconds(2f);
            StartCoroutine(SpawnPortal());
        }

        private IEnumerator CompleteFirstPrompt()
        {
            yield return new WaitForSeconds(2f);
            LeanTween.value(tutorialTextHint.gameObject, 1f, 0f,
1.5f).setOnUpdate(f =>
            {
                Color c = tutorialTextHint.color;
                c.a = f;
                tutorialTextHint.color = c;
            }).setOnComplete(() =>
                {
                    LeanTween.cancel(keyPressComplete.gameObject);
                    LeanTween.cancel(keyPressCompletealt.gameObject);
                    promptComplete.SetActive(false);
                    StartCoroutine(StartWallRunPrompt());
                }
            );
        }

        private IEnumerator StartWallRunPrompt()
        {
            yield return new WaitForSeconds(2f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_wallRunPromptTexts, 0);
            yield return new WaitForSeconds(2.8f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_wallRunPromptTexts, 1);
            yield return new WaitUntil(() =>
WallRunChecks["FirstWall"]);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_wallRunPromptTexts, 2);
            yield return new WaitUntil(() =>
```

```
WallRunChecks["SecondWall"]);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_wallRunPromptTexts, 3);
            yield return new WaitUntil(() =>
WallRunChecks["IslandReached"]);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_wallRunPromptTexts, 4);
        }

        private void FixedUpdate()
        {
            var allIsTrue = TutorialChecks.Values.All(value => value);
            if (allIsTrue && !_areWallsAppearing)
                StartCoroutine(MakeWallsAppear());
        }

        private IEnumerator IntroductionText()
        {
            yield return new WaitForSeconds(2.5f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_introductionTexts, 1);
            yield return new WaitForSeconds(2f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_introductionTexts, 2);
            yield return new WaitForSeconds(1.6f);
            StartCoroutine(InputPrompts());
        }

        private IEnumerator InputPrompts()
        {
            ImageTweening.ClearTextAlpha(ref tutorialTextHint, false);
            yield return new WaitForSeconds(2f);
            tutorialTextHint.text = _inputPromptTexts[0];
            promptW.SetActive(true);
            ImageTweening.AlphaPrompt(ref tutorialTextHint, ref
keyPressW, ref keyPressWalt, true);
            TutorialChecks["IntroductionComplete"] = true;
            yield return new WaitUntil(() => TutorialChecks["Forward"]);
```

```
            ImageTweening.ChangePrompt(ref tutorialTextHint, ref
promptW, ref promptS, ref keyPressS,
                ref keyPressSalt, ref keyPressW, ref keyPressWalt, ref
_inputPromptTexts, 1);
            nextKeyToPress = NextKeyPress.Backwards;
            yield return new WaitUntil(() =>
TutorialChecks["Backwards"]);
            ImageTweening.ChangePrompt(ref tutorialTextHint, ref
promptS, ref promptA, ref keyPressA,
                ref keyPressAalt, ref keyPressS, ref keyPressSalt, ref
_inputPromptTexts, 2);
            nextKeyToPress = NextKeyPress.Left;
            yield return new WaitUntil(() => TutorialChecks["Left"]);
            ImageTweening.ChangePrompt(ref tutorialTextHint, ref
promptA, ref promptD, ref keyPressD,
                ref keyPressDalt, ref keyPressA, ref keyPressAalt, ref
_inputPromptTexts, 3);
            nextKeyToPress = NextKeyPress.Right;
            yield return new WaitUntil(() => TutorialChecks["Right"]);
            ImageTweening.ChangePrompt(ref tutorialTextHint, ref
promptD, ref promptJump, ref keyPressSpace,
                ref keyPressSpacealt, ref keyPressD, ref keyPressDalt,
ref _inputPromptTexts, 4);
            nextKeyToPress = NextKeyPress.Jump;
            yield return new WaitUntil(() => TutorialChecks["Jump"]);
            nextKeyToPress = NextKeyPress.Complete;
            ImageTweening.AlphaPrompt(ref tutorialTextHint, ref
keyPressSpace, ref keyPressSpacealt, false);
            tutorialTextHint.text = _inputPromptTexts[5];
            promptJump.SetActive(false);
            promptComplete.SetActive(true);
            ImageTweening.ClearTextAlpha(ref tutorialTextHint, true);
            ImageTweening.ClearAlpha(ref keyPressComplete, true);
            StartCoroutine(CompleteFirstPrompt());
        }

        private IEnumerator MakeWallsAppear()
        {
```

```
            _areWallsAppearing = true;
            yield return new WaitForSeconds(2f);
            floatingWallController.TriggerWallMovement();
        }

        private IEnumerator StartWeaponTutorial()
        {
            yield return new WaitForSeconds(1.2f);
            _isWeaponGlowing = true;
            StartCoroutine(ShowWeaponText());
            yield return new WaitForSeconds(1.2f);
            if (_pistolOutline)
                _pistolOutline.OutlineWeapon();
        }

        private IEnumerator ShowWeaponText()
        {
            yield return new WaitForSeconds(1.6f);
            tutorialTextHint.gameObject.SetActive(true);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_weaponPromptTexts, 0);
            if (EnemyChecks["Fired"]) yield break;
            yield return new WaitForSeconds(2f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_weaponPromptTexts, 1);
        }

        private IEnumerator PistolRelatedDialogue()
        {
            yield return new WaitForSeconds(1.2f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_weaponPromptTexts, 2);
            yield return new WaitForSeconds(1.8f);
            ImageTweening.ChangeTextPromptOnly(ref tutorialTextHint, ref
_weaponPromptTexts, 3);
            var timerStart = Time.time;
            yield return new WaitUntil(() => Time.time - timerStart > 5f
|| EnemyChecks["Fired"]);
```

```
            switch (EnemyChecks["Fired"])
            {
                case false when !EnemyChecks["Killed"]:
                    ImageTweening.ChangeTextPromptOnly(ref
tutorialTextHint, ref _weaponPromptTexts, 4);
                    break;
                case true when !EnemyChecks["Killed"]:
                    ImageTweening.ChangeTextPromptOnly(ref
tutorialTextHint, ref _weaponPromptTexts, 5);
                    break;
            }
        }
    }
}
```

TutorialEnemy.cs

Description

Script

```
using UnityEngine;

namespace Tutorial
{
    public class TutorialEnemy : MonoBehaviour
    {
        [SerializeField] private TutorialController tutorialController;
        [SerializeField] private TutorialEnemyController
tutorialEnemyController;
        [SerializeField] private bool isHostile;


        private Animator _enemyAnimator;
        private Collider _enemyCollider;
        private static readonly int IsDead =
Animator.StringToHash("isDead");

        private void Start()
        {
            _enemyAnimator = GetComponent<Animator>();
            _enemyCollider = GetComponentInChildren<Collider>();
        }

        public void Die()
        {
            if (isHostile) return;
            _enemyAnimator.SetBool(IsDead, true);
            tutorialController.EnemyChecks["Killed"] = true;
            tutorialController.TutorialEnemyKilled();
            Destroy(_enemyCollider);
```

```
        }
    }
}
```

TutorialEnemyController.cs

Description

Script

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

namespace Tutorial
{
    public class TutorialEnemyController : MonoBehaviour
    {
        [SerializeField] private List<EnemyController> tutorialEnemies;
        [SerializeField] private CanvasScript canvasScript;

        public int EnemiesRemaining { get; set; }

        private void Start()
        {
            EnemiesRemaining = tutorialEnemies.Count;
        }

        public void EnemyKilled(EnemyController enemyKilled)
        {
            EnemiesRemaining--;
            tutorialEnemies.Remove(enemyKilled);
        }


        public void StartKillChallenge()
        {
            canvasScript.ShowKillChallengeUI(EnemiesRemaining);
            foreach (var enemy in tutorialEnemies)
```

```
                enemy.EnableEnemy();
            }
        }
    }
```