มหาวิทยาลัยอัสสัมชัญ
ASSUMPTION UNIVERSITY
of THAILAND

VINCENT MARY
SCHOOL of SCIENCE and
TECHNOLOGY

Senior Project Report

Dream Catchers 2
A Lost Boy: The Game

Min Khant Kyaw,
Pawat Asavapotiphan,
Thanakrit Chancharung

A. Chayapol Moemeng (Advisor)

CS 3200 Senior Project 1 (1/2020)

Senior Project Approval

Project title:	A Lost Boy: The Game
Academic Year:	1/2020
Authors:	Sunny Emmerich (6121200)
	Kazuhira Miller (6121300)
	Shalashaska Ocelot (6121400)
Project Advisor:	Dr Hal Emmerich

The Senior Project committee's cooperation between the Department of Computer Science and Information Technology, Vincent Mary School of Science and Technology, Assumption University had approved this Senior Project. The Senior Project in partial fulfilment of the requirement for the degree of Bachelor of Science in Computer Science and Information technology.

Approval Committee:

……..

(Dr Hal Emmerich)

Project Advisor

……..	……..
(Dr Drago Pettrovich Madnar)	(Frank Jaeger)
Committee Member	Committee Member

Abstract

A Lost Boy is a 2D platformer game developed as part of a senior project, aimed at delivering an immersive experience combining platforming, combat, and puzzle-solving elements. Drawing inspiration from *Hollow Knight*, the game features hand-crafted levels, dynamic camera controls, combat systems, and an engaging storyline. Utilizing Unity, this project showcases technical and creative proficiency, balancing original assets with those sourced from the Unity Asset Store. Despite limitations such as resource constraints and time pressures, the game demonstrates a solid structure and has the potential for further development. The project emphasizes the collaborative process of game development, incorporating design, animation, and sound, ultimately culminating in a well-rounded final product.

Table Of Contents

[Use a word processor to generate the table of contents]

Chapter 1: Introduction

1.1 Problem Statement

The problem is that game development is often not widely practiced or taught effectively in many educational institutions, leading to a lack of proper guidance for students interested in starting their journey in this field. As a result, most student projects, especially during university term projects, tend to be simple remakes of popular games like *Super Mario*, with little originality or innovation, relying heavily on copied online resources. To address this, there is a need to elevate the game development departments within faculties, providing students with the tools, knowledge, and creative freedom to produce viable, playable products featuring authentic characters and unique stories and techniques used from Major game industries. This approach will not only enhance the skills of aspiring game developers but also help showcase the faculty's strengths to attract future students interested in game development and create their own indie teams and compete in the market.

1.2 Scope of the project

What will covered in this project.

Table 1: scope of the project

	Functional Scope	Brief Explanation
1	6 Major Functions	Necessary core systems required for completeness of the game
2	6 Minor Functions	Minor systems that help with the visual, game play and mechanics
3	Game Level Design	4 stages of Levels
4	Asset Creation	The resource needed for players, enemies, boss, items and so on
5	Animation methods	Creative and technical process for managing state action
6	Basic Sound Design	Sound effects, Theme, BG sounds

What will not be covered in this project.

For this project, there are several areas we will not cover in depth. We will limit ourselves to implementing six major and six minor functions to maintain focus and ensure quality. Sound design will not be a primary focus, and while we will include basic audio, we won't go into extensive detail. Asset creation will not be fully original; some assets will be sourced from free libraries, and we will modify only a few. The game will have no more than four levels. We will not incorporate a database system, as all game data will be stored locally. Additionally, the project will focus on a single platform—Windows—with support for keyboard and controller input, without expanding to other platforms.

1.3 Proposed Methodology

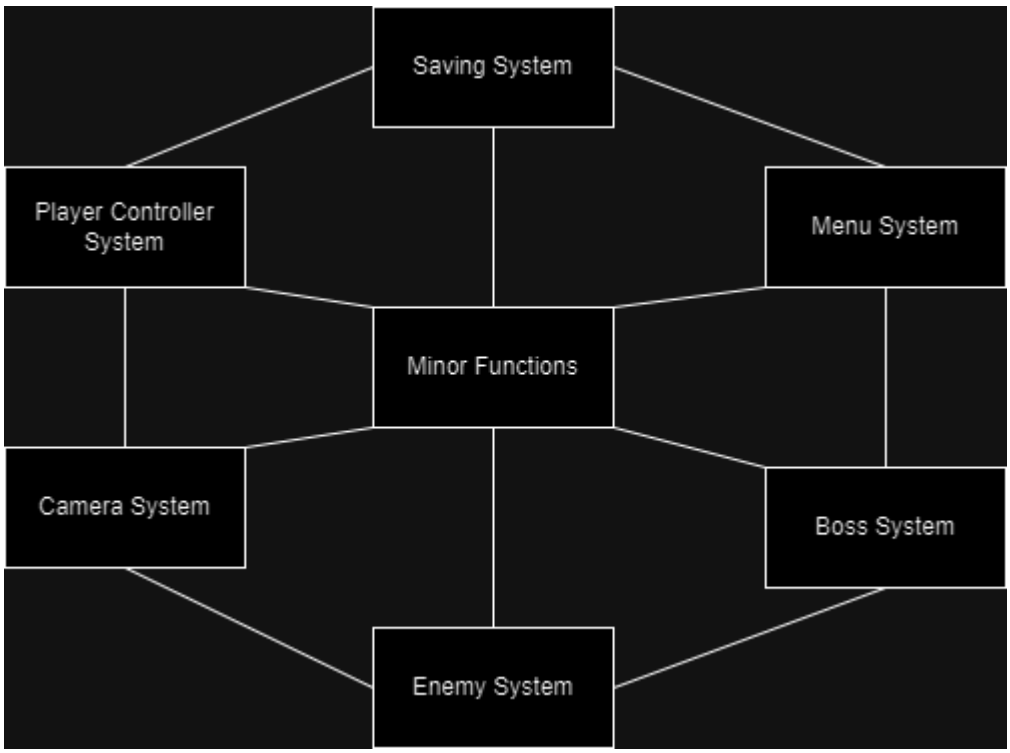


Figure 1: However the figure's caption is below.

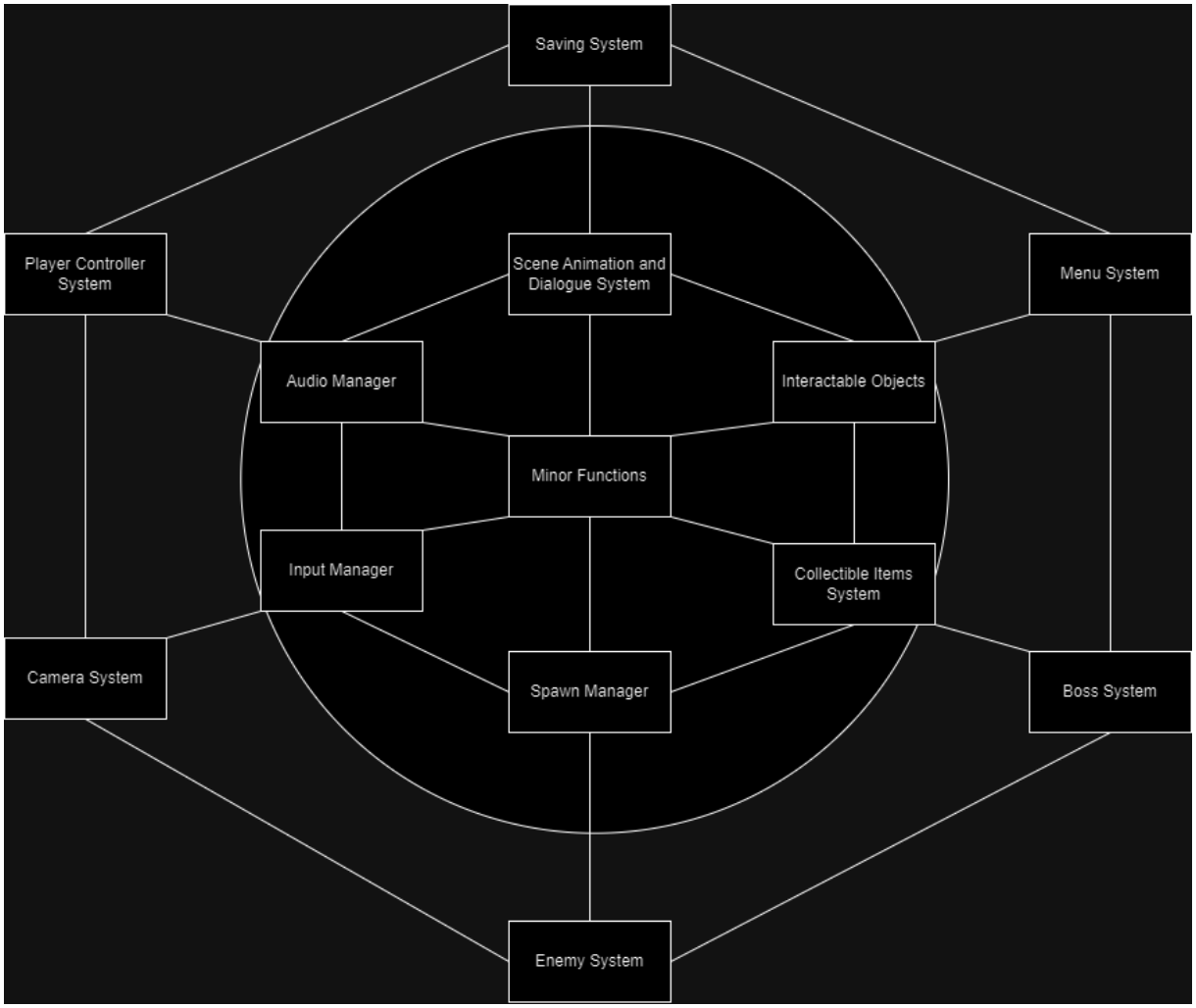


Figure 1: However the figure's caption is below.

Chapter 2: Preparation Process

2.1 Asset Preparation

The preparation process starts with preparing the asset. Some assets, we created on our own while the others, we borrowed and modified from free Unity Asset Store and internet to save time and efforts. After the assets are ready, we start programming and add animations to our characters.

In the figure below, the asset with containing individual terrain elements are originally packed in on PNG image. We slice them in the unity sprite editor and this is a memory effective way as each platform we create will reuse from this particular one image file.

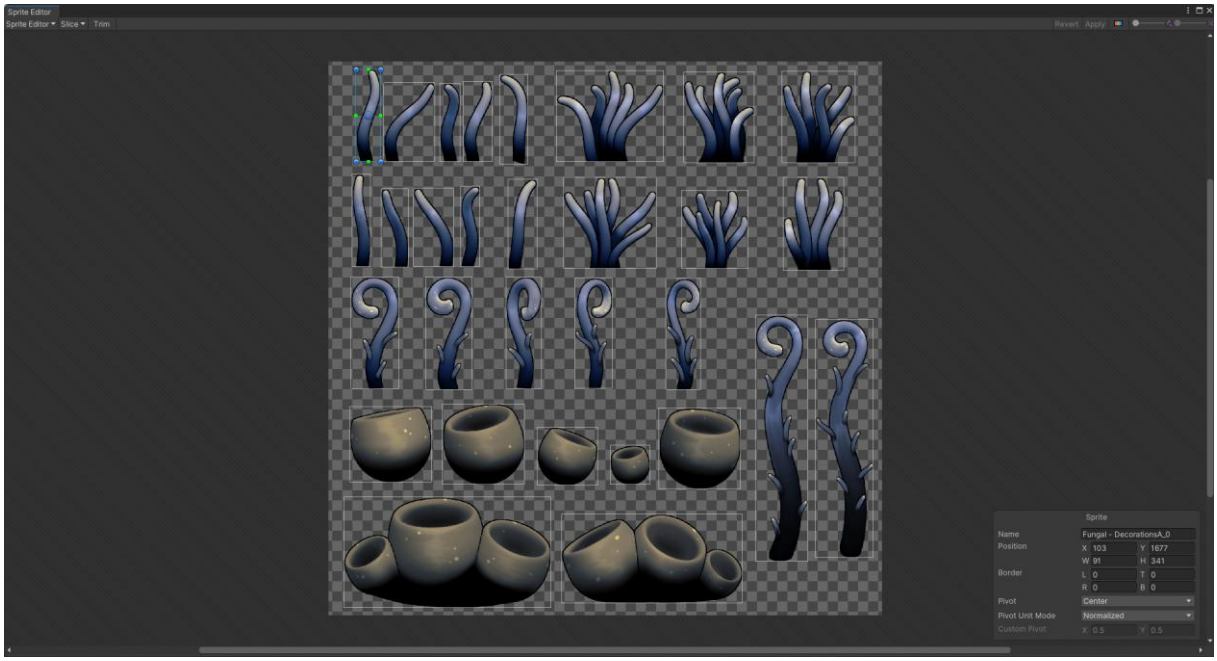


Figure 1: Mana on Left and Health bar on right

2.1 Animation Process

This is one of hardest processes for our game project that is not developer work. To be realistic, if we are creating a high visual game with story lines and cutscenes, we need a lot of animations and we cannot program them. This is usually the job of game artist and game designer in the team, and it should be impossible to make these things for our project, but our team has experience with these processes, so we made entire player animations for the lost boy. We made game animations in two different ways.

1. The industrial way (manual animation)
2. The Unity animation system (easy one)

1. The Industrial way

In the industrial way, everything is built up from the ground. First, we prepare those assets. Usually, the game artist will draw the assets in drawing software program and pack like below.

Figure 1: Mana on Left and Health bar on right

And they are imported into animation software which animators specially use for these types of 2D game projects. In our case, our choice is **Spine3D software.**

In Spine Interface, we have two modes, setup and animation. We set up the layers of our character and follow these 3 steps

1. Mange layers of the body parts
2. Change them into meshes and paint weights
3. Attach each part with rigging bones

Managing the layer makes animating so much easier and it is a very vital first step. For example, the stomach will cover the legs and the pant with cover them, one hand will be in front of the body, but the other hand will be behind as it is in side-view. The hair will have to cover the face, and the cape will cover the whole body. When we turn the raw image Png into mesh, we can build dotted mesh to move around the space and it is shown in figure. And we can't move those mesh dots one by one, we paint the weight and rig by the animation bones so moving one bone will move respective area of the body just like in human anatomy. This is how animation works in real life.

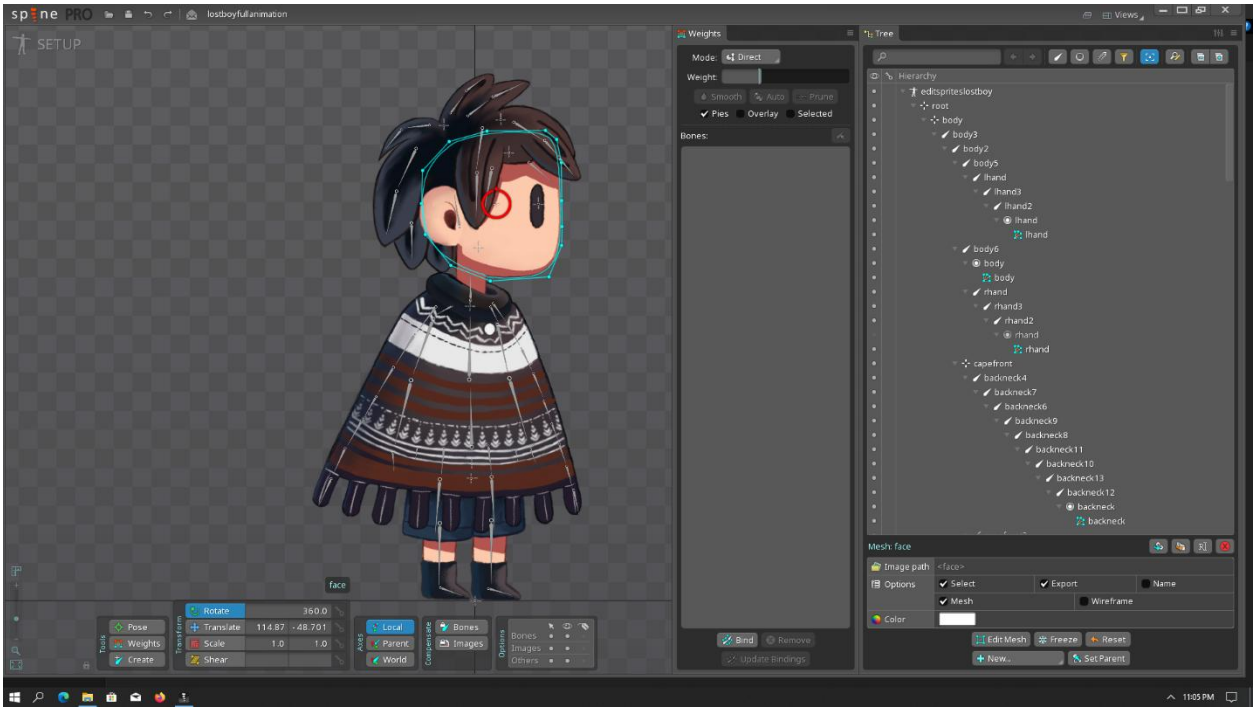


Figure 1: Mana on Left and Health bar on right

After that, each bone of the different body parts is moved to desired location to get different poses and recorded in different frames. This is a very time-consuming process to get perfect animation for each pose and repeat it again and again. The following snapshot is the animation frames of player jump from start to end

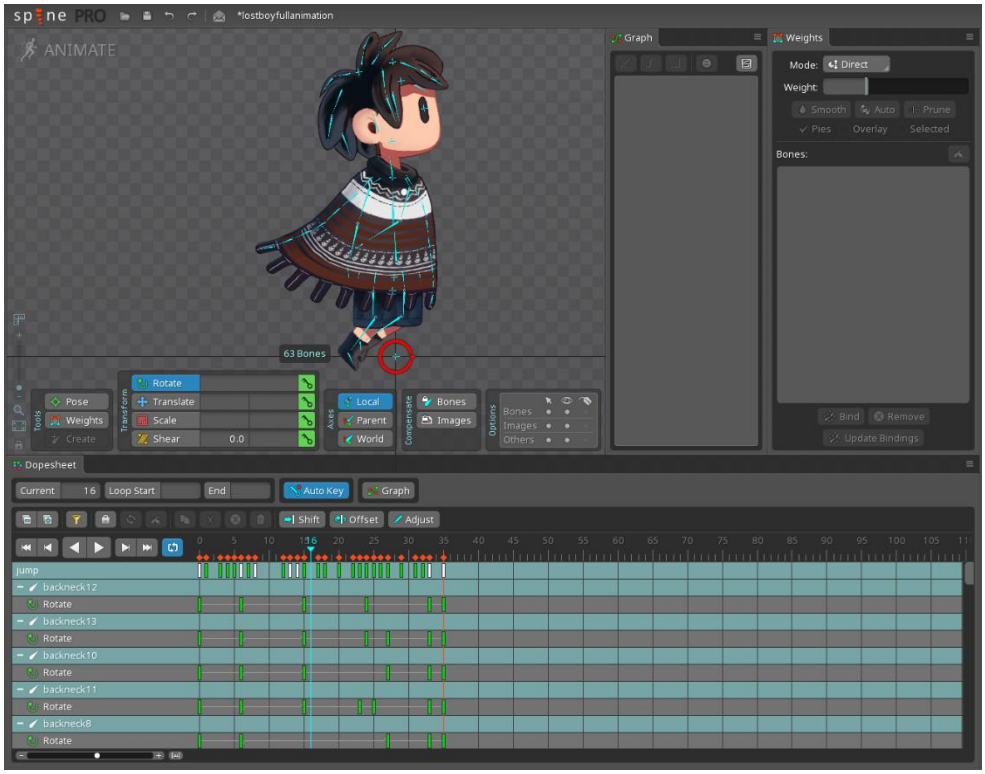


Figure 1: Mana on Left and Health bar on right

And finally, we export the frames into a file destination and move them into our unity project to process animation for each player's actions.

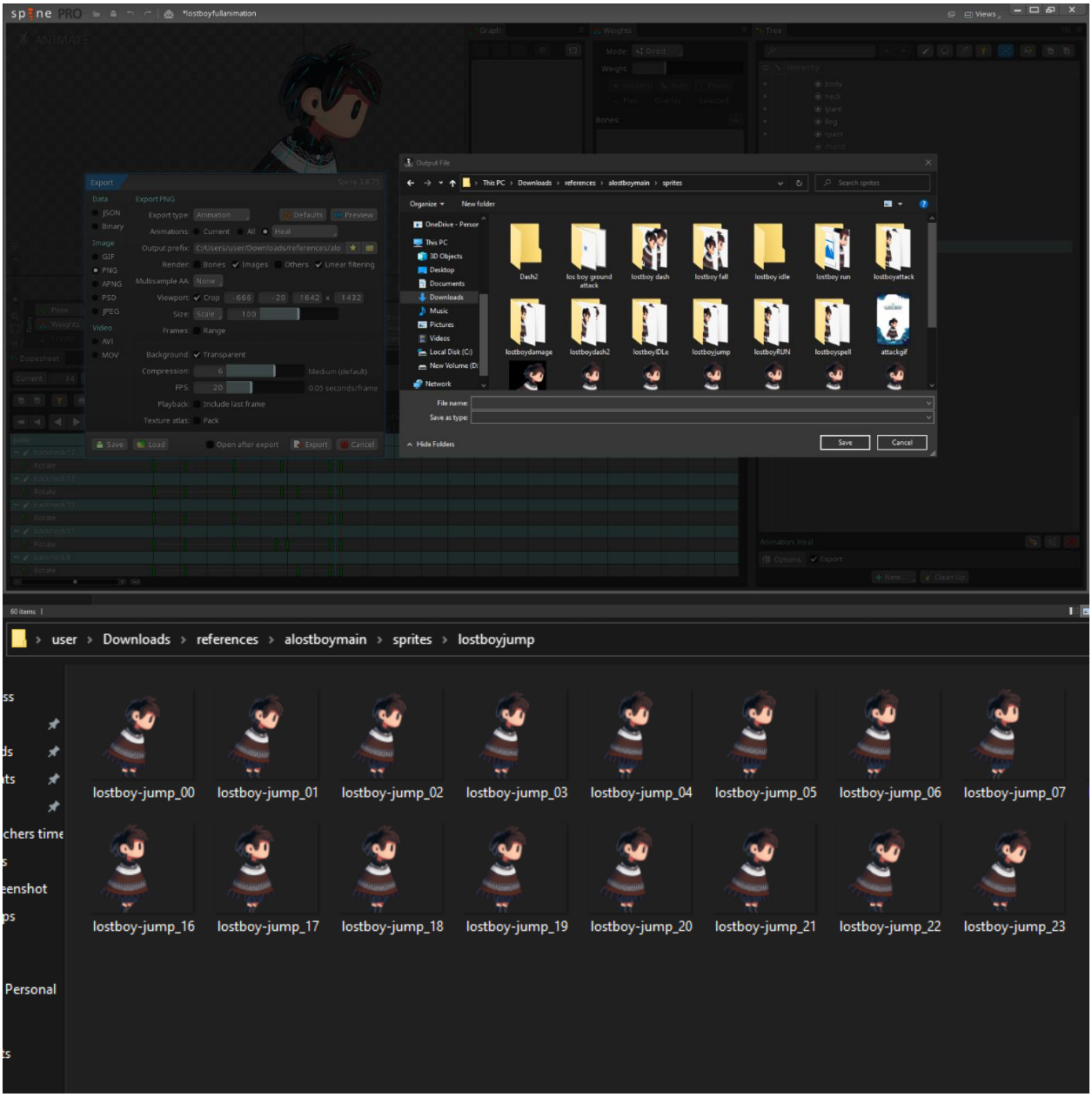


Figure 1: Mana on Left and Health bar on right

1. The Unity animation system

But let's be realistic, it is not very effective and easy to make animations for every character in the game like this detailing every frame of every pose. The more character you have, the more time consuming it will be, and we also find alternative easy ways for animating enemies.

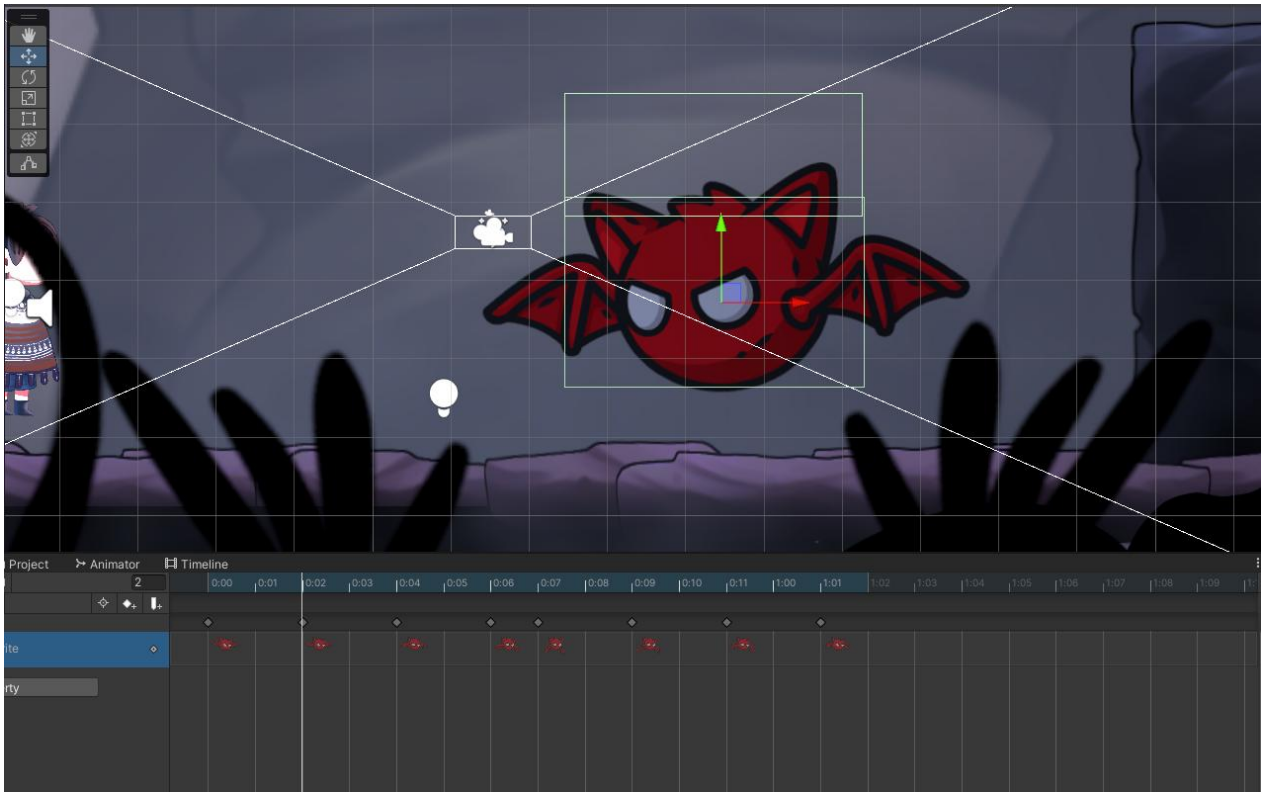


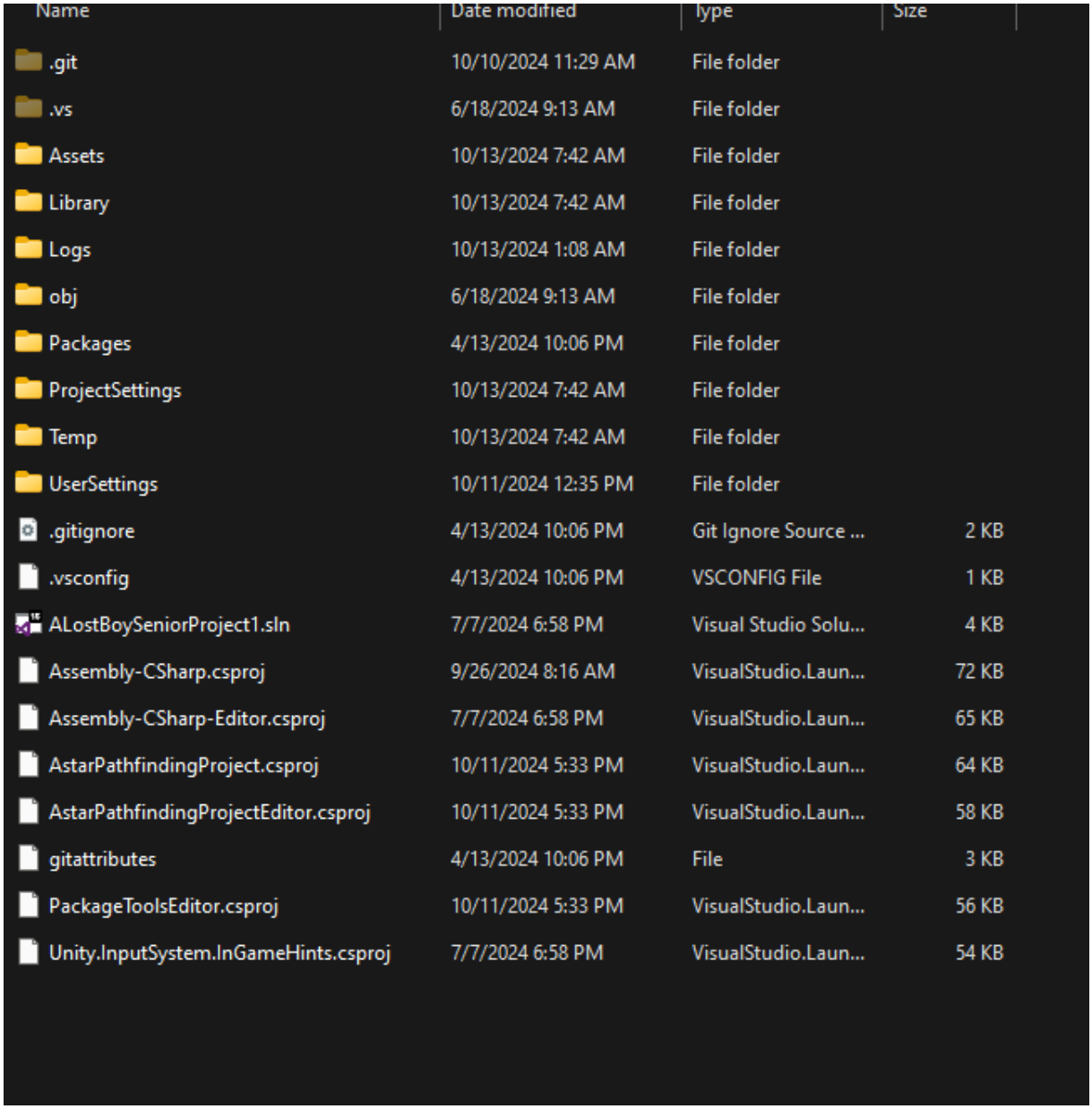
Figure 1: Mana on Left and Health bar on right

This method uses just one frame of the character and animates the moving positions, scales and rotations from frame start to end. Instead of taking several time and steps to manually hand animate, this is an easier and suitable way for indie game projects. However, it will not be as visually appealing as the manual way. We used combinations of manual and easy unity inbuilt animation system to achieve both quality and quantity.

2.1 Unity Project Setup

Setting up unity project is an easy step. Just create the project from 2D template and we are good to go. Assets folder is where most of our codes and graphic images and resources stay. We then push to the git hub for the first time and then git hub will remember every time we make changes.

Notes: Gitignore file is important here as it ignore the build file of the project when pushing to git hub.



Name	Date modified	Type	Size
.git	10/10/2024 11:29 AM	File folder	
.vs	6/18/2024 9:13 AM	File folder	
Assets	10/13/2024 7:42 AM	File folder	
Library	10/13/2024 7:42 AM	File folder	
Logs	10/13/2024 1:08 AM	File folder	
obj	6/18/2024 9:13 AM	File folder	
Packages	4/13/2024 10:06 PM	File folder	
ProjectSettings	10/13/2024 7:42 AM	File folder	
Temp	10/13/2024 7:42 AM	File folder	
UserSettings	10/11/2024 12:35 PM	File folder	
.gitignore	4/13/2024 10:06 PM	Git Ignore Source ...	2 KB
.vsconfig	4/13/2024 10:06 PM	VSCONFIG File	1 KB
ALostBoySeniorProject1.sln	7/7/2024 6:58 PM	Visual Studio Solu...	4 KB
Assembly-CSharp.csproj	9/26/2024 8:16 AM	VisualStudio.Laun...	72 KB
Assembly-CSharp-Editor.csproj	7/7/2024 6:58 PM	VisualStudio.Laun...	65 KB
AstarPathfindingProject.csproj	10/11/2024 5:33 PM	VisualStudio.Laun...	64 KB
AstarPathfindingProjectEditor.csproj	10/11/2024 5:33 PM	VisualStudio.Laun...	58 KB
gitattributes	4/13/2024 10:06 PM	File	3 KB
PackageToolsEditor.csproj	10/11/2024 5:33 PM	VisualStudio.Laun...	56 KB
Unity.InputSystem.InGameHints.csproj	7/7/2024 6:58 PM	VisualStudio.Laun...	54 KB

Figure 1: Mana on Left and Health bar on right

2.1 Project Structure

We have a lot of things happening in our project so it is best to have clean structure.

1. Our main player is at the top of hierarchy
2. Pause Menu Canvas to control the UI of the game
3. Light Collections to manage light settings
4. Camera control System
5. Bounds is the limit of where the player can see in the game
6. Checkpoints collection
7. Spawn manager responsible for enemies
8. Enemies and Doors (locked) collections

9. Level Design for the platforms and walls
10. NPC character collections
11. Animation Timeline
12. Event System to control the Input and UI

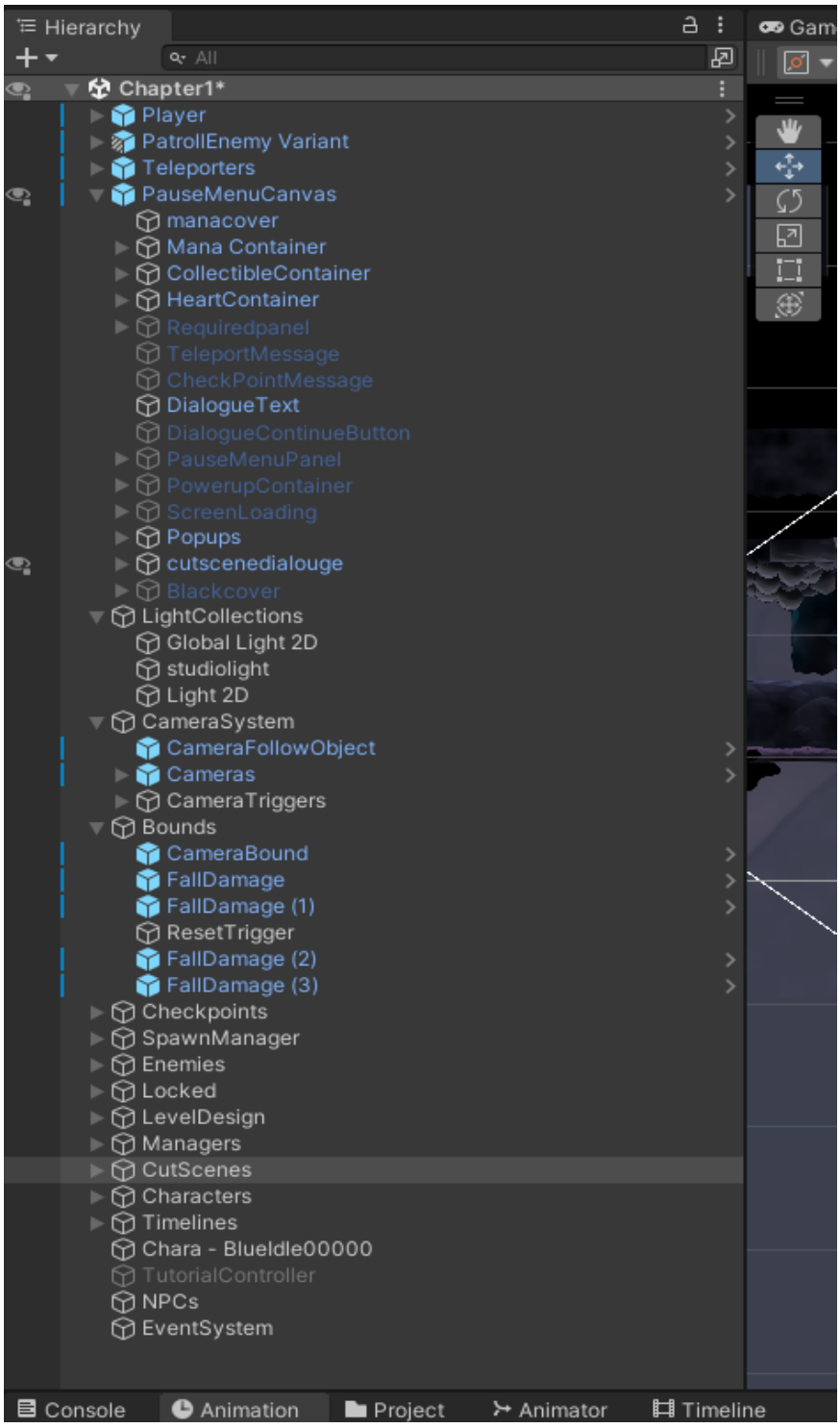


Figure 1: Mana on Left and Health bar on right

2.1 Git Hub Version Control

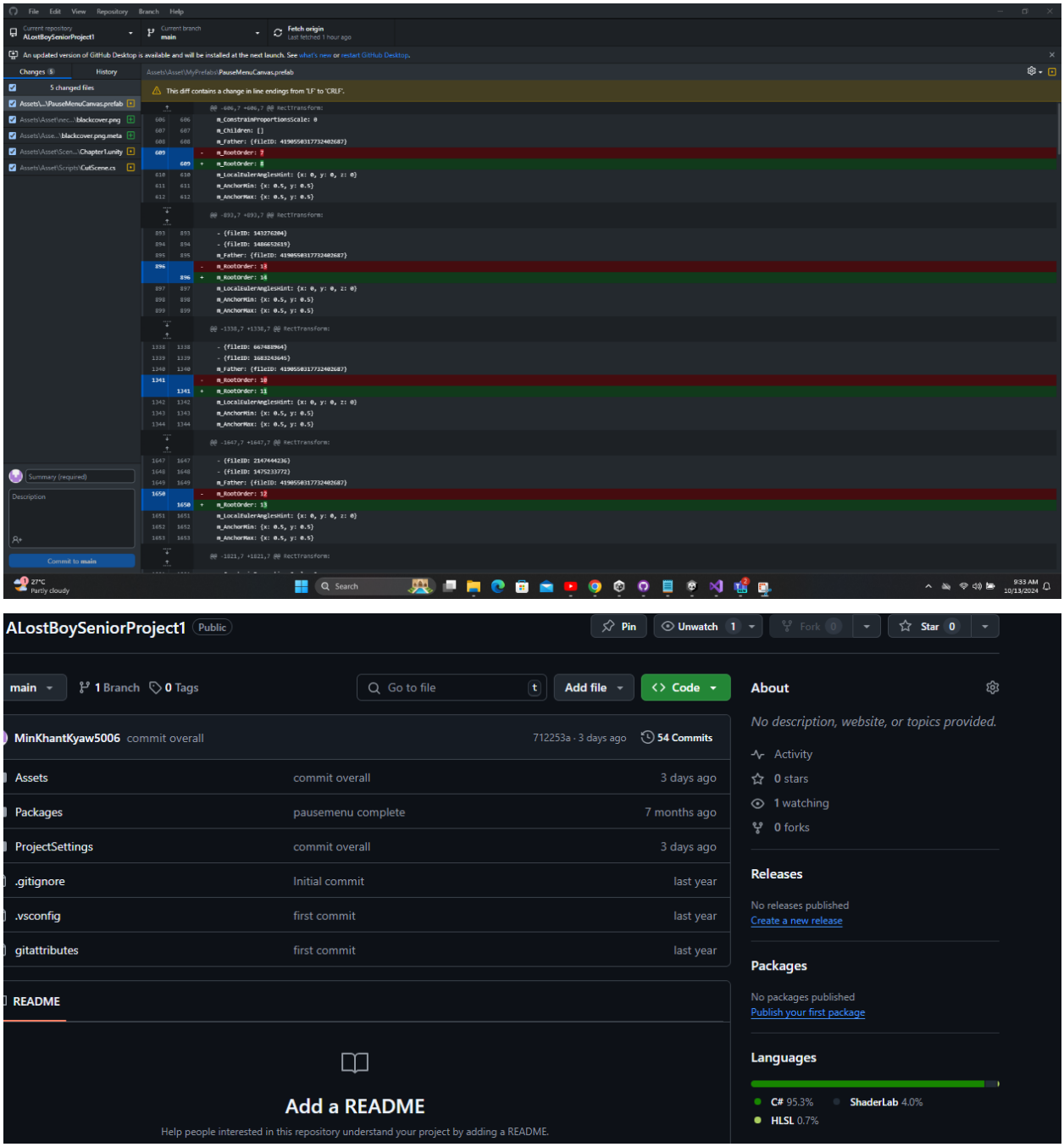


Figure 1: Mana on Left and Health bar on right

2.1 Tech Stacks

Sketch book: The first software to use for preparing drawing assets of characters, backgrounds, platforms and other objects.

Blender: This software is used for making postprocessing effects like sword slash, damage flashing, fire, snow, dust and other special effects.

Spine: The main software for all animation jobs.

Unity: The Game Development platform for final product

Figure 1: Mana on Left and Health bar on right

Chapter 3: Features

3.1 Player Controls and Combat System

Player Combat System: Players engage with enemies using a variety of combat mechanics, ranging from simple melee attacks to more complex moves involving combos and special abilities. The depth of combat can vary, offering everything from basic button presses to intricate systems that reward skillful execution.

Players have a **health bar** that decreases when they take damage from enemies or environmental hazards. Once their health reaches zero, they die, triggering a **death** animation, and then **respawn** at a designated checkpoint. In addition to health, player has a **mana** bar, which is used for special abilities and spells. Mana is typically filled when player attacks and kills enemies. Finally, Player have Inventory Lists of collectible items, gems (gold), diamonds, jade and red stones.

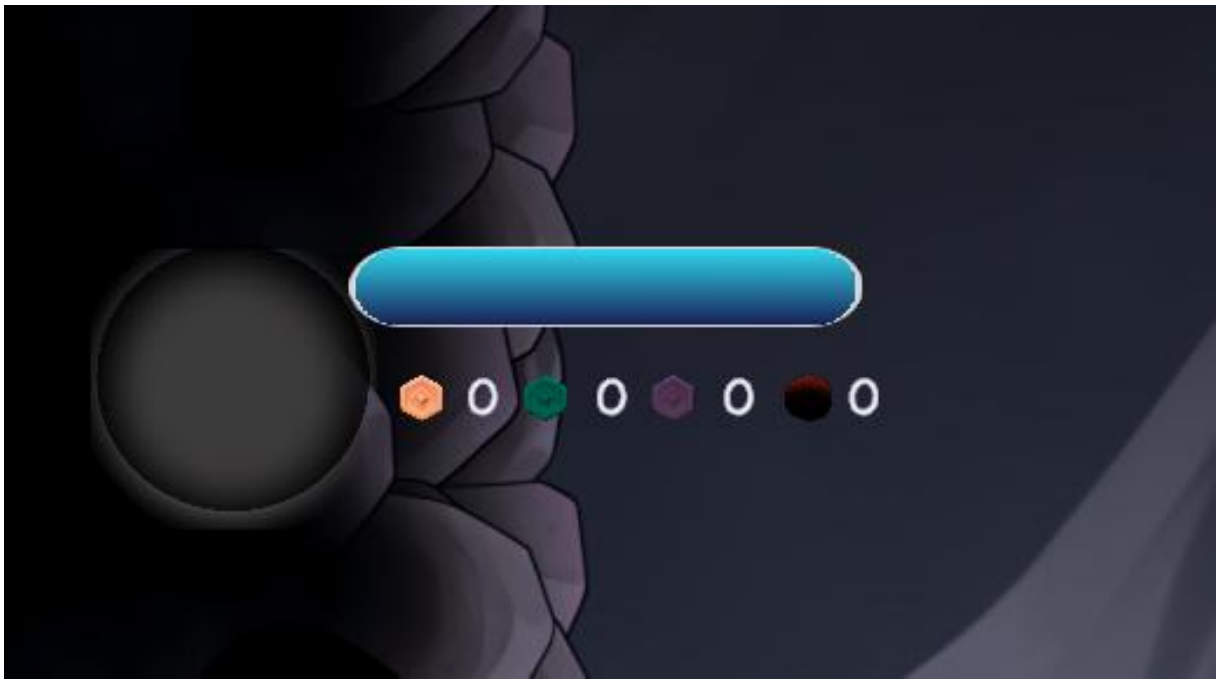


Figure 1: Mana on Left and Health bar on right

In terms of combat, players can perform **four-directional melee attacks**, allowing them to strike up, down, left, or right with slash attacks. These attacks can often be chained into **combos** for increased damage.

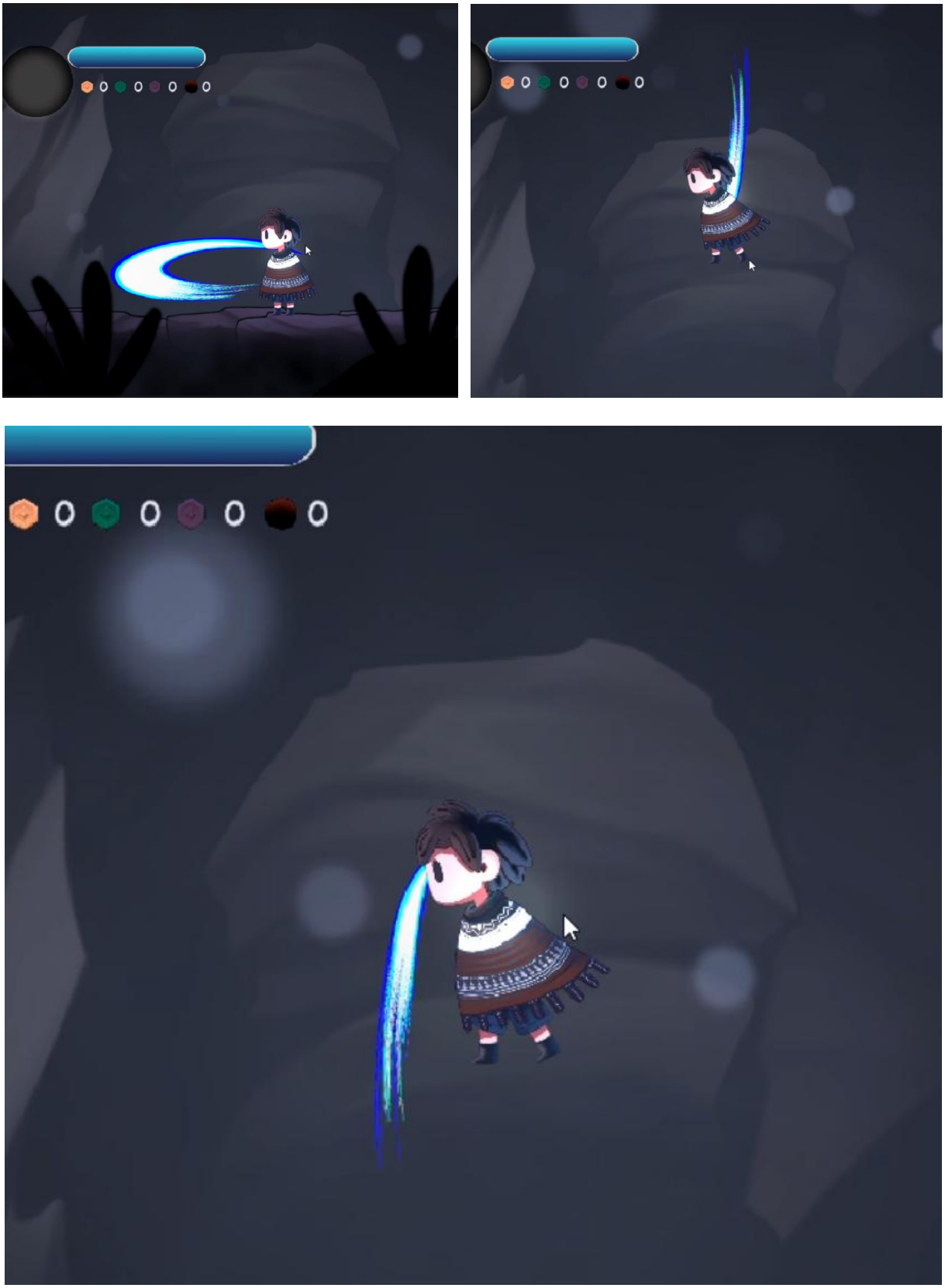


Figure 2: Sides, Up and Down Attacks

To enhance mobility, players have access to a **dash** ability, which lets them quickly evade enemy attacks or close in on targets. Additionally, players can **jump** and perform a **double jump**, and use it with **dash**, adding verticality to combat and allowing them to avoid obstacles or reach higher ground.

Figure 3: Run, Jump and Dash Actions

When players take damage, they can use **healing items** or abilities to restore their health, providing opportunities for recovery during or after combat. Finally, for more powerful attacks, players can unleash **directional superpowers**—special abilities aimed in specific directions that deal significant damage or have other impactful effects. These superpowers typically consume mana and can help players turn the tide of battle. Better to see in Game Play.

With this combination of health management, dynamic movement, melee and ranged attacks, and powerful special abilities, the combat system offers a versatile and engaging experience. Players can adapt their strategies based on the situation, making each encounter feel unique and challenging.

3.2 Check Point Saving System

Checkpoint Saving: The checkpoint system allows the game to save the player's progress at specific points, ensuring that players can return to these spots after certain events, such as losing a life, restarting the game, or encountering unexpected issues like a crash. When players reach a checkpoint, the game's state is automatically saved, capturing their position, inventory, and other relevant progress. This system is **our heart of the game** and the most difficult system to implement. We have to implement so many functions for this system just to see the line “checkpoint saved!” and see if it is actually saved in local data.

Figure 4: Check Point Saved

If the player **returns to the main menu**, **quits the game**, or if the game **crashes**, they will begin from the last checkpoint when they resume. Similarly, if the player **dies** during gameplay, they will be respawned at the most recent checkpoint, ensuring they don't lose significant progress.

The system also works through the **pause menu**, where players can choose to **restart from the last checkpoint**, bringing them back to the most recent saved location without resetting the entire game. However, if they choose to **restart the chapter**, the game will bring them back to the **very first checkpoint** in that scene, providing a fresh start from the beginning of the chapter. The hardest thing about saving system is it has to work with all the other systems, player, main menu and enemies. But the most important partner is **Menu System** which will be covered next.

3.3 Menu System

Menu System Overview: The game features two primary menus: The **Main Menu** and the **Pause Menu**, each offering different options for managing gameplay and progress.

Main Menu

The **Main Menu** is presented when the player starts the game and offers the following options:

1. **New Game**: Players can start a fresh playthrough and save their progress in one of four available save slots. Each save slot records the player's journey, allowing them to manage multiple playthroughs or different progress points.
2. **Continue Game**: This option lets the player pick up from their **most recent saved progress**, either from the latest checkpoint or saved location in one of the active save slots. It provides a quick way to jump back into the game without selecting a specific save slot.
3. **Load Game**: Players can manually select and load from one of the **four save slots**. Each slot contains a saved game at various progress points, giving players flexibility to revisit different parts of their playthrough or switch between different chapters.
4. **Quit**: Exits the game and returns the player to the desktop or console interface.


A Lost Boy
New Game
Continue
Load Game
Quit


Game in Progress
Chapter3
Game in Progress
Chapter2
Game in Progress
Chapter3
Game in Progress
Chapter1
Back

Figure 5: Main Menu and save slots

Pause Menu

The **Pause Menu** is accessible during gameplay and allows players to manage their current session without exiting the game entirely. The following options are available:

1. **Resume**: Allows players to return to the game, continuing from the exact point where they paused, without any interruptions.
2. **Restart from Checkpoint**: If the player chooses this option, the game will reload the **last checkpoint** reached in the current session. This is useful when players want to retry a section after a difficult encounter or make better choices.
3. **Restart Chapter**: This option restarts the entire chapter or level, sending the player back to the **first checkpoint** of that chapter. This provides a fresh start within the current section of the game for those who wish to redo everything from the beginning of the chapter.
4. **Main Menu**: This option exits the current session and returns the player to the Main Menu. This will only save the last checkpoint so anything after that checkpoint will not be saved if they go back to main menu.

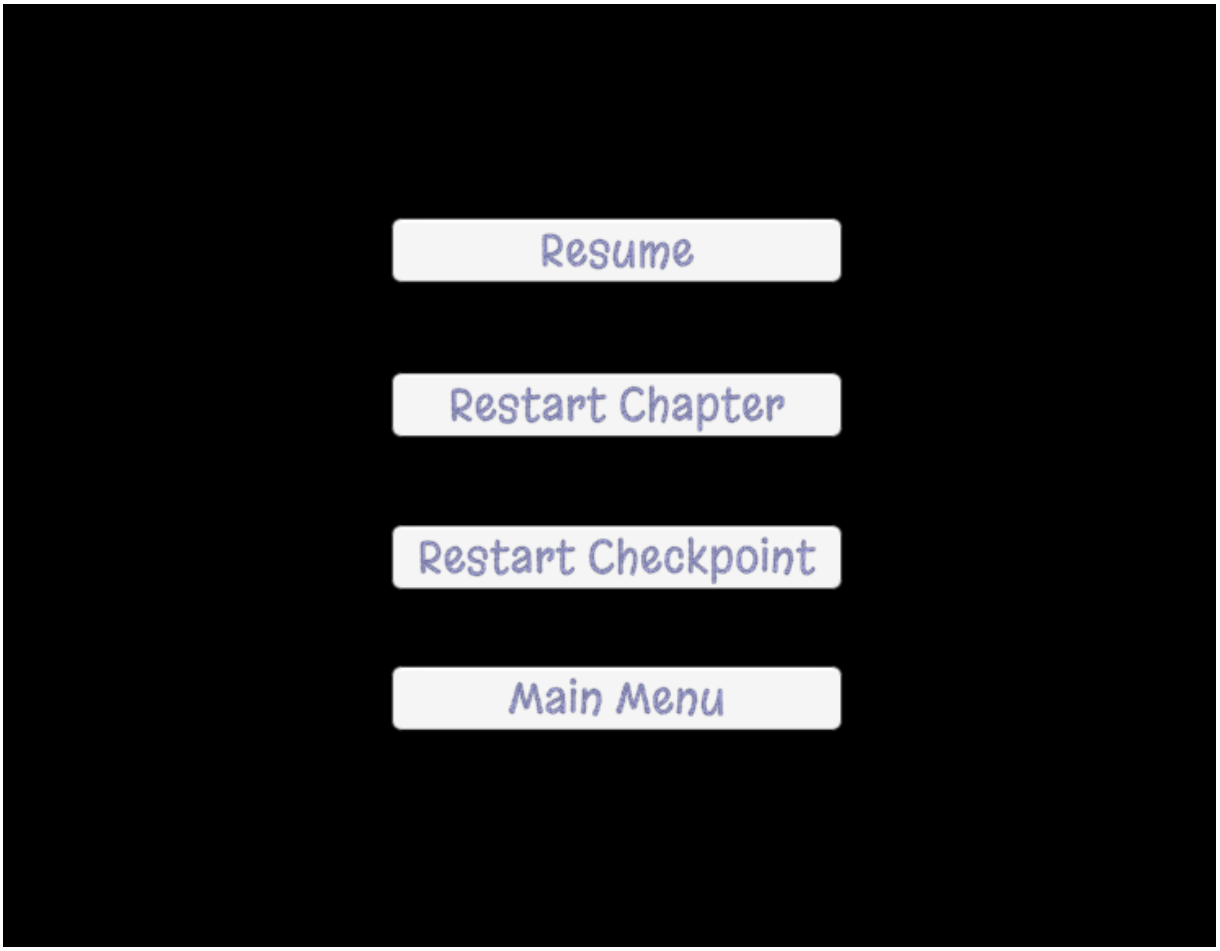


Figure 6: Pause Menu

3.4 Advance Camera system

Why call it an "Advanced Camera System" rather than just a simple "Camera System"? In many classic 2D games like **Super Mario**, the camera follows the player with a basic, static approach—always centered on the player as they move left, right, or jump. While this is effective, it doesn't provide much depth or dynamic interaction with the environment, especially in modern game design, where storytelling, exploration, and immersion are key elements.

Our **Advanced Camera System** goes beyond the basics, providing a more immersive experience through several nuanced behaviors. Here's how it differs:

1. **Delayed Camera Follow**: Unlike static camera systems, this method introduces a slight delay when the player moves, creating a subtle "catch-up" effect. As the player runs or jumps, the camera doesn't snap immediately to follow, but rather drifts slightly behind. This gives a sense that the camera is almost "following" the player, making the movement feel smoother and more natural. It also enhances immersion by creating a dynamic relationship between the player and their perspective of the game world.

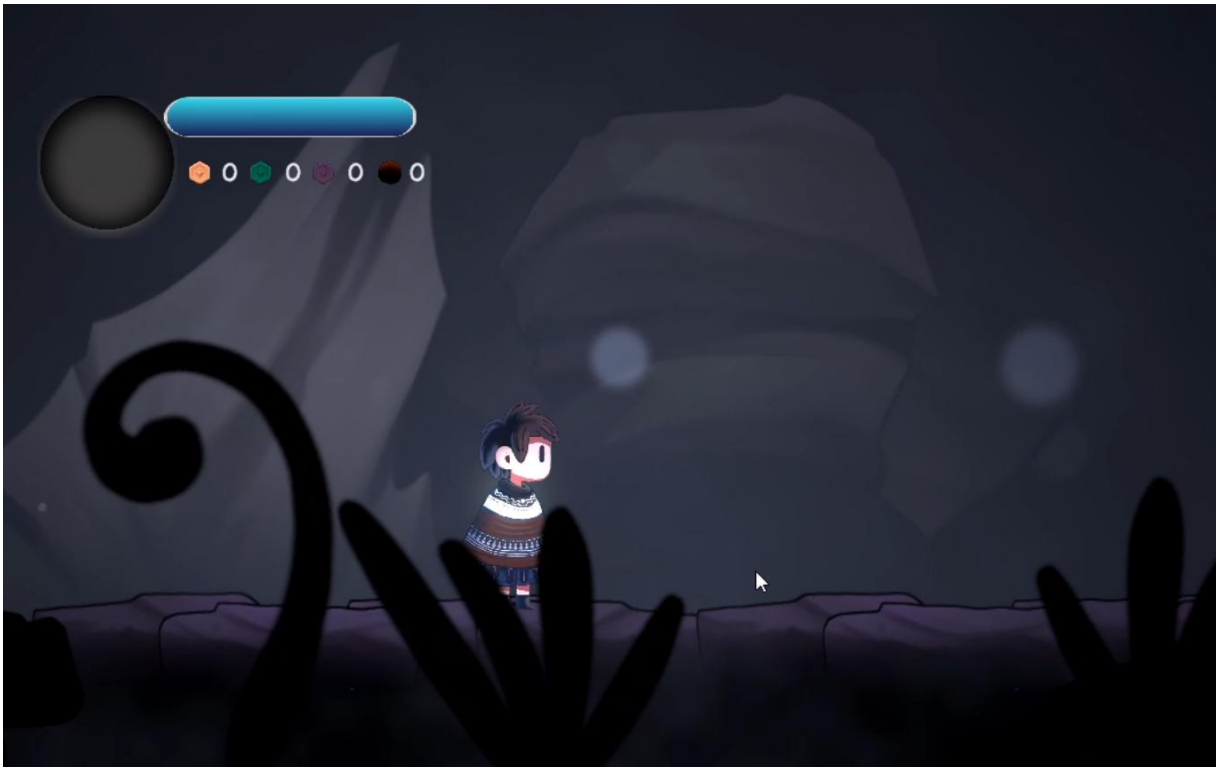


Figure 7: Delayed Camera and not centered yet

2. **Camera Zoom In and Out**: At certain points in the game, the camera can zoom in or out based on triggers placed in the environment. Zooming in can be used to focus on

critical story elements, drawing attention to a character or action, while zooming out can provide a better view of larger scenes or areas. This dynamic zooming allows for more cinematic storytelling or can simply help make large sections of a level more visible to the player.

Figure 8: Zoom Out Camera

3. **Camera Pan in Four Directions**: The camera can pan in any of the four directions (up, down, left, right) to reveal more of the scene, guiding the player's exploration. For example, if the player reaches the edge of a cliff, the camera might pan downward to show that there's a platform below, signaling that it's safe to jump. This system helps players anticipate what's ahead or below them, enhancing both exploration and decision-making.

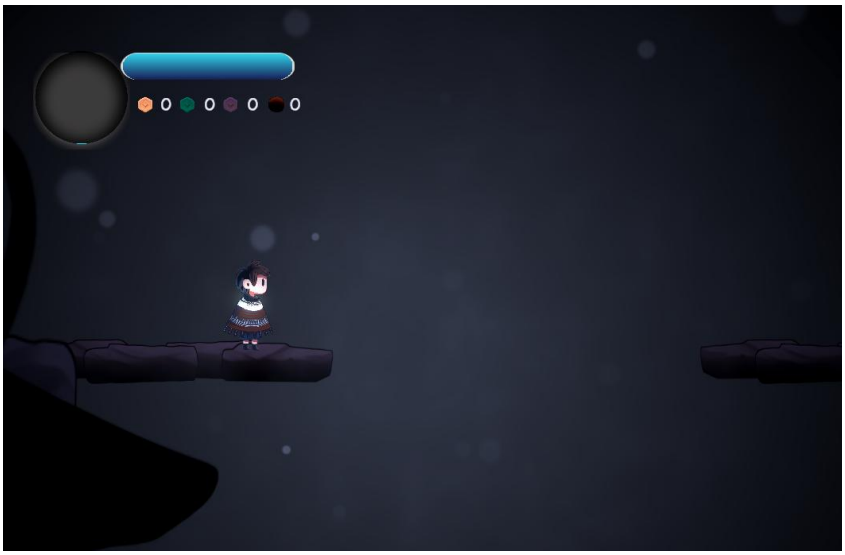


Figure 10: Camera pans right to show another platform

4. **Mix of Zoom and Pan**: In some situations, zooming and panning can be combined for greater effect. For instance, the camera might zoom out and pan across a large scene to reveal a significant portion of the level, perhaps to show the player's next objective or hint at what's coming up.

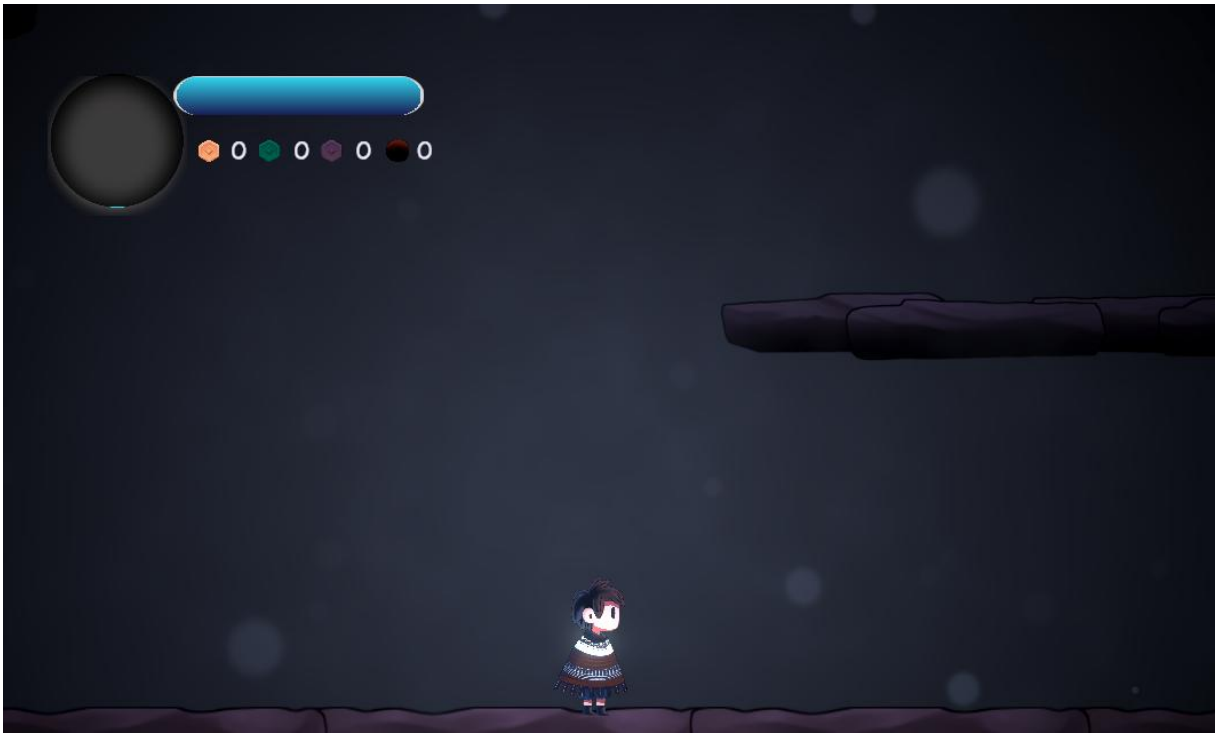


Figure 11: Panning up while zooming out

5. **Pan Fall**: When a player is falling from a height, the camera will respond more quickly than usual, panning down rapidly to show what lies beneath. This is the opposite of the delayed camera follow—when the player is in freefall, they need to see what's below to make quick decisions. Whether it's to prepare for a safe landing or avoid danger, the faster camera movement ensures that the player is never left guessing.

Figure 12: Camera showing ground even before player lands

6. **Camera Boundaries**: The camera is restricted by boundaries within each scene, meaning it won't show parts of the level that are out of reach or irrelevant to the player. For example, if the player reaches a solid wall, the camera will stop moving when the player does, ensuring that off-limits areas beyond the wall remain hidden. This maintains immersion by keeping the focus on the playable environment and prevents awkward views of the edges of the game world.

7. **Camera Shake**: To enhance the impact of combat, the camera incorporates a **shake effect** whenever the player successfully lands a hit on an enemy. This subtle shake creates a sense of power and impact, adding weight to attacks and making combat feel more visceral. The intensity of the camera shake can vary depending on the strength of the attack or the enemy being hit. It helps convey the physicality of the action, making every hit feel satisfying and dynamic.

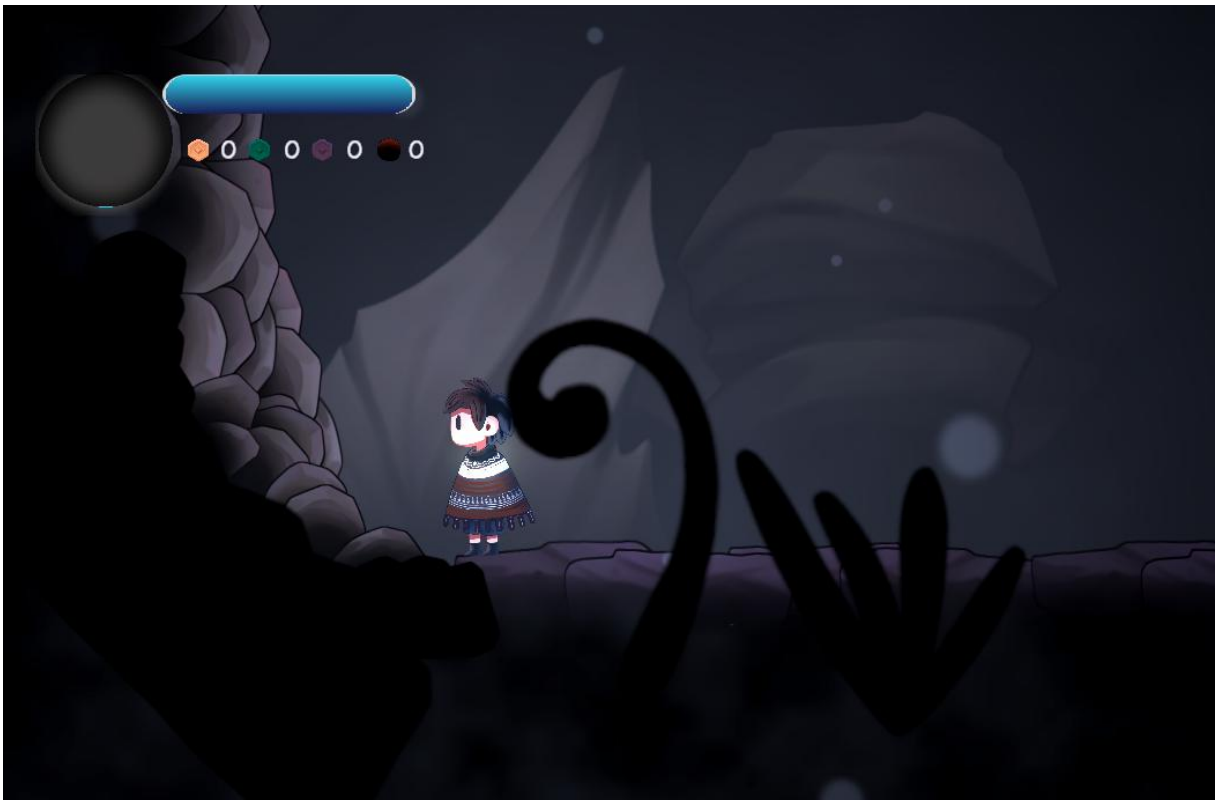


Figure 13: Camera won't show what's outside the wall

This **Advanced Camera System** allows for a more dynamic, engaging, and interactive gameplay experience. Whether through subtle camera delays that create a sense of realism, zooms that emphasize important moments, or panning that aids in exploration, the system provides players with both functional and cinematic elements.

3.5 Enemy system

The **Enemy System** in our game introduces a variety of enemies, each with distinct behaviors and movement patterns to challenge the player in different ways. This diversity ensures that players must adapt their strategies based on the unique actions of each enemy type. Here's a breakdown of the enemies:

1. **Patrolling Enemy**: This is a standard ground-based enemy that moves back and forth within a fixed path. The patrolling enemy provides a simple challenge as it sticks to predictable, linear movement patterns. Players need to time their movements carefully to avoid detection or damage. This enemy often serves as the player's first encounter with hostile creatures and helps them learn basic avoidance techniques.

Figure 14: Patrolling Enemy

2. **Following Walking and Shooting Enemy**: This enemy combines two behaviors: it walks toward the player and shoots projectiles when within range. This creates a multi-faceted threat where the player must manage both dodging the enemy's movement and avoiding its projectiles. The balance between chasing the player and maintaining a shooting distance adds depth to the player's combat and evasive strategies.

Figure 14: Walking and Shooting Enemy

3. **Flying Follow Enemy**: A flying enemy that actively follows the player, constantly tracking their movement. Since it hovers above ground level, it adds verticality to the threat, forcing players to consider not only lateral movement but also how they navigate higher platforms or obstacles to evade pursuit. These enemies are relentless, often chasing the player across different terrains.

Figure 15: Flying Enemy

4. **Flying and Following Shooting Enemy**: A more advanced version of the flying enemy, this type not only follows the player but also fires projectiles. This combination makes the enemy especially dangerous, as it pressures the player both vertically and horizontally. Players must be agile, dodging the projectiles while finding ways to outmaneuver the enemy in mid-air or across platforms.

Figure 16: Flying Shooting Enemy and its projectile

5. **Small Spider (Straight)**: The small spider moves vertically up and down along a set path, often positioned on walls or ceilings. Its limited range of motion makes it predictable, but its speed or placement in tight areas can make it tricky to avoid. Players need to time their jumps or movements to bypass these creatures, often in narrow spaces where vertical movement is key.

Figure 17: Small Spider

6. **Big Spider (Horizontal/Diagonal Movement)**: Unlike the small spider, the big spider moves in a straight line either horizontally or diagonally. This type of spider covers more ground and may move faster, providing a greater threat to players navigating wide-open spaces. The diagonal movement adds an extra layer of challenge, as players need to predict the spider's path more carefully compared to those that move in simple vertical or horizontal patterns.

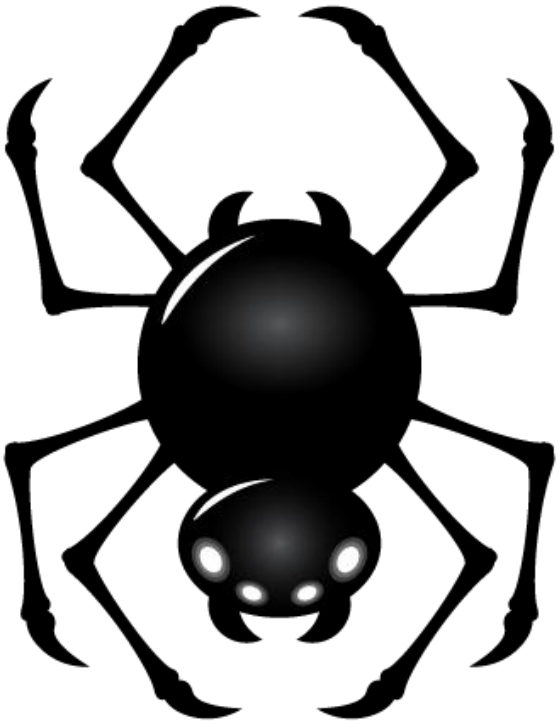


Figure 18: Big Spider

7. **360 Platform Rotating Enemy**: This enemy is attached to a platform and rotates 360 degrees around it. Players must time their movements carefully to avoid being hit as they traverse the platform. The rotating enemy tests the player's patience and precision, particularly in platforming sections where the enemy's constant movement makes jumping to or from the platform dangerous. The player needs to predict the rotation and plan their actions accordingly.

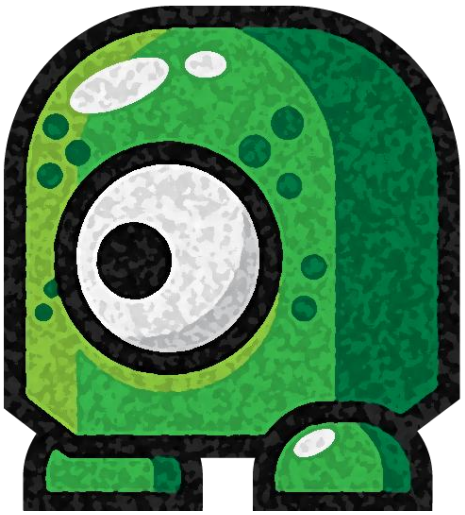


Figure 19: 360-degree rotating enemy

The **Enemy System** adds depth and complexity to the game's combat and traversal mechanics. Each enemy type introduces different threats that require the player to adapt their approach. So far, we have 7 enemies. The 8th enemy will be our final boss.

3.6 Boss Fight AI

The **Boss Fight System** in our game is designed to be dynamic, challenging, and unpredictable, making each encounter feel fresh and exciting. The boss engages the player through a variety of complex behaviors and attack patterns that evolve over three stages. Here's a detailed look at how the boss operates:

Figure 20: Boss

1. **Boss Idle Mode**: In between attacks or after specific sequences, the boss enters an idle state. This provides the player with a brief respite to anticipate the next move or take quick actions. The idle mode is a short window where the boss is not actively attacking or following the player but can transition into any action at any time.

Figure 21: Boss in Idle Mode

2. **Boss Run (Follow Player)**: In this mode, the boss actively pursues the player by running towards them. The boss can close gaps quickly, forcing the player to stay mobile and avoid getting cornered. The boss run is the foundation for its melee attacks, as it positions the boss within striking range of the player.

Figure 21: Boss in Idle Mode

Figure 22: Boss Lunge and Triple Slash

3. **Boss Lunge Attack**: If the player moves too far away, the boss will perform a fast, lunging attack with its sword. This move allows the boss to rapidly close the distance

and catch the player off guard. The lunge is powerful and covers a large distance, making it a dangerous move if the player doesn't react in time.

4. **Boss Triple Slash**: The boss performs a rapid sequence of three slashes with its sword. This attack requires quick dodging or blocking from the player, as it can deal significant damage if not properly avoided. The triple slash can happen while the boss is near the player, creating an intense combat situation.

5. **Boss Parry**: To add another layer of challenge, the boss can parry the player's attacks. When the player tries to land a hit, the boss may block or deflect it, leaving the player vulnerable to a counterattack. The parry move adds a strategic element, forcing players to think carefully about when and how to attack.

   (The above actions form the core of **Stage 1** of the boss fight, providing a balanced combination of offense and defense.)

Figure 23: Boss Parry and block player's attacks

6. **Boss Stun**: After taking enough damage or when specific conditions are met, the boss becomes stunned. During this time, the boss is vulnerable, giving the player an opportunity to land free hits or strategize. The stun period typically happens at the end of Stage 1 or during transitions between stages, allowing the player a chance to regroup.

Figure 24: Boss Stun

7. **Stage 2 Actions**: After the stun, the boss enters **Stage 2**, where its attacks become more aggressive and varied.

8. **Boss Jump and Knock to the Ground with Sword**: In Stage 2, the boss adds a jump attack where it leaps into the air and slams down with its sword, causing a shockwave that can knock the player back if they are too close. This move increases the boss's threat range and forces the player to be mindful of both vertical and horizontal distances.

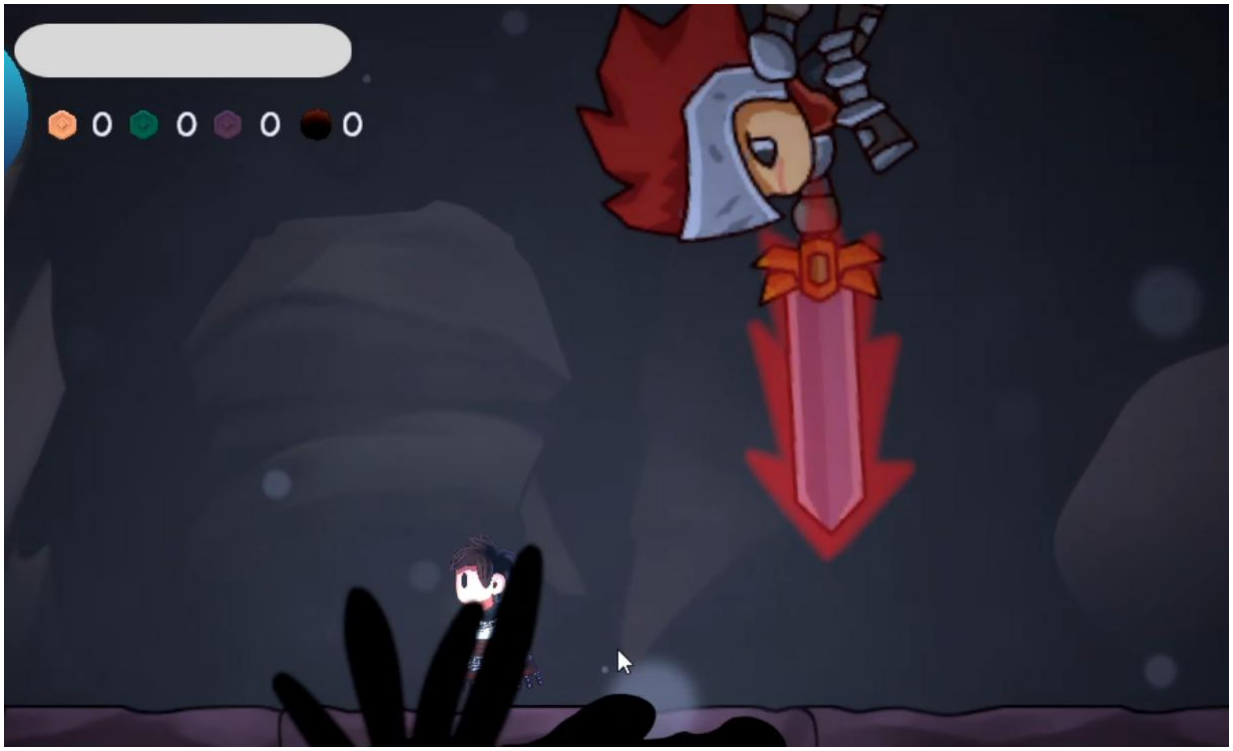


Figure 25: Boss Jump down attack

9. **Boss Throws Fireballs**: Another new attack in Stage 2 is the ability to throw fireballs at the player. These projectiles create a ranged threat that requires the player to dodge or block, adding complexity to the fight. The fireball attack can be used in combination with other moves to pressure the player from a distance.

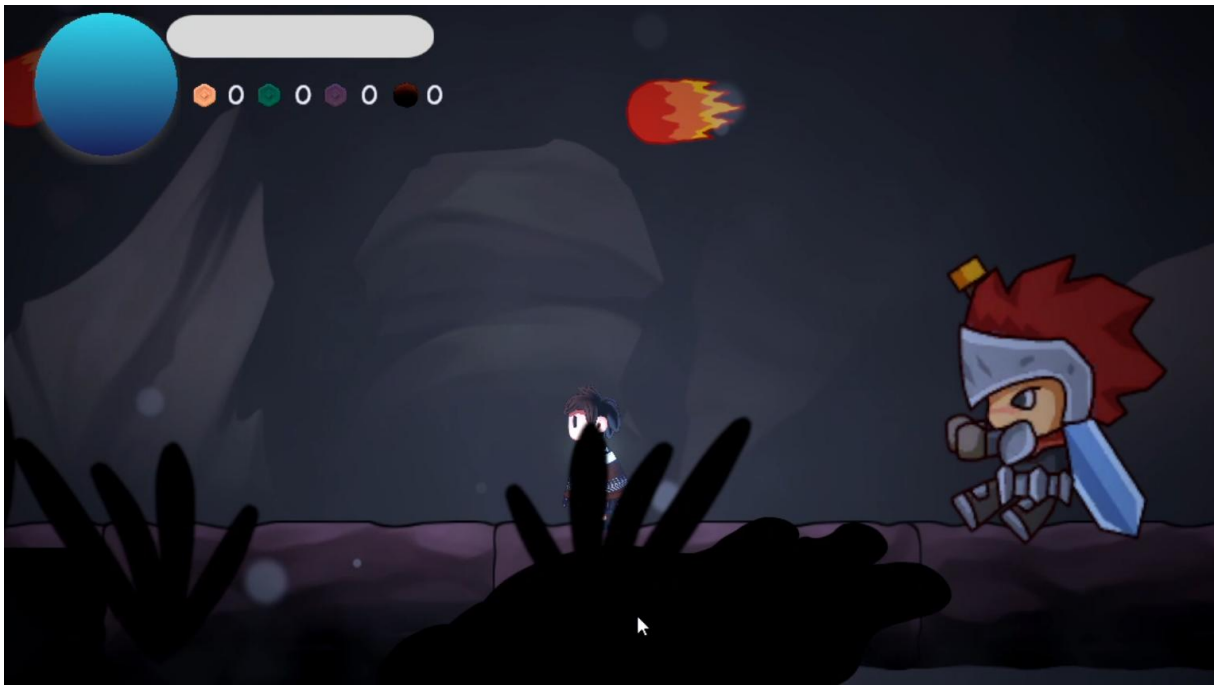


Figure 25: Boss Throw Fireball

10. **Stage 3 Actions**: In **Stage 3**, the boss becomes even more dangerous, utilizing all the previous moves while adding new abilities that make it harder to predict and counter.

11. **Boss Floats in Air and Throws Fireballs**: In the final stage, the boss can float in the air while throwing fireballs, in **3 directions**, making it difficult for the player to reach it with melee attacks. The player must find creative ways to deal damage while dodging fireballs from above. This introduces a whole new dimension to the fight, as the player now has to contend with an airborne boss. Besides, all the **other moves above** will also include again **from time to time**.

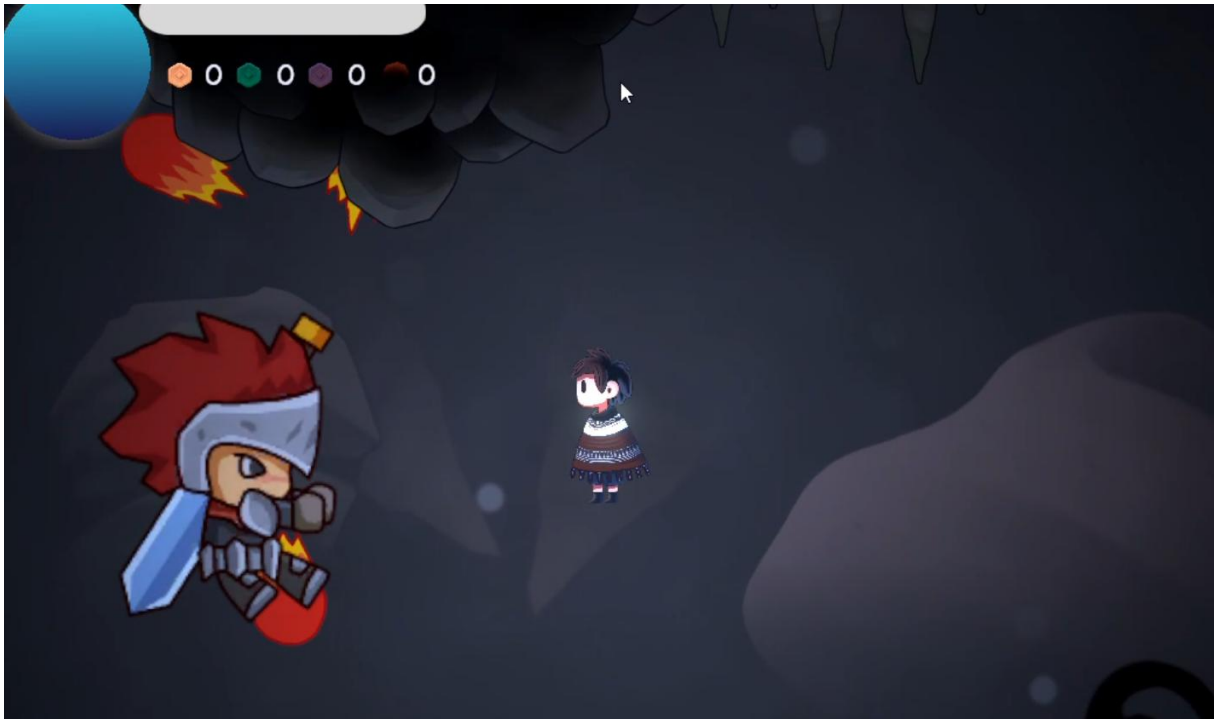


Figure 26: Boss Fly and throw fireballs in 3 directions

12. **Boss Death**: Once the player has dealt enough damage through all stages, the boss will finally die, ending the fight. The death sequence marks the end of the battle and rewards the player for successfully navigating the complex and unpredictable attack patterns.

Figure 27: Boss Die and Disappear

13. **Unpredictable AI Attack Patterns**: To keep the fight dynamic and prevent the player from memorizing attack sequences, the boss fight incorporates AI-driven randomness. The AI scrambles the boss's actions, ensuring that attacks are not always repeated in the same order or pattern. This unpredictability makes each encounter unique, requiring the player to stay on their toes and react quickly to whatever the boss throws at them. Boss fights have total of 10 actions, Idle, Run, Lunge, Parry, Triple Slash, Sword Bend Down, Barrage fireballs, Fly Barrage, Stun, Die.

3.7 Four Stages of Level Designs

Level Design: This Part of the Chapter is all about creativity and artistic skills instead of technical part. The game has **four stages**, each with a **different terrain**. Stage 1 is set in **a cave** with narrow paths and obstacles. Stage 2 moves to **a rainforest**, with dense trees and natural challenges. Stage 3 takes place in **mushroom valley**, with large mushrooms and tricky platforms. Stage 4 returns to a cave for the boss fight. Each stage has a distinct environment but stays straightforward in design.

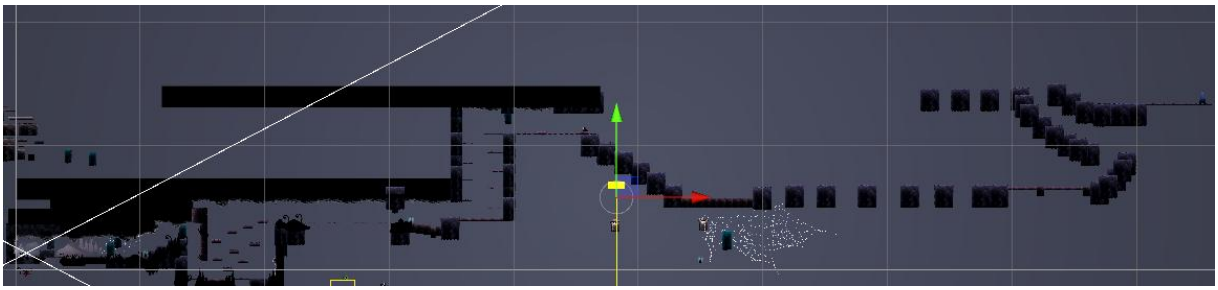


Figure 28: Level 1 Map

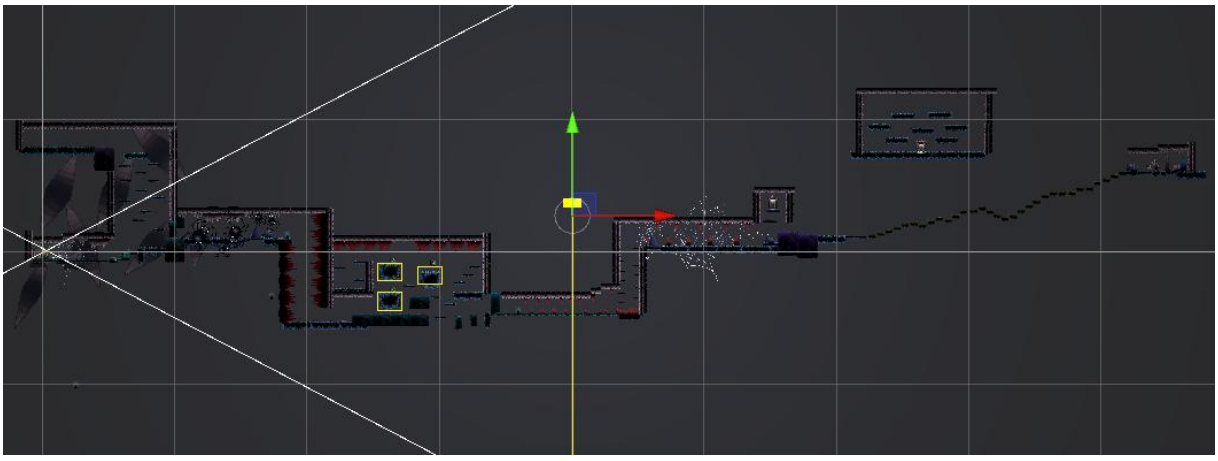


Figure 29: Level 2 Map

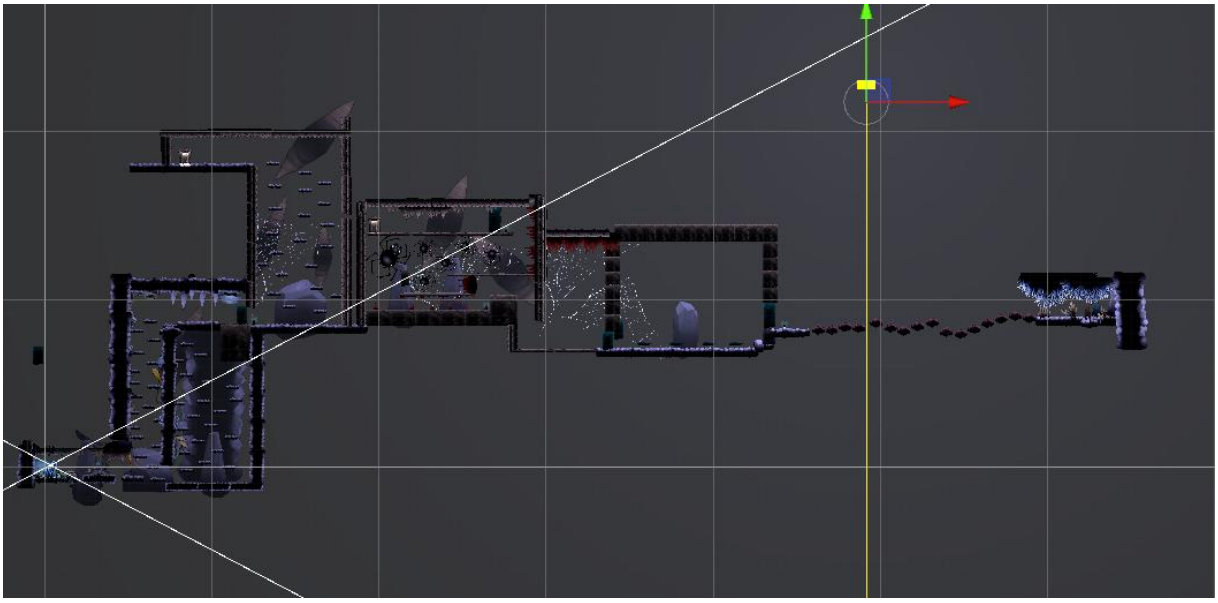


Figure 30: Level 3 Map

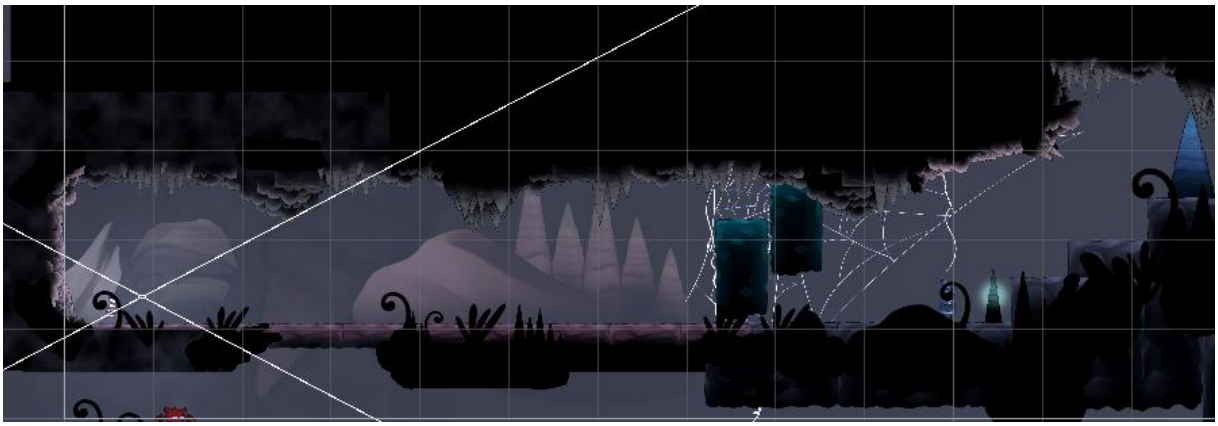


Figure 31: Level 4 Map

Want details closeup map? Play the Game!

3.8 Scene Animation and Dialogue System

The **Scene Animation and Dialogue System** in our game has three key components. First, **cutscene animations** play when the player enters certain areas, triggering cinematic moments where animations unfold, and black bars appear at the top and bottom of the screen (often called **cinematic widescreen effect**) to create a movie-like feel. Second, during **NPC encounters**, dialogue appears on the screen, allowing the player to read and understand the storyline. These

dialogues help build the narrative and character interactions. Lastly, there's a **tutorial guide** that provides instructions to the player, such as learning the control keys, to help them get familiar with the gameplay mechanics.

Figure 32: Scene Animation and Dialogue with NPC

3.9 Challenges and Interactable Objects

In our game, we've kept the **challenges and interactable objects** straightforward to ensure it remains fun and engaging without being overly difficult. Players will need to **unlock doors** to progress through levels, with each door requiring one of two objectives: either **fighting**

spawning enemies or **finding collectible items** like gems, diamonds, jade, or red stones based on a quest. Some doors simply open when interacted with, while others **close behind the player** once passed, preventing backtracking. There are also **mission doors** that unlock after completing one of two different challenges and **teleportation doors** that transport players between areas. Players will encounter **normal platforms** for travel, and **falling platforms** that collapse when touched. **Fall damage** is a factor, as falling from a cliff will result in the player dying and respawning at the last checkpoint. Finally, the game includes various **spikes and traps** to add some risk to exploration.

1. **Unlock Doors**: To progress through levels, players need to unlock doors, which act as gateways between different areas.
2. **Two Objectives per Door**: Each door comes with two objectives—either fighting enemies or finding specific collectible items. The player must complete one of these objectives to unlock the door.
3. **Fight Spawning Enemies**: Certain doors unlock only after the player defeats waves of spawning enemies, making combat a key challenge.
4. **Find Collectible Items**: Players can also unlock doors by collecting items such as gems, diamonds, jade, or red stones, depending on the quest.
5. **Open Door**: Some doors simply open when the player interacts with them, allowing easy access to the next area.

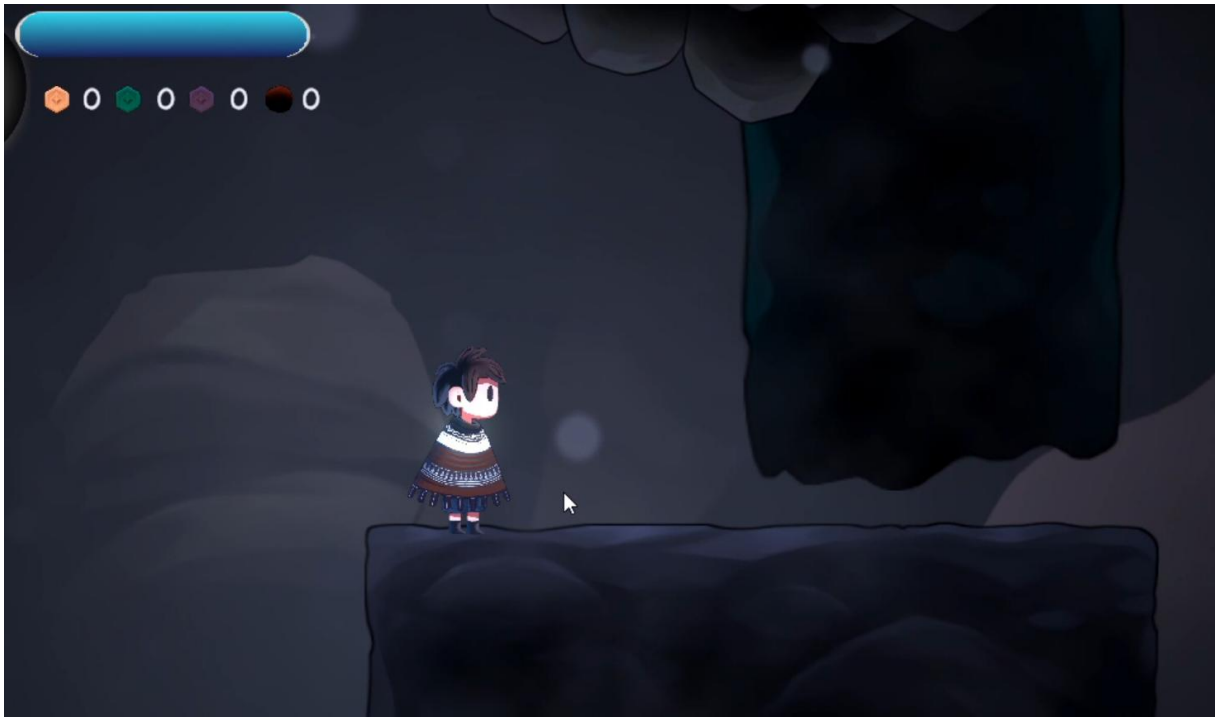


Figure 33: Interaction with Open Door

Figure 34: Door Open as Player Walks

1. **Close Door**: Once the player passes through these doors, they automatically close behind them, and the player cannot go back to the previous area.

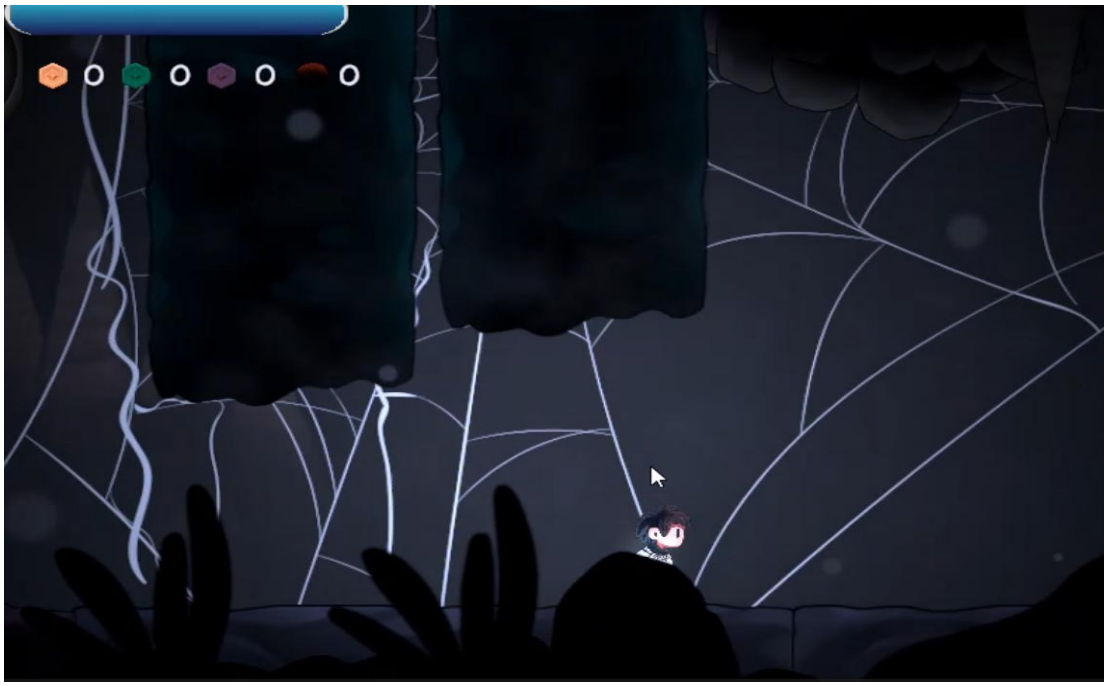


Figure 34: Door is initially opened

Figure 35: Door closed when player passed through

2. **Mission Door**: These doors require the player to complete one of two different challenges to unlock, offering flexibility in gameplay.

Figure 36: Mission Type 1

Figure 37: Mission Type 2

3. **Teleportation Door**: These doors teleport the player between two locations, allowing fast travel within a level.

Figure 38: Teleport Doors

4. **Normal Platform**: Standard platforms that the player can walk or jump across to move through the environment.

Figure 39: Normal Platforms

5. **Falling Platform**: These platforms collapse when the player steps on them, forcing quick movement to avoid falling.

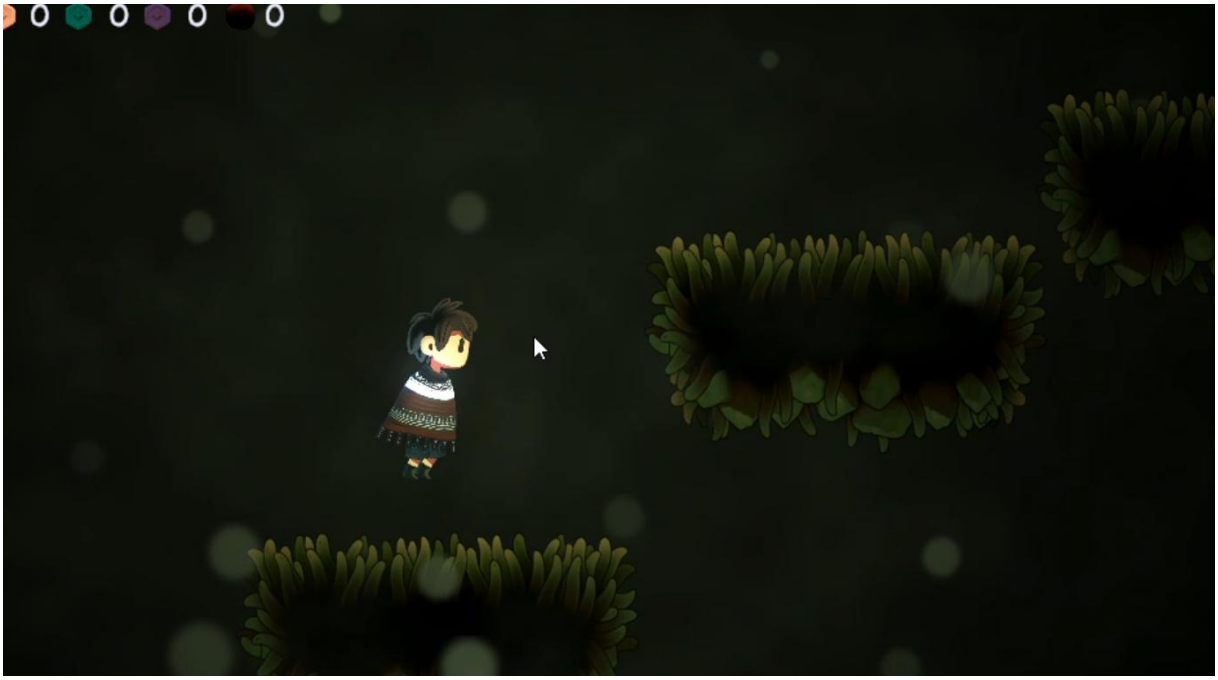


Figure 40: Fall Platforms

6. **Fall Damage**: If the player falls off a cliff and lands in certain areas, they will die and respawn at the last checkpoint.

Figure 41: Falling to the fall damage area

7. **Spikes and Traps**: The game includes spikes and other traps that can harm or kill the player if not avoided, adding an extra layer of danger to the levels.

Figure 42: Spikes on right side

8. **Jump Pads**: Jump pads allows player to hop on it and it will send player to very high places.

Figure 43: Jump Pads

Each of these elements contributes to a balanced mix of exploration, combat, and puzzle-solving that keeps the gameplay engaging without being overly complex.

3.10 Sounds and Graphic Design

We don’t have anything to show details about sound design as it is not in our scope. We simply have 3 theme songs for each levels and background effect of the animals in the forest which resources from royalty-free websites and You Tube. As for game design, we have explored what our graphics of the game look like.

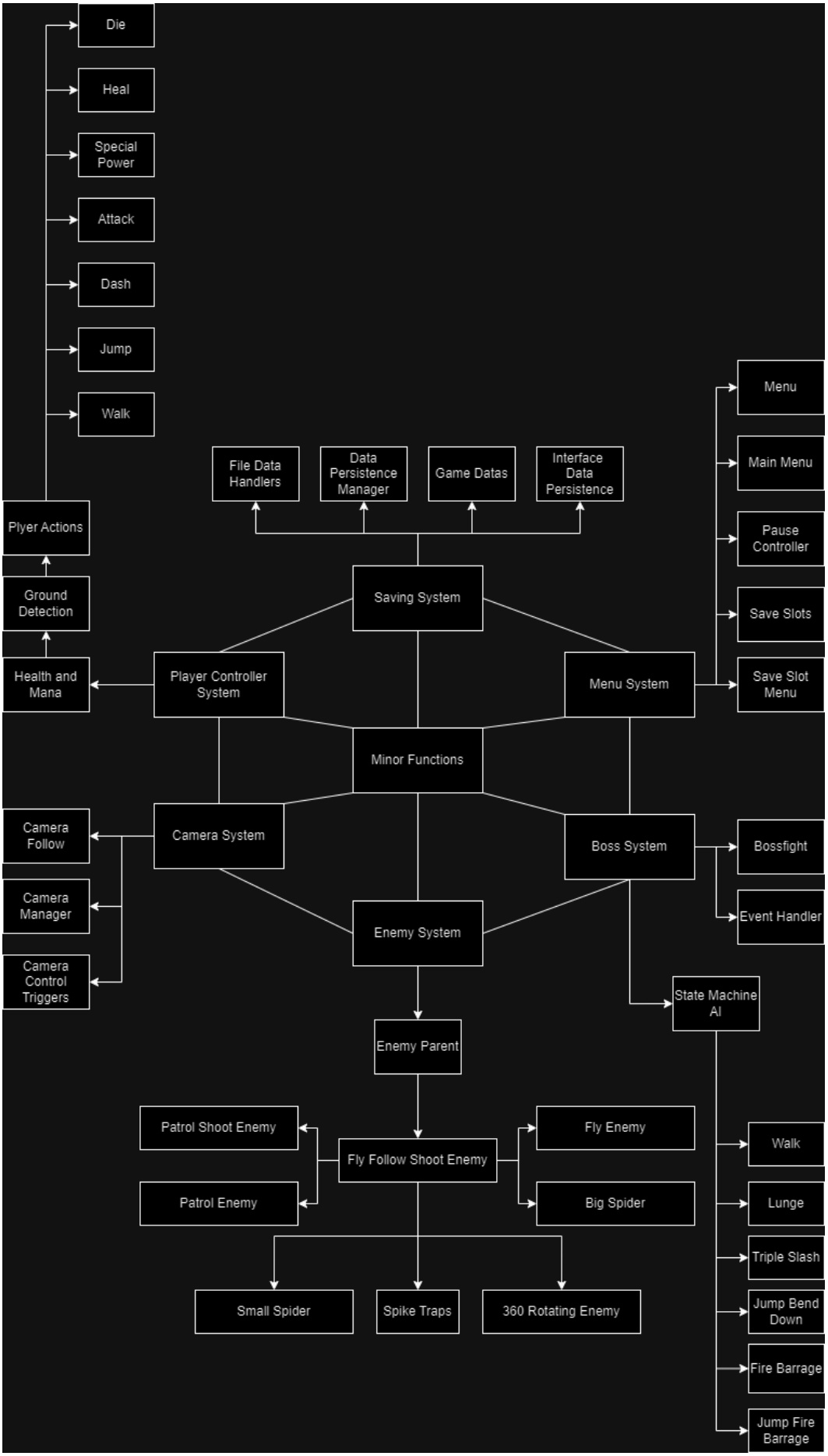
The sound effects we have are as below.

1. Level 1, 2, 3, 4 Themes
2. Background Sound effects
3. Player actions sounds jump, run, dash, attack, etc.
4. Doors open close effects

Chapter 4: Functionalities of Game (Logic, Code)

4.1 Major Functions

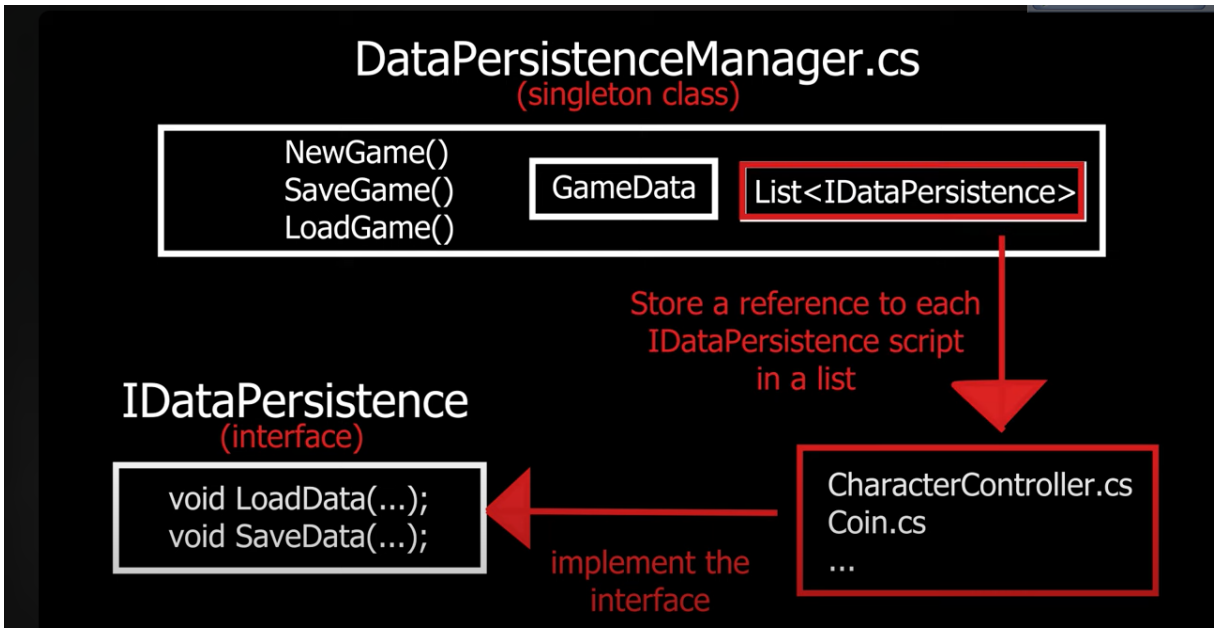
1. Saving System
2. Main Menu and Pause Menu System
3. Player Controller System
4. Camera System
5. Enemy System
6. Boss AI System


Die
Heal
Special Power
Attack
Dash
Jump
Walk
Plyer Actions
Ground Detection
Health and Mana
File Data Handlers
Data Persistence Manager
Game Datas
Interface Data Persistence
Saving System
Player Controller System
Menu System
Minor Functions
Menu
Main Menu
Pause Controller
Save Slots
Save Slot Menu
Camera Follow
Camera Manager
Camera Control Triggers
Camera System
Boss System
Bossfight
Event Handler
Enemy System
State Machine AI
Enemy Parent
Patrol Shoot Enemy
Patrol Enemy
Fly Follow Shoot Enemy
Fly Enemy
Big Spider
Small Spider
Spike Traps
360 Rotating Enemy
Walk
Lunge
Triple Slash
Jump Bend Down
Fire Barrage
Jump Fire Barrage

4.2 Saving System

The **core concept of a save system** in games is to store the player's current progress, allowing them to resume from that exact point at a later time. It enables players to quit and restart the game without losing important milestones, configurations, or progress.

1. Core Concepts



We need a manager to make data persistent across the scenes, we will call it data persistence manager. It is the most important part of the entire saving system and will mange new data, load data, and save data and saving slots.

DataPersistenceMananger

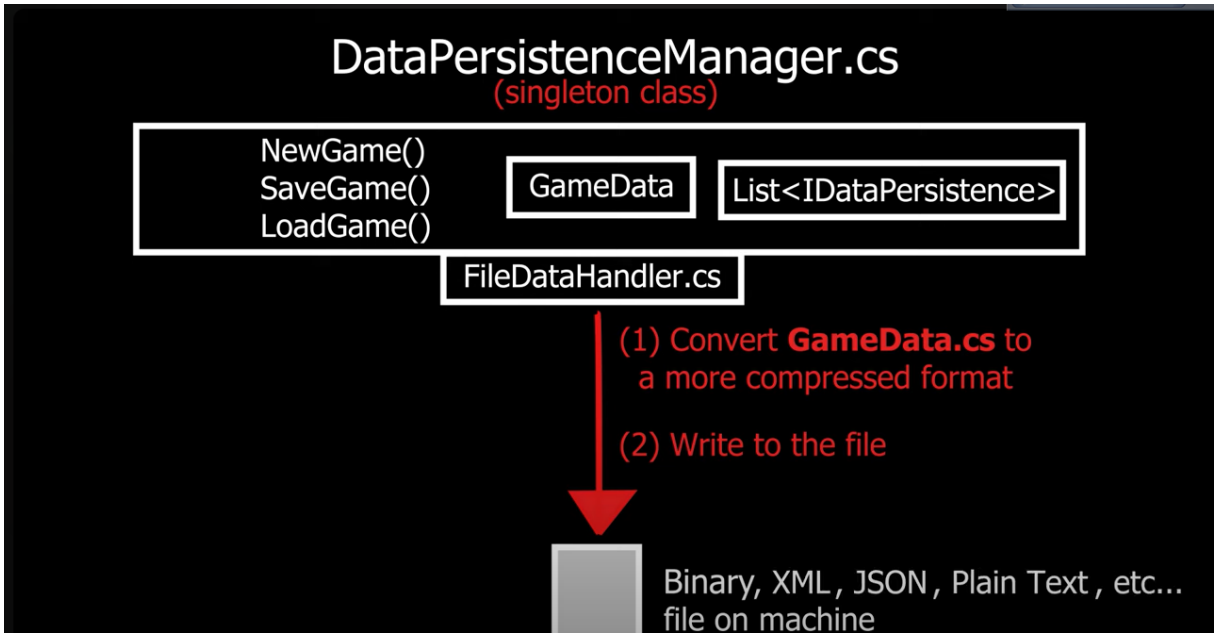
We need to specify what data types we are storing.

Game Data

We need to accept to the scripts of related objects we have to save. For example, if we want to save the player’s position, we need to access to player controller script with Interface script.

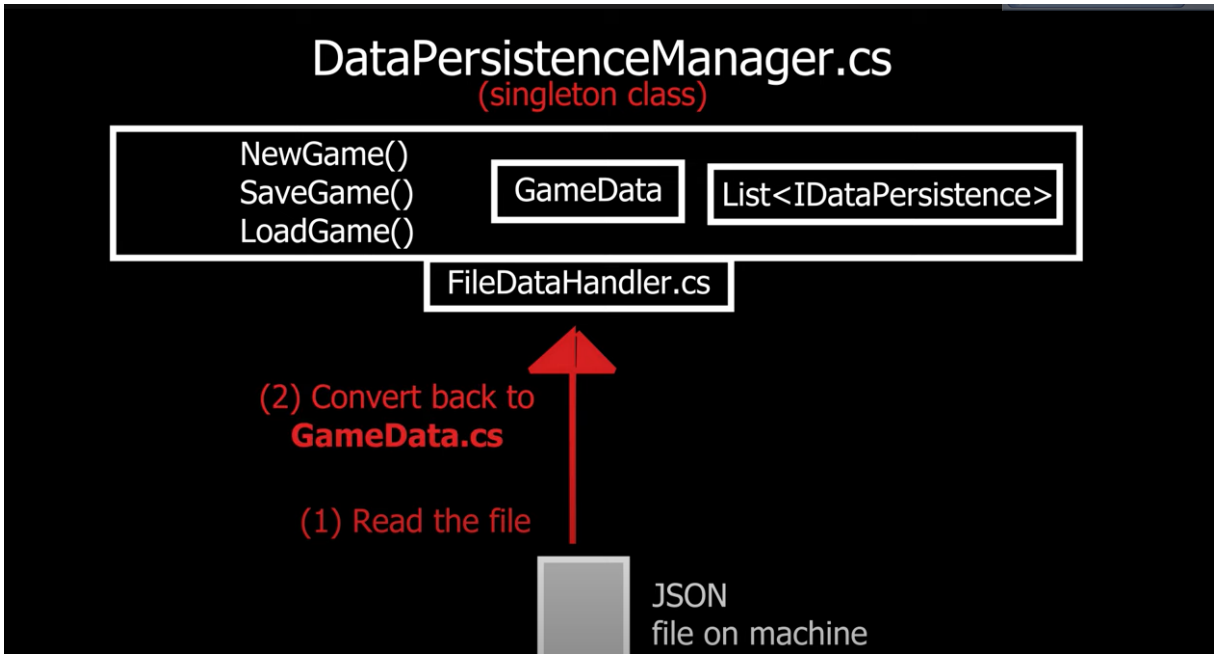
I DataPersistence.

In the figure above, the data persistence manager acts as a singleton class and it has access to many of the scripts that has data persistence interface and add two methods in them, load data and save data.

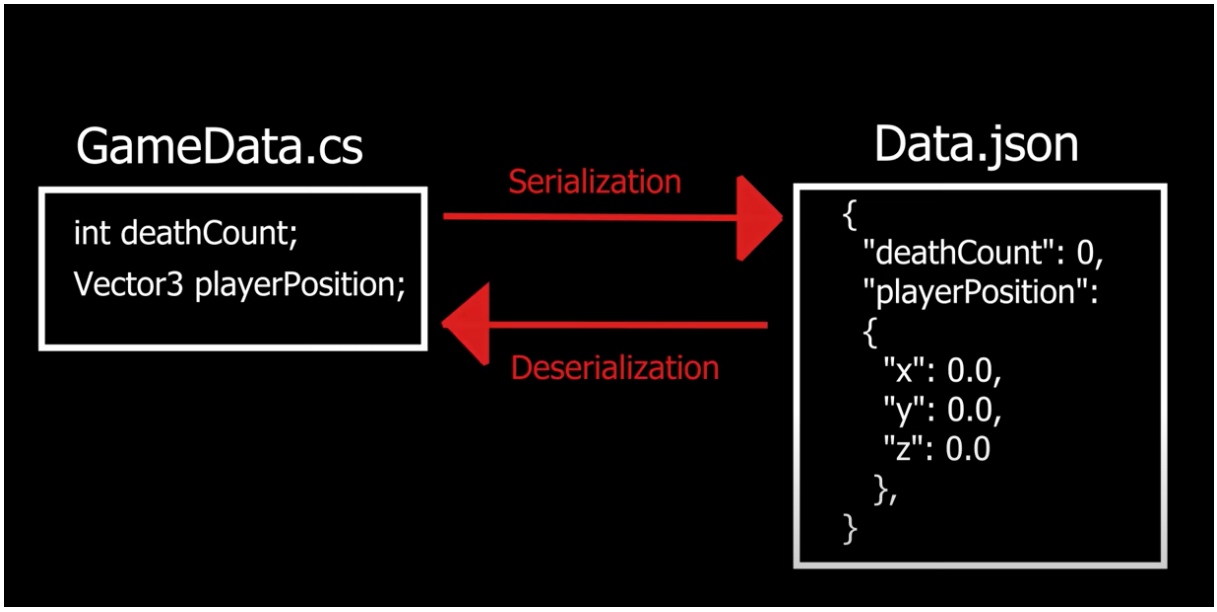


We need to actually save those save somewhere. We can save them in our local device using I/O operation function **File Data Handler**.

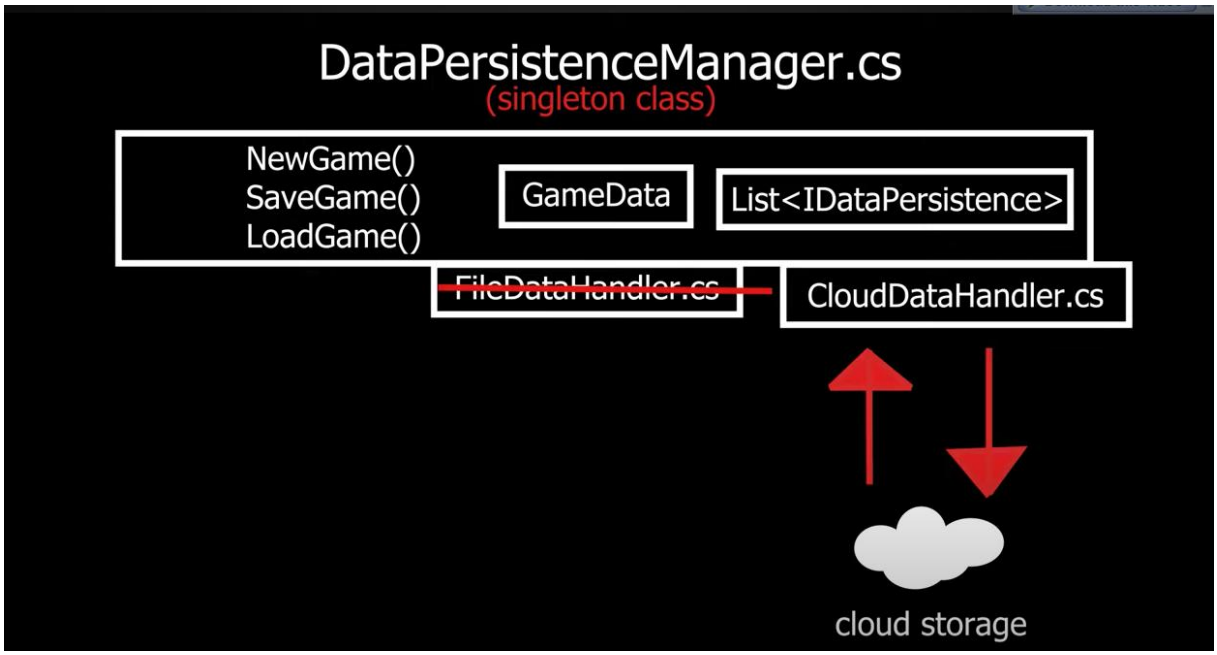
In Figure above, the game data we initialized int the beginning will now have data saved , and the file data handler will convert the game data into a compressed format JSON and store in a destination . This process is typically for saving new data.



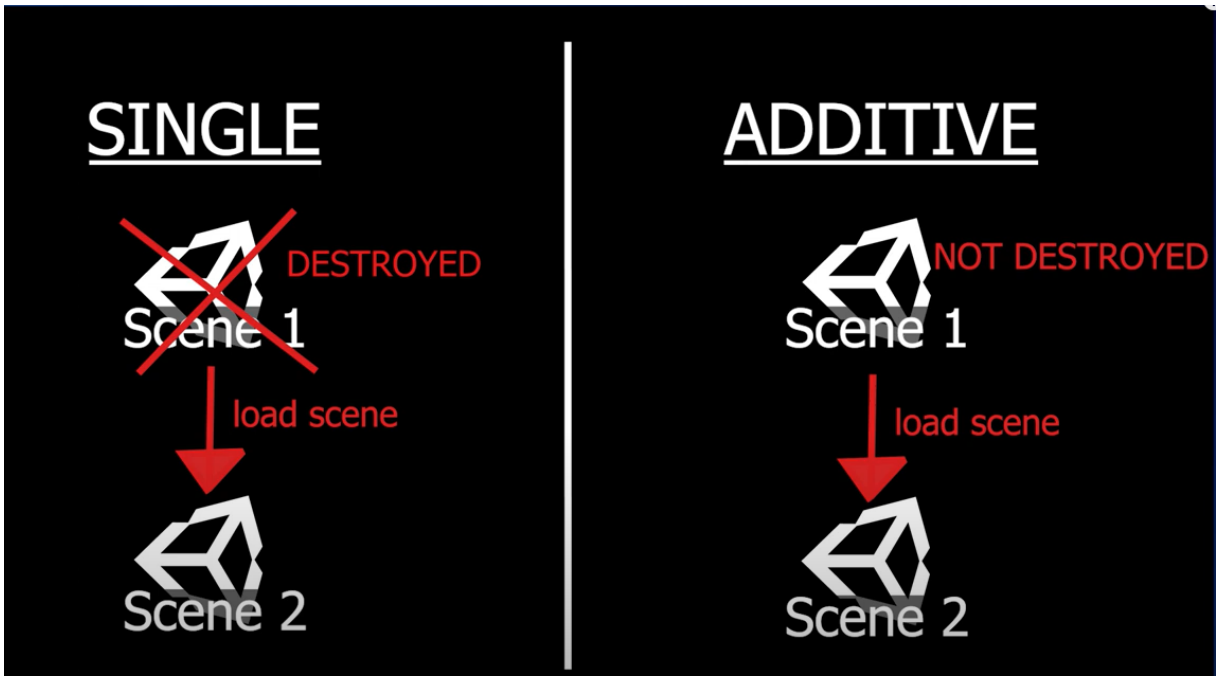
When Load Game is needed, those JSON files on local machine will come to file data handler. The handler will covert JSON back to game data and feed into the manager to process the request of the game. This process is typically for loading game.



Transferring from game data to json is called **Serialization** and otherwise, **Deserialization**. Storing as JSON file make easier for developers to test successfulness of the saving system and it is a must to accept data from local machine to Unity as C Data.



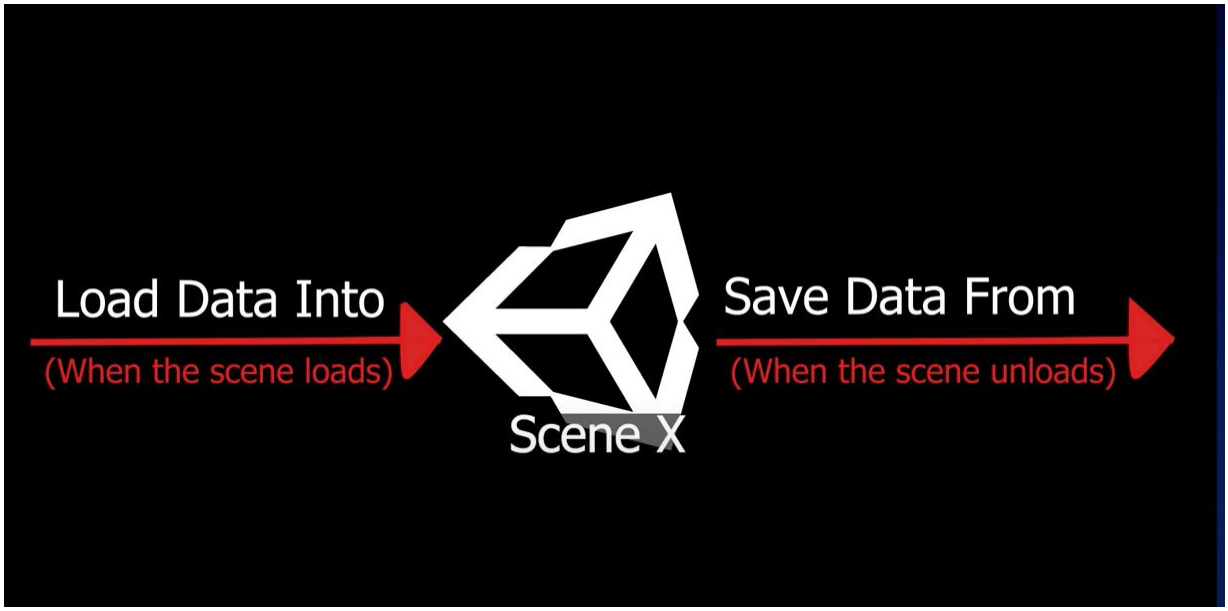
This Save system architecture is very easy to expand and replace with different storing options. We can easily change from local saving to cloud saving and the rest of the system remains the same.



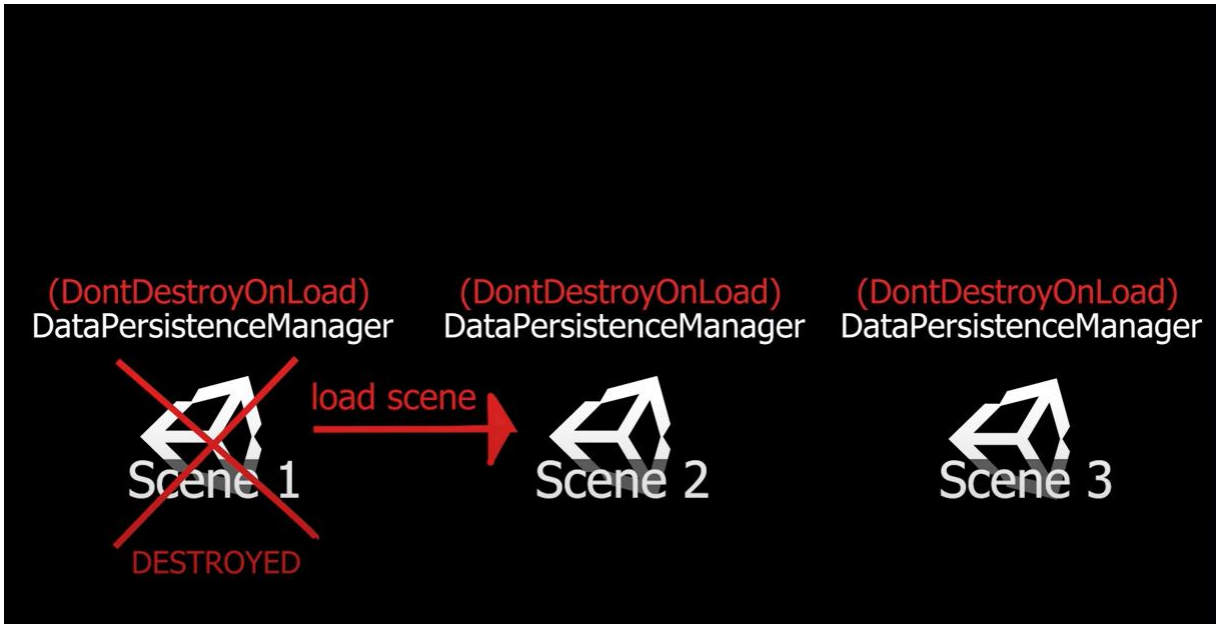
We also need to consider with our Scene Management and how to transfer from each scene to scene, from main menu to levels. In Unity, there is two ways of handling scenes,

1. Single: load one scene and destroy previous one.
2. Additive: load one scene but previous one not destroyed.

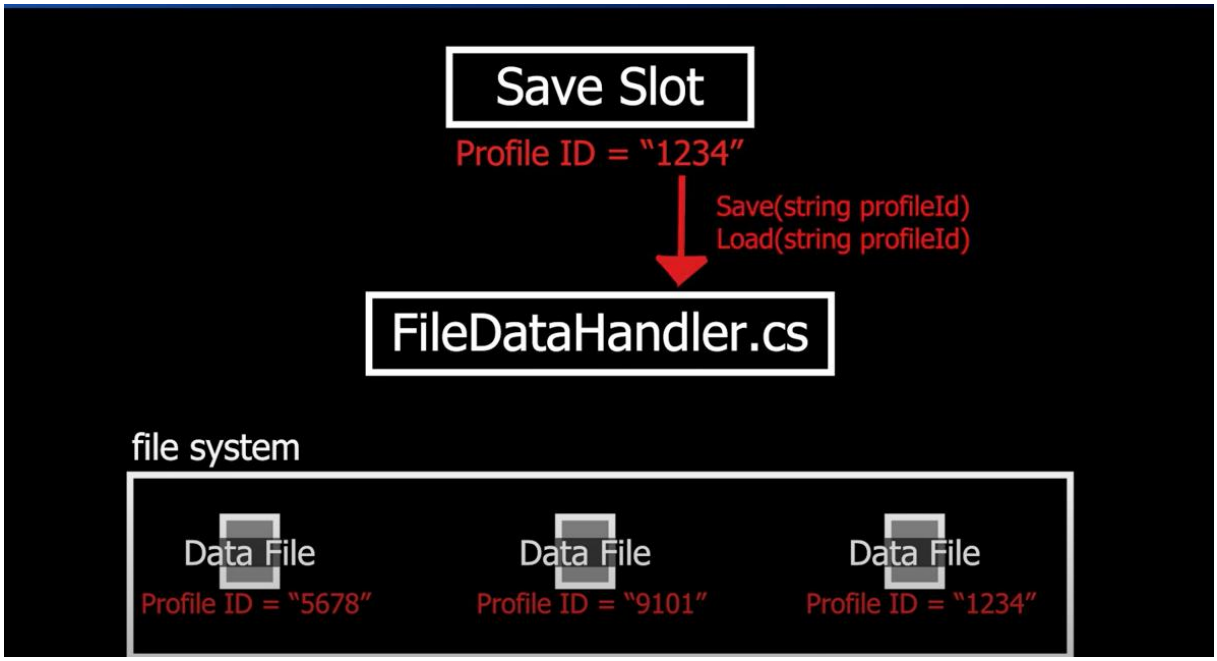
We chose first method for optimization and simplicity of the project and we have a problem with our current saving system approach.



As we explained in above, our system wants to load data when we load a scene, chapter1 or chapter 2, etc. We want to save those data according to check points when we move to another scene. But since whenever we move to next scene, the previous scene is destroyed, so the data saving manager is all destroyed and the data is not saved.

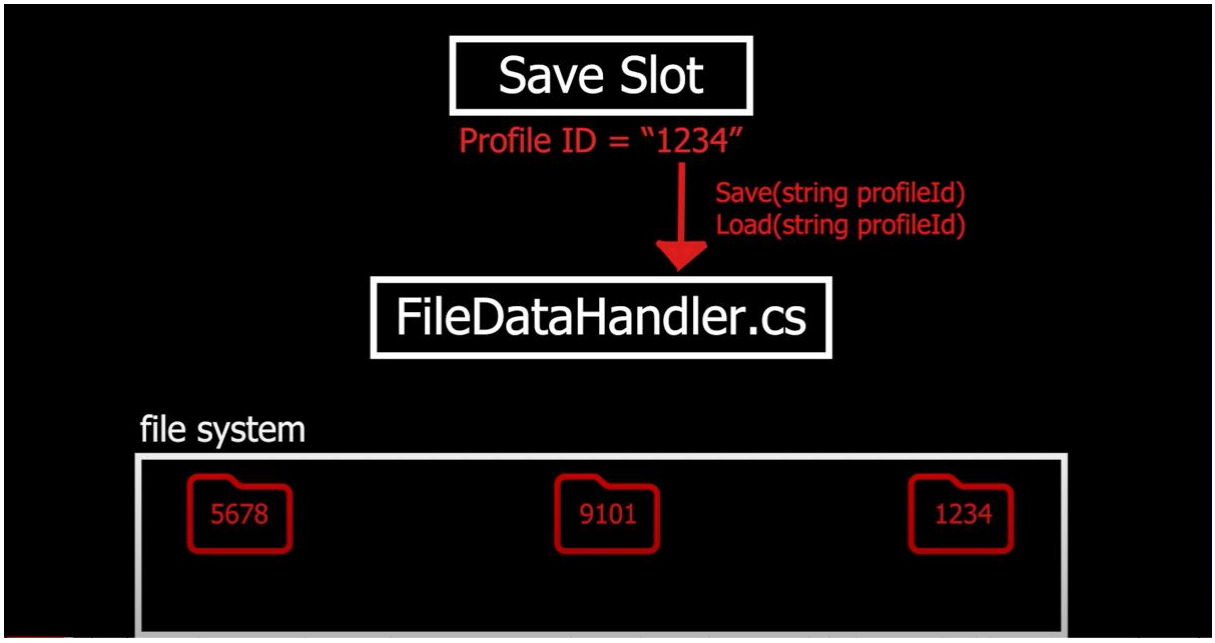


To solve this problem, we use “Don’t Destroy on Load” which carry the data persistence manager to other scenes. As shown in Figure above, when Scene 1 is destroyed, the data persistence instance is carried to next scene. That way it won’t be destroyed, and it will complete its job of saving and loading data.

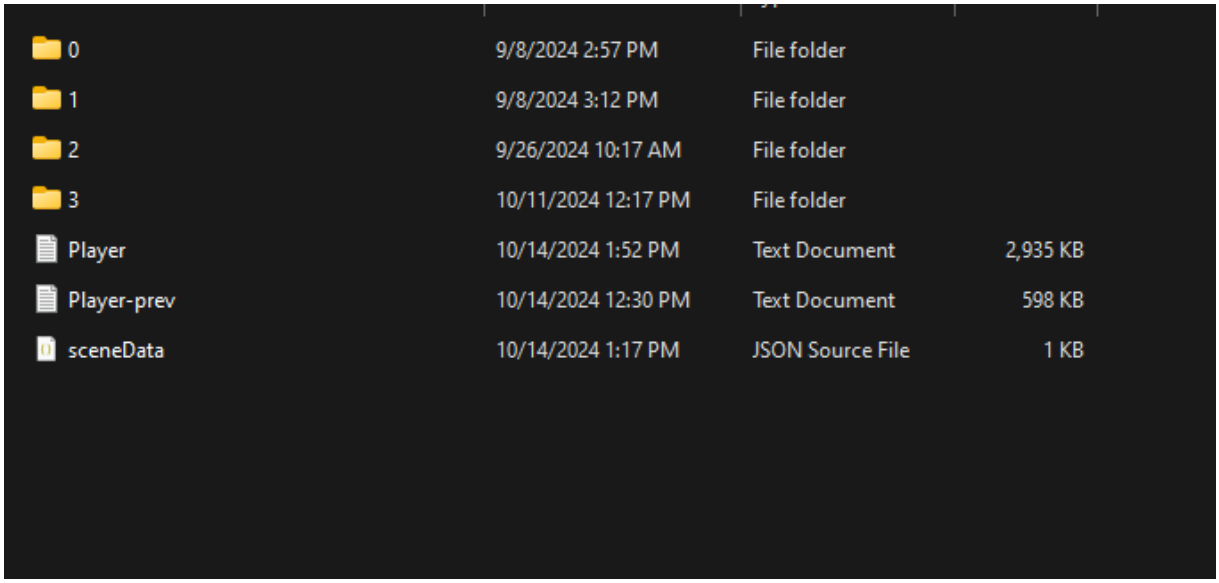


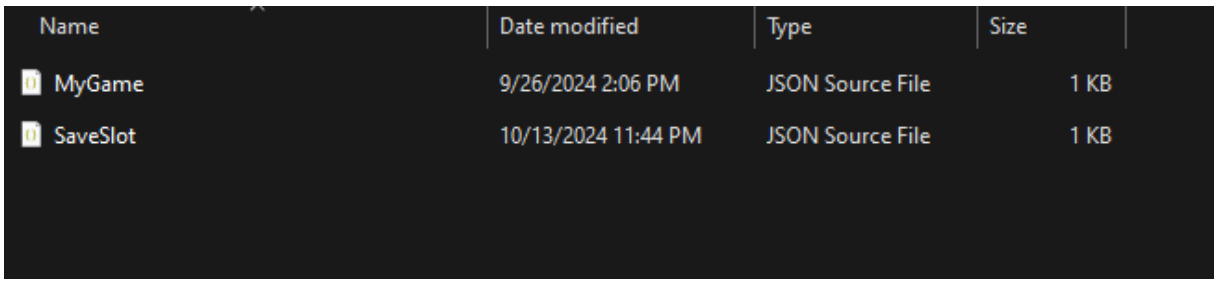
We want to give our players multiple save slots to save different progress. We will add Save slots entity that communicates with our file handler and add following things.

1. Folder for each save slot file
2. Profile ID for unique identifier
3. Time stamp to know when it was saved



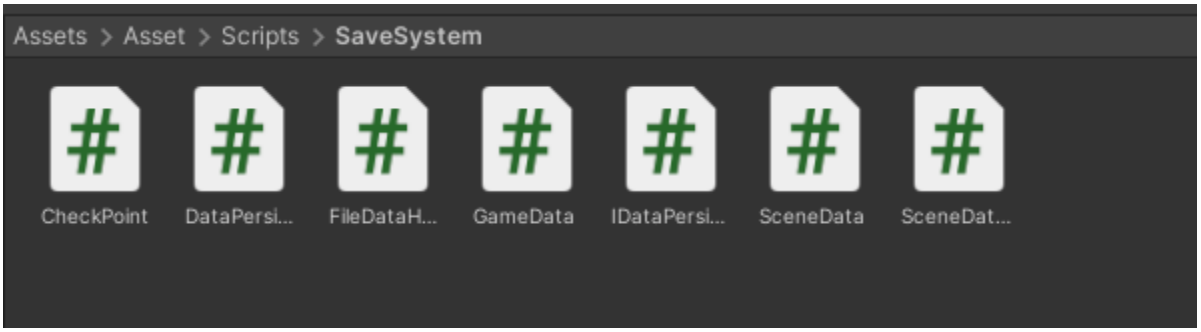
With this approach, we can actually check in the directory where the data exists and see the structure like the below figures.





- My game is Game Data file.
- Save Slot is the saved data of scenes related to each slot so the game will know which scene to load when player comes back to play.

Above Figure show how a saved json data should look. We have data of player's X and Y positions. And Player has finished 4 checkpoints of the chapter. We also store Boolean "should reset" to determine when to reset the chapter. The rest attributes are not used in this project. Let's go through the code functions of the system in next pages.



2. Game Data Class

The `Game Data` class is the core structure that holds all the important information about the player's progress, such as their position, health, mana, and more. When the game is saved, all this data gets stored, and when the game is loaded, it's restored from the saved file.

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

[System.Serializable]
24 references
public class GameData
{
    public Vector2 playerPosition;
    public long lastUpdated; //timestamp storage
    public bool shouldReset = false;

    public List<int> checkpointIDs = new List<int>();//
    ///public float health;
    //public float mana = 0; // Default mana value

    //public List<int> completedTutorialIDs = new List<

    //public bool tutorialStarted = false; // Add this



    2 references
    public GameData()
    {
        playerPosition = Vector2.zero; // Default playe
        //health = 0f;

    }
}
```

- **player Position**: This variable stores the exact location of the player in the game as a **Vector2**, which means it holds two numbers representing the player's position on the map.
- **last Updated**: This variable holds a **timestamp** of when the game was last saved. It helps track when the latest save happened.
- **should Reset**: This is a flag to check if the game should be reset or not, which will be used for restarting chapter or checkpoints.
- **Checkpoint IDs**: A **list** that stores the IDs of the checkpoints that the player has activated throughout the game. This ensures that when the game is loaded, the player's progress through different checkpoints is retained.

The **constructor** (`GameData()`) initializes default values, like setting the player's starting position to (0,0) and health to 0, ensuring that there's a baseline state when a new game begins.

3. IDataPersistence Interface

The `IDataPersistence` interface is a **contract** for any class that wants to handle saving or loading data. It has two methods that classes must implement:

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
        8 references
5   public interface IDataPersistence
6   {
            2 references
7       void LoadData(GameData data);
8
            2 references
9       void SaveData(ref GameData data);
10
11  }
12
```

```
//------------------------------------------------------------------------
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System; // Add this line
using UnityEngine.EventSystems;
using UnityEngine.SceneManagement; // Add this line for scene management
using UnityEngine.InputSystem;
//------------------------------------------------------------------------

Unity Script (2 asset references) | 54 references
public class PlayerController : MonoBehaviour, IDataPersistence
{
```

```
//-----------------------Save&Load-----------------------------------------------------
//loading player data
2 references
public void LoadData(GameData data)
{
    this.transform.position = data.playerPosition;

}

//saving player data
2 references
public void SaveData(ref GameData data)
{
    data.playerPosition = this.transform.position;

}
//-----------------------Save&Load-----------------------------------------------------
```

- **LoadData(GameData data)**: This method is responsible for loading the data stored in the `GameData` object when the game starts or is resumed from a saved file.
- **SaveData(ref GameData data)**: This method takes the current game state, modifies the `GameData` object to reflect that state, and saves it. The `ref` keyword means that changes made to the `GameData` object inside this method will affect the original object passed in.
- **Used in Other Scripts**: The above figures show how `IDataPersistence acts as interface and uses its method in player script saving the player positions.`

This system allows any class that interacts with saving or loading game data to implement these methods and integrate seamlessly with the rest of the save system and **other scripts**.

4. Data Persistence Manager

The **DataPersistenceManager** is the core script that handles saving and loading game data. Here's a step-by-step breakdown of how the system works

Singleton Pattern (Initialization)

The first task of the script is to ensure that only one instance of the **DataPersistenceManager** exists throughout the game.

```
public static DataPersistenceManager instance { get; private set; }

private void Awake()
{
    if (instance != null)
    {
        Debug.LogError("Found more than one Data Peristence Manager in the scene.");
        Destroy(this.gameObject);
        return;
    }
    instance = this;
    DontDestroyOnLoad(this.gameObject);
}
```

Explanation: This ensures that there's a single instance of the manager across all scenes. The `DontDestroyOnLoad()` method keeps the manager active across scene transitions, and if another instance exists, it gets destroyed. We explored this concept at the beginning of this chapter.

Loading and Managing Game Data

- **Loading game data**: The manager attempts to load existing game data or creates new data if no saved data is found.
- **GameData class**: Stores the player's progress, such as health, position, checkpoints, etc.

```
public void LoadGame()
{
    if (disableDataPersistence) return;

    gameData = dataHandler.Load(selectedProfileId);

    if (gameData == null)
    {
        Debug.Log("No saved data was found. Starting a new game.");
        if (initializeDataIfNull)
        {
            NewGame(); // Creates new game data if there's none
        }
    }
    else
    {
        foreach (IDataPersistence dataPersistenceObj in dataPersistenceObjects)
        {
            dataPersistenceObj.LoadData(gameData); // Loads data into relevant scripts
        }
    }
}
```

Explanation: This snippet shows how the system checks for saved data. If no data exists, a new game is started. Otherwise, it loads the game data into all objects that implement the **IDataPersistence** interface.

Saving Game Data

- **Saving game progress**: The `SaveGame()` method updates the current state of the game and writes it to the file.

```
public void SaveGame()
{
    if (disableDataPersistence) return;

    if (this.gameData == null)
    {
        Debug.LogWarning("No data was found. A new game needs to be started before data c
        return;
    }

    foreach (IDataPersistence dataPersistenceObj in dataPersistenceObjects)
    {
        dataPersistenceObj.SaveData(ref gameData); // Collects data from objects
    }

    dataHandler.Save(gameData, selectedProfileId); // Saves data to file
    gameData.lastUpdated = System.DateTime.Now.ToBinary(); // Time-stamp
}
```

Explanation: The system iterates through all game objects that implement **IDataPersistence** and updates their data. The final game state is then saved into a file using the **FileDataHandler**.

Handling Scenes and Checkpoints

The script also manages transitions between scenes and saves scene-related data separately using a **SceneDataHandler**. It tracks the last scene the player was in and ensures that upon loading, the player is returned to the correct location.

We will explain File and Scene Data Handlers in next pages.

```
public void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    sceneData.currentScene = scene.name;
    sceneDataHandler.SaveSceneData(sceneData);
    LoadGame(); // Loads game data when a new scene is loaded
}
```

Explanation: This code runs every time a new scene is loaded. It updates the current scene data and calls the `LoadGame()` function to ensure the game state is restored correctly.

New Game Logic

If the player starts a new game, the save data is reset, and default values are initialized. This ensures the game begins in a clean state.

```
public void NewGame()
{
    gameData = new GameData(); // Reset game data
    shouldReset = true; // Indicates a new game has started
}
```

Explanation: This method is called to start a fresh game, resetting game progress and initializing all necessary data.

Managing Profiles and Save Slots

The system supports multiple profiles, allowing different users to save their progress separately. The selected profile ID is used to load or save data specific to that user.

```
public void ChangeSelectedProfileId(string newProfileId)
{
    this.selectedProfileId = newProfileId;
    LoadGame(); // Loads game data for the new profile
}
```

Explanation: This allows the system to switch between different player profiles, ensuring each player's data is kept separate in different save slots.

5. File and Scene Data Handlers

The `FileDataHandler` class is responsible for saving and loading game data in Unity. It manages file operations, such as reading from and writing to JSON files, which store the game state or player profiles. This functionality is critical for persistent data, allowing players to resume progress in a game. Below is an explanation of its key methods.

Constructor

The constructor initializes the `FileDataHandler` object, setting the path where data files are stored and the name of the file.

```
public FileDataHandler(string dataDirPath, string dataFileName)
{
    this.dataDirPath = dataDirPath;
    this.dataFileName = dataFileName;
}
```

- **`dataDirPath`**: Directory where files are stored.
- **`dataFileName`**: Name of the file (e.g., `GameData.json`).

Load Method

This method is responsible for loading game data from a file. It reads a file, deserializes the JSON into a `GameData` object, and returns the object for use in the game.

```
public GameData Load(string profileId)
{
    if (profileId == null)
    {
        return null;
    }

    string fullPath = Path.Combine(dataDirPath, profileId, dataFileName);
    GameData loadedData = null;

    if (File.Exists(fullPath))
    {
        try
        {
            using (FileStream stream = new FileStream(fullPath, FileMode.Open))
            {
                using (StreamReader reader = new StreamReader(stream))
                {
                    string dataToLoad = reader.ReadToEnd();
                    loadedData = JsonUtility.FromJson<GameData>(dataToLoad);
                }
            }
        }
        catch (Exception e)
        {
            Debug.LogError("Error loading data: " + fullPath + "\n" + e);
        }
    }
    return loadedData;
}
```

Process:

1. Checks if the profile ID is valid.
2. Combines the directory path and profile ID to locate the file.
3. If the file exists, reads its contents and deserializes the JSON back into a `GameData` object.
4. Handles errors gracefully, logging them with `Debug.LogError`.

Save Method

This method saves game data by serializing a `GameData` object to JSON and writing it to a file.

```
public void Save(GameData data, string profileId)
{
    if (profileId == null)
    {
        return;
    }

    string fullPath = Path.Combine(dataDirPath, profileId, dataFileName);

    try
    {
        Directory.CreateDirectory(Path.GetDirectoryName(fullPath));
        string dataToStore = JsonUtility.ToJson(data, true);

        using (FileStream stream = new FileStream(fullPath, FileMode.Create))
        {
            using (StreamWriter writer = new StreamWriter(stream))
            {
                writer.Write(dataToStore);
            }
        }
    }
    catch (Exception e)
    {
        Debug.LogError("Error saving data: " + fullPath + "\n" + e);
    }
}
```

Process:

1. Verifies the profile ID before saving.
2. Creates the directory if it doesn't exist.
3. Serializes the `GameData` object to JSON format using `JsonUtility`.
4. Writes the JSON data to the file.
5. Catches and logs any errors.

LoadAllProfiles Method

This method retrieves all profiles by iterating over directories and loading the associated `GameData` for each profile.

```
public Dictionary<string, GameData> LoadAllProfiles()
{
    Dictionary<string, GameData> profileDictionary = new Dictionary<string, GameData>();

    IEnumerable<DirectoryInfo> dirInfos = new DirectoryInfo(dataDirPath).EnumerateDirector
    foreach (DirectoryInfo dirInfo in dirInfos)
    {
        string profileId = dirInfo.Name;
        string fullPath = Path.Combine(dataDirPath, profileId, dataFileName);

        if (!File.Exists(fullPath))
        {
            Debug.LogWarning("Skipping directory: " + profileId);
            continue;
        }

        GameData profileData = Load(profileId);
        if (profileData != null)
        {
            profileDictionary.Add(profileId, profileData);
        }
        else
        {
            Debug.LogError("Error loading profile: " + profileId);
        }
    }
    return profileDictionary;
}
```

Process:

1. Loops through all directories in the data folder.
2. For each directory, checks if the data file exists and loads the profile data.
3. Adds successfully loaded profiles to a dictionary.

GetMostRecentlyUpdatedProfile Method

This method identifies the most recently updated profile by comparing the `lastUpdated` field in each `GameData` object.

```
1 reference
public string GetMostRecentlyUpdatedProfile()
{
    string mostRecentprofileId = null;

    Dictionary<string, GameData> profilesGameData = LoadAllProfiles();
    foreach (KeyValuePair<string, GameData> pair in profilesGameData)
    {
        string profileId = pair.Key;
        GameData gameData = pair.Value;

        //defensive check
        //skip this entry if the gamedata is null
        if (gameData == null)
        {
            continue;
        }

        //if this is the first data we've come across that exists, its the most recent so far
        if (mostRecentprofileId == null)
        {
            mostRecentprofileId = profileId;
        }
        //otherwise, compare to see which date is the most recent
        else
        {
            DateTime mostRecentDataTime = DateTime.FromBinary(profilesGameData[mostRecentprofileId].lastUpdated);
            DateTime newDateTime = DateTime.FromBinary(gameData.lastUpdated);
            //the greatest DateTime value is the most recent
            if (newDateTime > mostRecentDataTime)
            {
                mostRecentprofileId = profileId;
            }
        }

    }
    return mostRecentprofileId;
}
```

Process:

1. Iterates over loaded profiles.
2. Compares the `lastUpdated` field to find the most recent profile.
3. Returns the ID of the most recent profile.

The `FileDataHandler` class core functions include saving/loading serialized data in JSON format and managing multiple player profiles. **The** Scene`DataHandler` **works the same as file data handlers but it deals with chapter scenes instead of player data which is Scene Data script.**

6. Check Point

The purpose of a checkpoint is to save the player's progress at specific points in the game, so if they lose or restart, they can resume from the latest checkpoint instead of starting over. Here's a breakdown of the key components of the script:

Key Components:

1. **Checkpoint ID:**
    - Each checkpoint has a unique `checkpointID` which helps identify the checkpoint.
    - This ID is used to determine whether the player has already triggered that checkpoint and saved their progress at that location.

2. **OnTriggerEnter2D:**
    - This method detects when the player (an object with the tag "Player") enters the checkpoint's trigger area.
    - It checks if the player has already activated the current checkpoint by looking at the list of checkpoint IDs stored in `GameData`.

3. **Checkpoint List:**
    - A list of checkpoint IDs (`checkpointIDs`) is managed by `DataPersistenceManager` to track which checkpoints have been triggered. This prevents re-saving the game at the same checkpoint multiple times.
    - If the checkpoint's ID isn't in the list, it gets added, and the game is saved.

4. **Game Saving:**
    - When the player activates a new checkpoint (i.e., when the checkpoint's ID isn't already in the list), the game saves progress via `DataPersistenceManager.instance.SaveGame()`. This stores the player's state at that point in the game.

5. **Audio and Visual Feedback:**
    - A checkpoint sound (`checkpointSound`) plays to provide auditory feedback when the player activates the checkpoint.
    - A message (`checkpointSavedText`) is displayed to inform the player that their progress has been saved. The message appears for 3 seconds and is controlled using a coroutine (`ShowCheckpointSavedMessage()`).

```
public int checkpointID; // Unique identifier for the checkpoint

Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Player"))
    {
        Debug.Log("Checkpoint with ID " + checkpointID + " triggered.");

        var checkpointIDs = DataPersistenceManager.instance.GameData.checkpointIDs;

        Debug.Log("Checking from the list: " + string.Join(", ", checkpointIDs));

        if (!checkpointIDs.Contains(checkpointID))
        {
            Debug.Log("No current checkpoint in the list. Adding checkpoint ID " + checkpointID);
            checkpointIDs.Add(checkpointID);



            DataPersistenceManager.instance.SaveGame();
            Debug.Log("Game Saved at Checkpoint" + checkpointID);
            // Now show the checkpoint saved message
            StartCoroutine(ShowCheckpointSavedMessage());
            checkpointSound.Play();
        }
        else
        {
            Debug.Log("Current checkpoint already exists in the list. Not saving.");
        }
    }
}


1 reference
private IEnumerator ShowCheckpointSavedMessage()
{
    checkpointSavedText.gameObject.SetActive(true);
    yield return new WaitForSeconds(3); // Display the message for 3 seconds
    checkpointSavedText.gameObject.SetActive(false);
}
```

7. Summary:

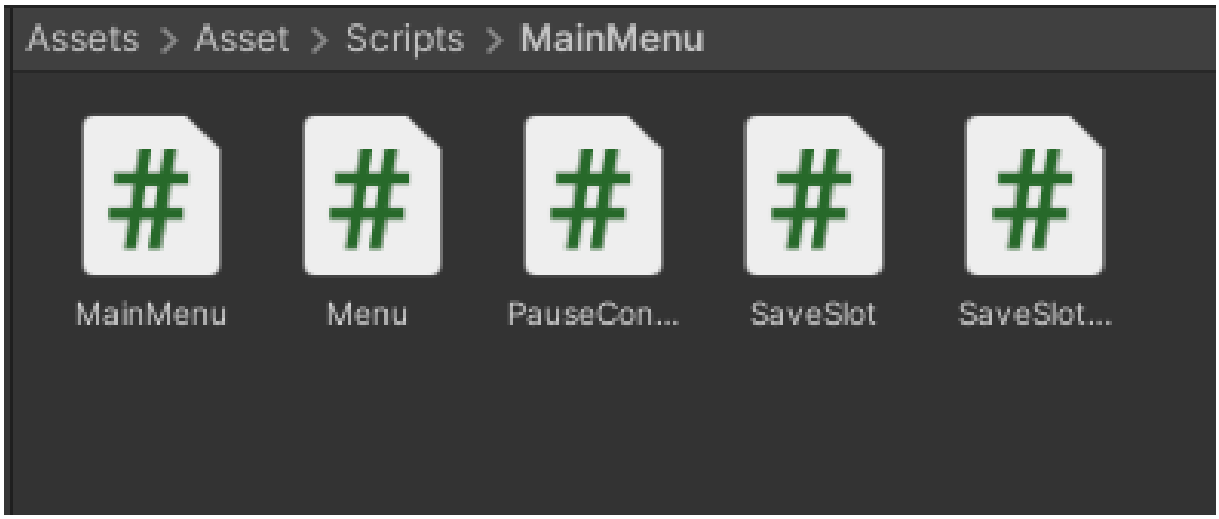
- When the player touches a checkpoint, the system checks whether that checkpoint has been activated before (using its ID).
- If not, it saves the game, plays a sound, and displays a "checkpoint saved" message.
- If the checkpoint has already been triggered, it skips saving to avoid redundant operations.

4.3 Main Menu and Pause Menu System

1. Core Concepts

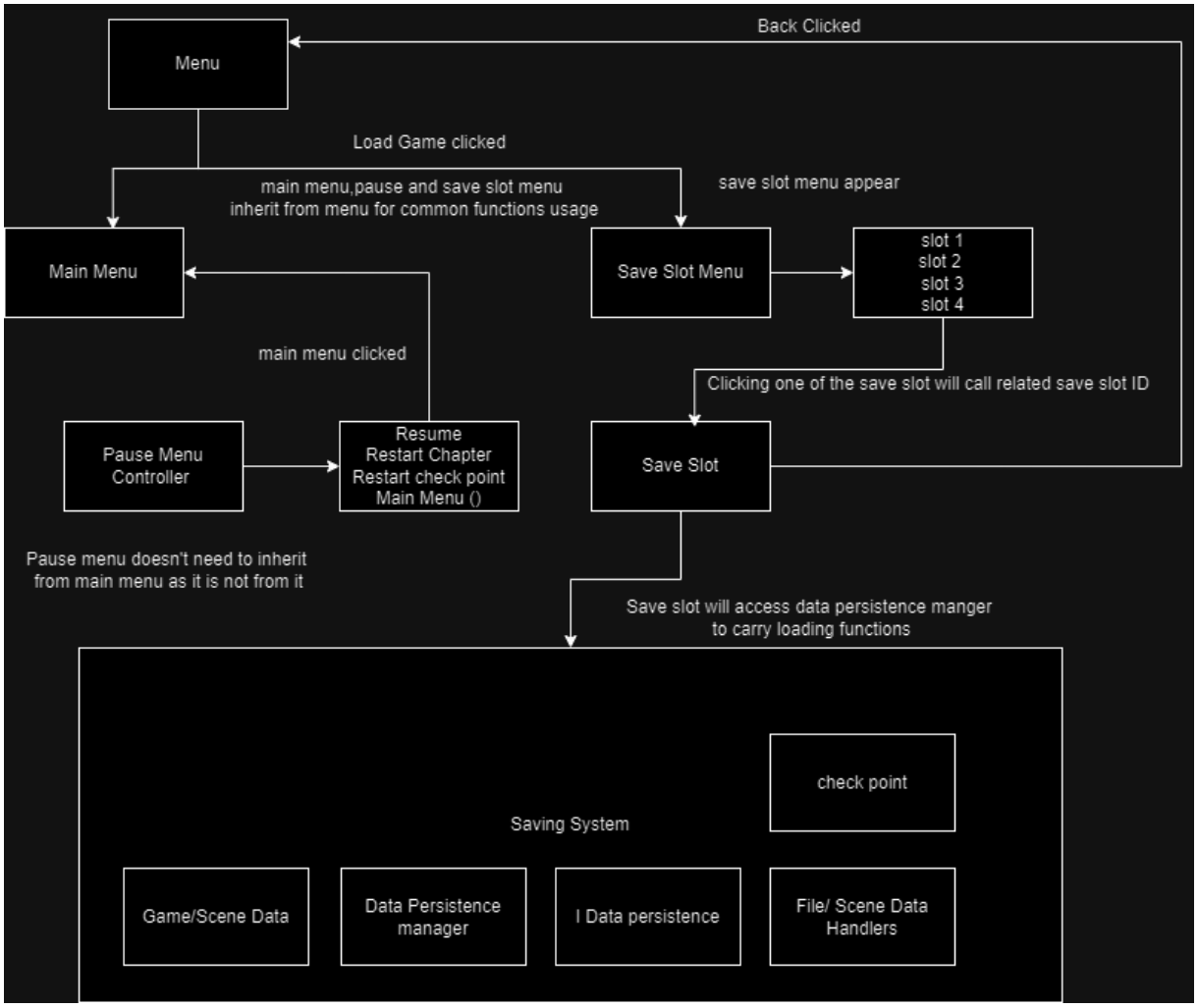
We want to give our players multiple save slots to save different progress. We will add Save slots entity that communicates with our file handler and Main Menu.

These are the Functions in the system.



1. Menu
2. Main Menu
3. Save Slot
4. Save Slot Menu
5. Pause Controller

The game begins with the **Main Menu**, where players can select options like "Load Game," which activates the **Save Slots Menu**. This menu displays available **Save Slots** for user selection, allowing players to click on a slot to load a saved game, which interacts with the **Data Persistence Manager** to retrieve the corresponding game data and transition to the relevant scene. If a player clicks the back button in the **Save Slots Menu**, it deactivates itself and returns control to the **Main Menu**. Additionally, the **Pause Controller** can activate a pause menu at any time during gameplay, enabling users to pause the game and choose to either resume or return to the **Main Menu** without affecting the functionality of the **Save Slots Menu**. Consider the Figure below for clearer picture.



2. Main Menu

This **Main Menu** script is responsible the first thing user will see when opening the game . It provides functionality for navigating between different menu options, starting a new game, continuing an existing game, loading a saved game, and quitting the application.

Menu Navigation and Initialization: The script ensures that certain buttons (like *Continue* and *Load Game*) are interactable based on whether there is existing game data. If no game data exists, these buttons are disabled to prevent loading or continuing a non-existent game.

```
public void Start()
{
    // Check if game has data, if no data, disable continue and load buttons
    if (!DataPersistenceManager.instance.HasGameData())
    {
        continueGameButton.interactable = false;
        loadGameButton.interactable = false;
    }
}
```

Start a New Game: When the player clicks the "New Game" button, it opens the save slot menu (to select a save slot) and deactivates the main menu. This prepares the system for starting a new game.

- `saveSlotsMenu.ActivateMenu(false);`: Activates the save slot selection menu without loading existing game data.
- `this.DeactivateMenu();`: Hides the main menu.

```
public void OnNewGameClicked()
{
    saveSlotsMenu.ActivateMenu(false);
    this.DeactivateMenu();
}
```

Continue Game: The *Continue* button allows the player to load a saved game. It uses the **DataPersistenceManager** to load the game data and transition to the appropriate scene based on the saved data.

- `LoadGame()`: Retrieves saved game data and loads it into memory.
- `DetermineSceneToLoad()`: Determines the correct scene based on saved game data.

- `SceneManager.LoadSceneAsync(sceneToLoad);`: Loads the appropriate gameplay scene

```
public void OnContinueGameClicked()
{
    DisableMenuButton(); // Disable menu buttons to prevent multiple clicks
    DataPersistenceManager.instance.LoadGame(); // Load saved data
    DataPersistenceManager.instance.SaveGame(); // Optionally save game data

    string sceneToLoad = DataPersistenceManager.instance.DetermineSceneToLoad(); // D
    Debug.Log($"Loading scene: {sceneToLoad}");

    if (loadingScreen != null)
    {
        loadingScreen.SetActive(true); // Display loading screen
    }

    SceneManager.LoadSceneAsync(sceneToLoad); // Asynchronously load the scene
}
```

Load Game: This function shows the save slots menu, allowing the player to select which saved game to load. It deactivates the main menu during this process.

```
public void OnLoadGameClicked()
{
    saveSlotsMenu.ActivateMenu(true); // Open save slots for loading
    this.DeactivateMenu(); // Hide the main menu
}
```

Quit Game: When the *Quit* button is clicked, the application will quit. It also handles quitting from the Unity editor by stopping the play mode.

```
public void OnQuitGameClicked()
{
    Debug.Log("Quit Game Clicked");
    Application.Quit();

    #if UNITY_EDITOR
    UnityEditor.EditorApplication.isPlaying = false; // Stops play mode in Unity Edi
    #endif
}
```

3. Pause Controller

The **Pause Controller** script handles pausing and resuming the game, managing the pause menu, restarting chapters or checkpoints, transitioning to the main menu, controlling game audio during pause, and showing tutorials. It uses Unity's **Player Input** system to detect pause inputs and allows for managing the game's flow and state when paused or resumed.

1. Pausing and Resuming the Game:

- The script toggles the game's paused state based on the player's input (using the "PauseOpen" action in the **PlayerInput** system or manually with the 'R' key).

```
2 references
public void Pause()
{
    isPaused = true;
    pauseMenuUI.SetActive(true);
    Time.timeScale = 0f; // Pause the game
    ControlAudioSources(true); // Mute or pause all audio sources
                               // Use InputManager to switch to UI action map
}

2 references
public void Resume()
{
    isPaused = false;
    pauseMenuUI.SetActive(false);
    Time.timeScale = 1f; // Resume game time
    ControlAudioSources(false); // Unpause all audio sources
                                // Use InputManager to switch back to Player action map
}
```

- It adjusts the **Time.timeScale** to `0f` (paused) or `1f` (running), freezes or resumes game time accordingly, and controls all audio sources except for the main theme and background music.

2. Restarting the Chapter or Last Checkpoint:

- The **RestartChapter()** and **RestartFromLastCheckpoint()** methods allow players to restart the current chapter or checkpoint. They ensure that the game state is either reset or reloaded from the last save point.

```
0 references
public void RestartChapter()
{
    // Ensure DataPersistenceManager instance is available
    if (DataPersistenceManager.instance != null)
    {
        // Indicate that a reset is requested
        DataPersistenceManager.instance.shouldReset = true;

        // Save the game to apply the reset immediately
        DataPersistenceManager.instance.SaveGame();

        // Ensure game time is resumed before reloading
        Time.timeScale = 1f;

        if (loadingScreen != null)
        {
            loadingScreen.SetActive(true); // Activate the loading screen
        }


        // Reload the current scene
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
    else
    {
        Debug.LogError("DataPersistenceManager instance not found. Cannot restart chapter.");
    }
}
```

```
0 references
public void RestartFromLastCheckpoint()
{
    // Check if DataPersistenceManager instance is available
    if (DataPersistenceManager.instance != null)
    {
        if (loadingScreen != null)
        {
            loadingScreen.SetActive(true); // Activate the loading screen
        }

        // Reload the current scene
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);

        // Load the most recent game state, which should be the last checkpoint
        DataPersistenceManager.instance.LoadGame();

        Debug.Log("Restarted from last checkpoint");
        pauseMenuUI.SetActive(false);
        Time.timeScale = 1f;


    }
    else
    {
        Debug.LogError("DataPersistenceManager instance not found. Cannot restart from last checkpoint.");
    }
}
```

3. Transitioning to the Main Menu:

- The **LoadMainMenu()** method manages the transition back to the main menu, ensuring that the current scene's data is saved for future use. It updates the previous and current scene names before transitioning.

```
public void LoadMainMenu()
{
    SceneData currentSceneData = DataPersistenceManager.instance.sceneDataHandler.LoadScen
    currentSceneData.previousScene = currentSceneData.currentScene;
    currentSceneData.currentScene = "GameMainMenu";
    DataPersistenceManager.instance.sceneDataHandler.SaveSceneData(currentSceneData);

    Time.timeScale = 1f;
    SceneManager.LoadScene("GameMainMenu");
}
```

4. Save Slot

The `SaveSlot` class is a Unity component responsible for managing individual save slots within the game's save/load system. It displays whether there is data associated with the save slot and presents relevant information such as the last played level and the player's progress.

1. Serialized Fields

The class contains several serialized fields that are set in the Unity Inspector:

```
[Header("ProfileId")]
[SerializeField] private string profileId = "";

[Header("Content")]
[SerializeField] private GameObject noDataContent;
[SerializeField] private GameObject hasDataContent;

[SerializeField] private TextMeshProUGUI percentageCompleteText;
[SerializeField] private TextMeshProUGUI levelDisplay;

private Button saveSlotButton;
```

- **`profileId`**: A unique identifier for the save slot, differentiating between player profiles or save files.
- **`noDataContent`**: A `GameObject` displayed when there is no saved data for the slot.
- **`hasDataContent`**: A `GameObject` shown when there is saved data.
- **`percentageCompleteText`**: Currently not used.
- **`levelDisplay`**: A UI text element showing the name of the level associated with the save.
- **`saveSlotButton`**: A reference to the button component for user interactions.

2. Setting Save Slot Data

The primary functionality of the `Save Slot` class is implemented in the `Set Data` method, which updates the UI based on the provided game data and profile ID:

```
1 reference
public void SetData(GameData data, string profileId)
{
    // Check if there's no data for this profileId
    if (data == null)
    {
        noDataContent.SetActive(true);
        hasDataContent.SetActive(false);
    }
    else
    {
        noDataContent.SetActive(false);
        hasDataContent.SetActive(true);

        // Load the save slot data for this profile
        SaveSlotData saveSlotData = DataPersistenceManager.instance.LoadSaveSlotData(profileId);
        if (saveSlotData != null && !string.IsNullOrEmpty(saveSlotData.lastPlayedScene))
        {
            // Directly use the scene name as the level display
            levelDisplay.text = saveSlotData.lastPlayedScene;
        }
        else
        {
            // Use a default text or placeholder if no specific scene data found
            levelDisplay.text = "Start Your Adventure!"; // Or "Level 1" if you prefer
        }

        // Update the game progress percentage text
        percentageCompleteText.text = "Game In Progress"; // Update this based on your game's logic
    }
}
```

- The method first checks if `data` is `null` to determine if there is no saved game data for the profile.
- If `data` is `null`, it activates `noDataContent` and deactivates `hasDataContent`.
- If `data` is valid, it loads the save slot data using the `DataPersistenceManager` and updates the `levelDisplay` to show the last played scene. If no scene data is available, it displays a default message.
- The method also updates the `percentageCompleteText` to indicate that a game is in progress.

3. Accessing Profile ID and Interactivity Control

```
2 references
public string GetProfileId()
{
    return this.profileId;
}

3 references
public void Setinteractable(bool interactable)
{
    saveSlotButton.interactable = interactable;
}
```

The `GetProfileId` method returns the `profileId` associated with the save slot. This allows other components to identify which save slot the user is interacting with.

The `SetInteractable` method allows enabling or disabling the save slot button.
This functionality is useful for managing user interactions based on the game's logic (e.g., disabling slots without data).

5. Save Slot Menu

1. Handling Save Slot Selection

The `OnSaveSlotClicked` method processes interactions with a selected save slot:

```
0 references
public void OnSaveSlotClicked(SaveSlot saveSlot)
{
    // Disable all buttons
    DisableMenuButtons();

    // Update the selected profile ID for data persistence
    string profileId = saveSlot.GetProfileId();
    DataPersistenceManager.instance.ChangeSelectedProfileId(profileId);

    if (loadingScreen != null)
    {
        loadingScreen.SetActive(true); // Activate the loading screen
    }

    string sceneToLoad;

    // Check whether we are loading a game or starting a new game
    if (isLoadingGame)
    {
        // If loading a game, attempt to load the last played scene
        SaveSlotData saveSlotData = DataPersistenceManager.instance.LoadSaveSlotData(profileId);
        sceneToLoad = saveSlotData != null && !string.IsNullOrEmpty(saveSlotData.lastPlayedScene)
                      ? saveSlotData.lastPlayedScene
                      : "Chapter1"; // Fallback to "Chapter1" if no data found
    }
    else
    {
        // If starting a new game, ignore last played scene and load "Chapter1"
        DataPersistenceManager.instance.NewGame();
        sceneToLoad = "Chapter1";
    }

    //save game data
    //DataPersistenceManager.instance.SaveGame();

    // Load the determined scene
    SceneManager.LoadSceneAsync(sceneToLoad);
}
```

- The method starts by disabling all buttons to prevent further interactions during the loading process.
- It updates the selected profile ID for data persistence and activates the loading screen.
- Depending on whether the game is being loaded or a new game is starting, it determines the scene to load. If loading a game, it checks the last played scene and falls back to "Chapter1" if no data exists. If starting a new game, it initializes the game data before loading "Chapter1".

2. Menu Activation

The `ActivateMenu` method displays the menu and populates the save slots:

```
2 references
public void ActivateMenu(bool isLoadingGame)
{
    //Set this menu to be active
    this.gameObject.SetActive(true);

    //set mode
    this.isLoadingGame = isLoadingGame;

    //load all of  the profiles that exist
    Dictionary<string, GameData> profilesGameData = DataPersistenceManager.instance.GetAllProfilesGameData();

    //loop through each save slot in the UI and set the content appropriately
    GameObject firstSelected = backButton.gameObject;

    foreach (SaveSlot saveSlot in saveSlots)
    {
        GameData profileData = null;
        string profileId = saveSlot.GetProfileId(); // Get the profile ID for each save slot
        profilesGameData.TryGetValue(profileId, out profileData);

        // Pass both profileData and profileId to the SetData method
        saveSlot.SetData(profileData, profileId);

        //disable interaction if not data in saveslot
        //enable if have data
        if (profileData == null && isLoadingGame)
        {
            saveSlot.Setinteractable(false);
        }
        else
        {
            saveSlot.Setinteractable(true);
            //defaulting saveslot position
            if (firstSelected.Equals(backButton.gameObject))
            {
                firstSelected = saveSlot.gameObject;
            }
        }
    }
    //set the first selected button
    StartCoroutine(this.SetFirstSelected(firstSelected));
}
```

- The method activates the menu and retrieves all existing game profiles.
- It iterates through each `SaveSlot`, retrieving the associated profile data and updating the slot's display with the `SetData` method.
- It manages the interactivity of save slots based on whether they contain data, ensuring that users cannot select empty slots when loading a game.

3. Menu Buttons Deactivation

The `DeactivateMenu` method hides the menu when it's no longer needed. This method simply sets the menu's active state to false.

```
1 reference
public void DeactivateMenu()
{
    this.gameObject.SetActive(false);
}

1 reference
private void DisableMenuButtons()
{
    foreach(SaveSlot saveSlot in saveSlots)
    {
        saveSlot.Setinteractable(false);
    }
    backButton.interactable = false;
}
```

The `DisableMenuButtons` method disables all buttons in the menu. This functionality ensures that the player cannot interact with the menu while a loading process is underway.

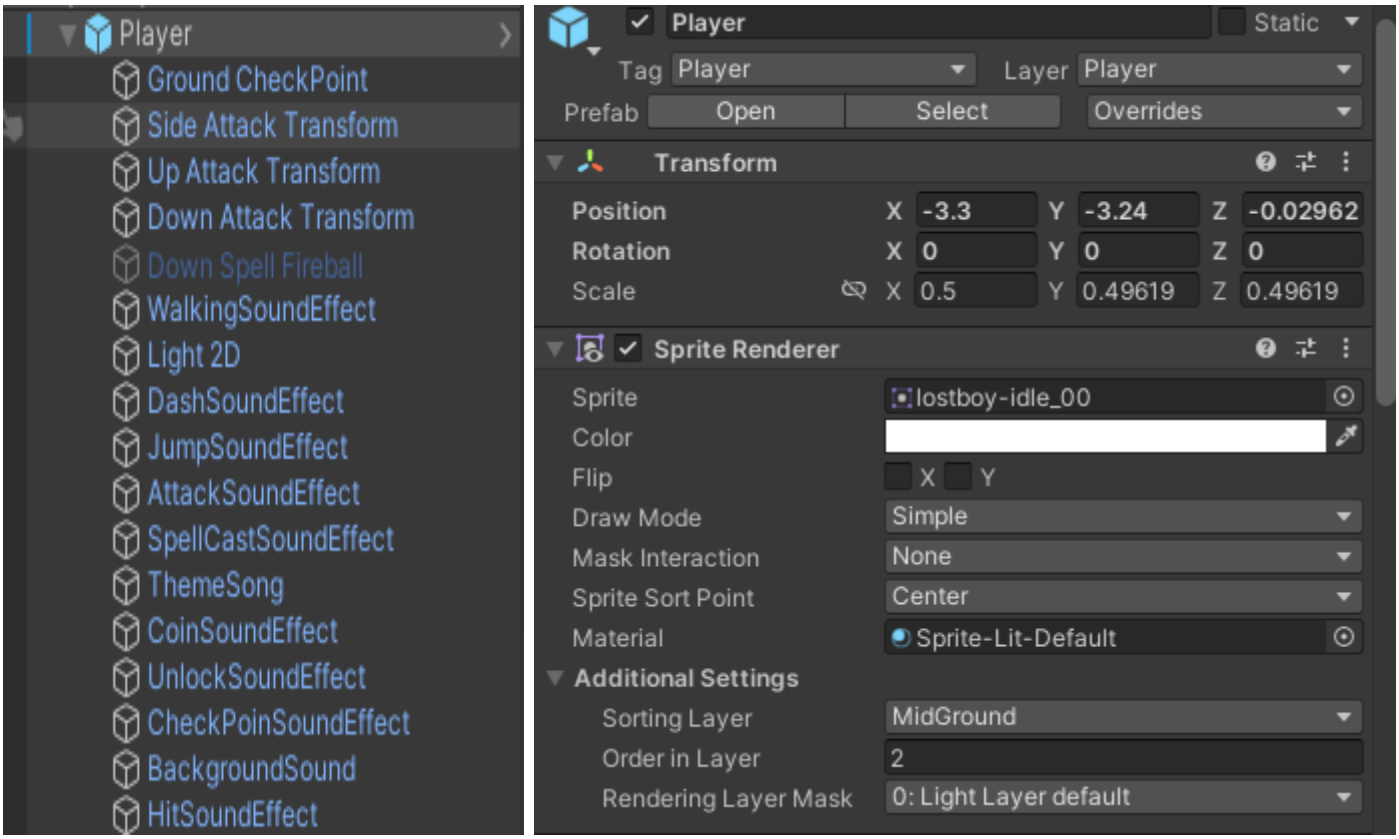
4.4 Player Controller System

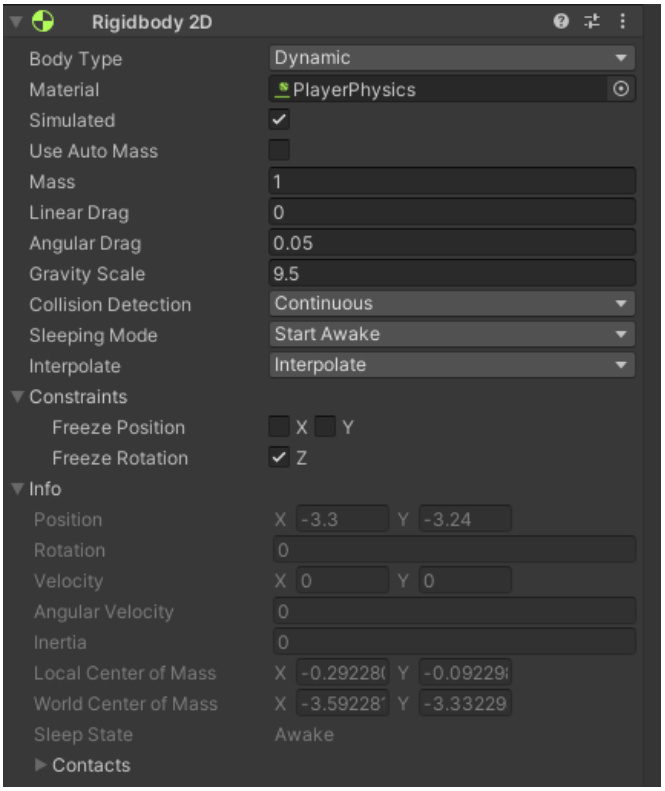
1. Core Concepts

Player controller is the spot light the game and it is second most difficult systems to implement after saving system. In the project file, the script for player has over 1500 lines of code. Thus, we cannot cover every single functions or codes in this report but we will go through how player can control each action like idle, run, jump, attack, special power, healing, dashing and how player states are transition from one to another. We will explore how setting up the player prefab (object) works.

In hierarchy, the player has several child objects
1. Ground check point to check whether player is standing on platform or on the wall or falling
2. Side, Up, Down attack points to mark the area that will be effected damage to enemies
3. Light 2D object to highlight the player in the scene

4.The rest are sound effects of each actions

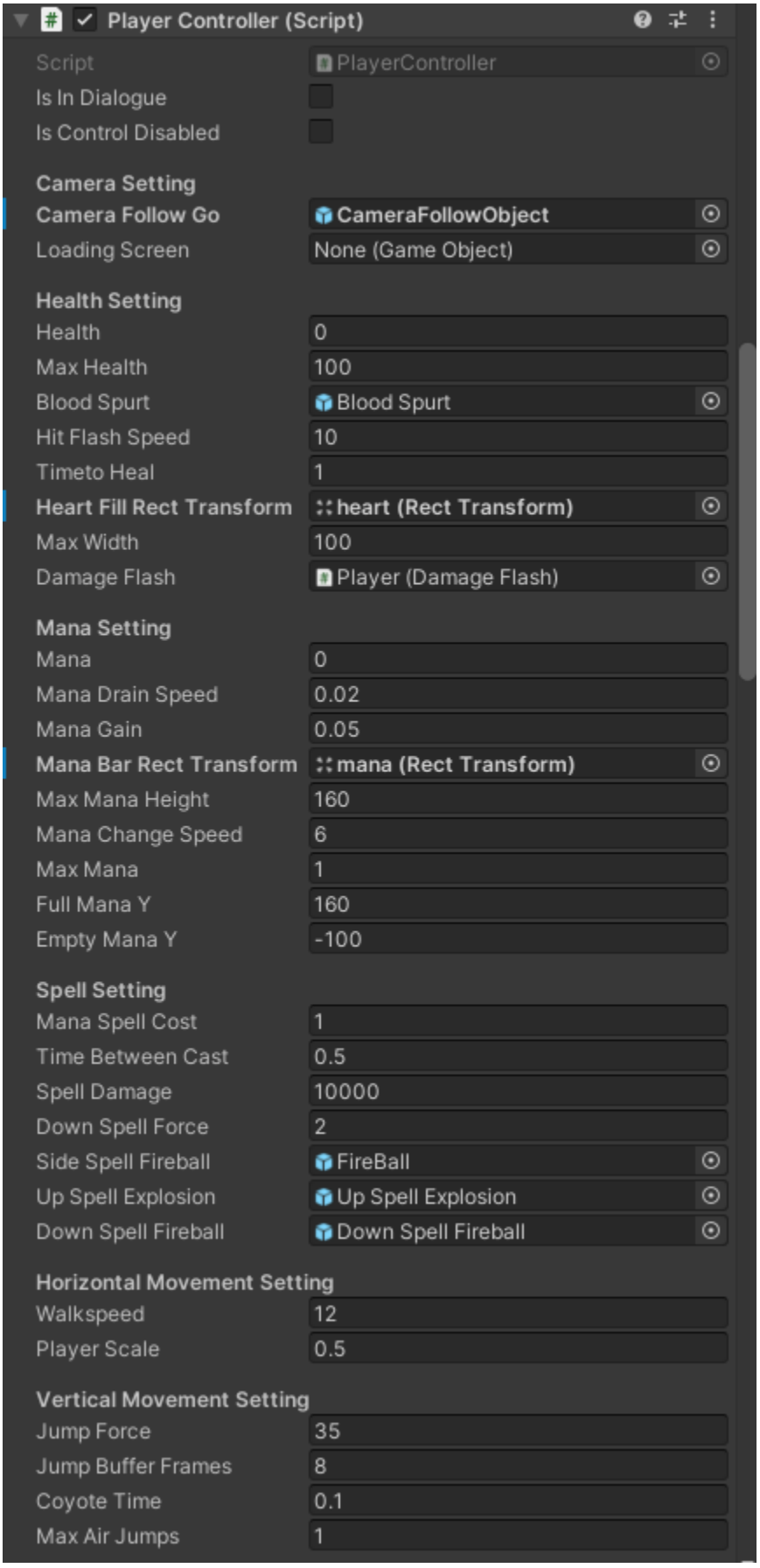


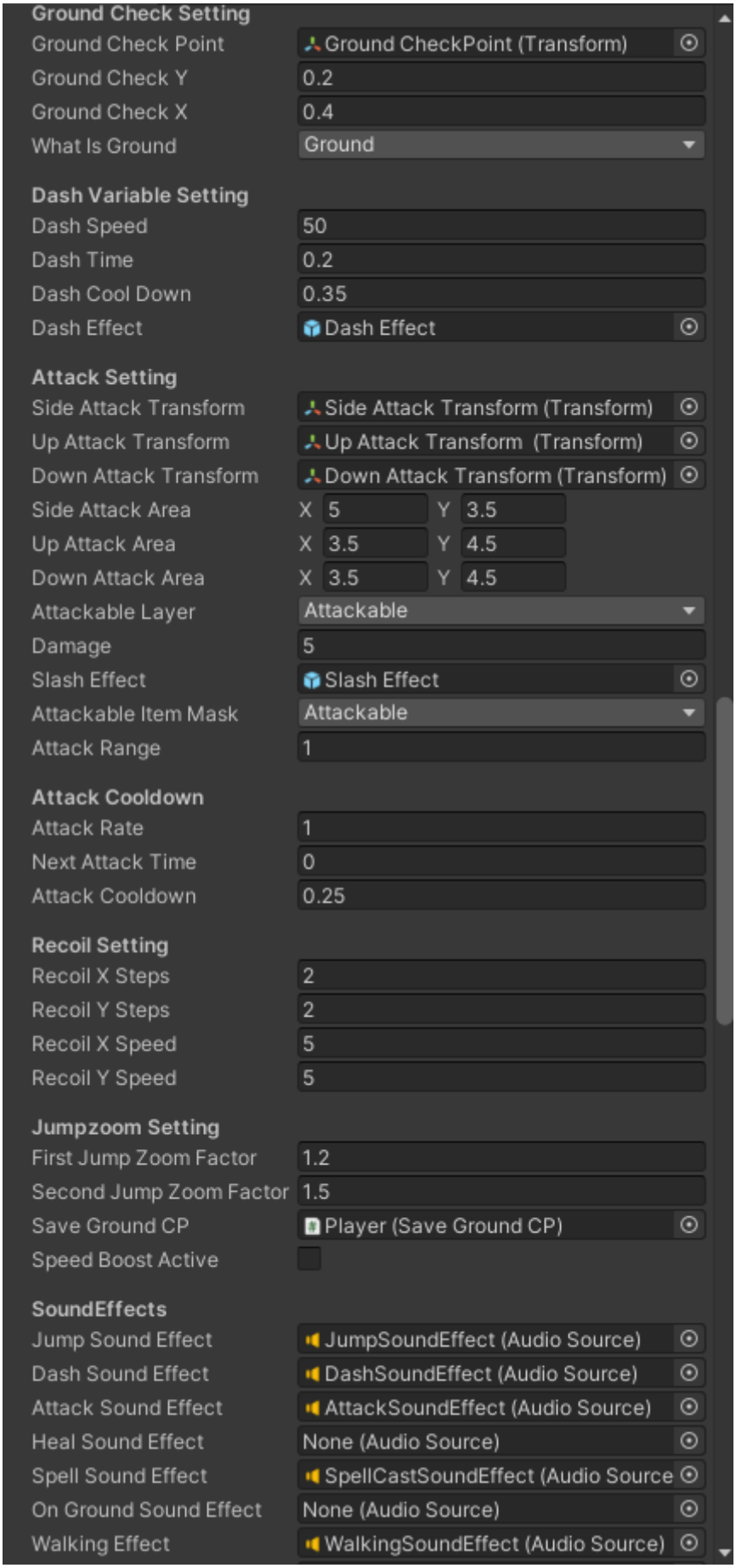


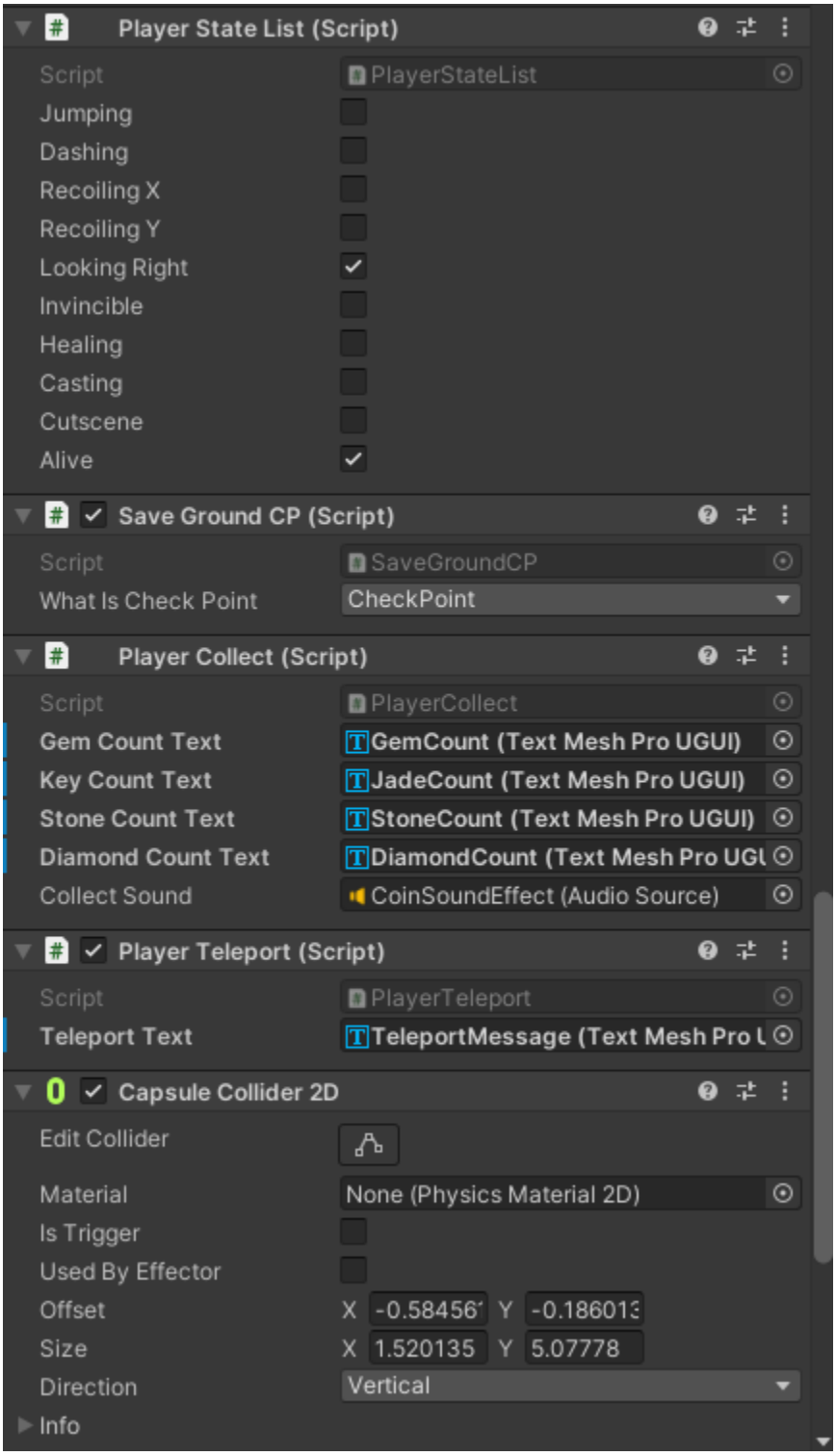
In unity, we work with components that hold different properties including program scripts. The first script is **Sprite Render** which show the image of player.

Second Component is Rigid Body 2d which is to calculate gravity value of player so we can manage jump, walking and falling. We set Collision Detection mode at Continuous to always keep detecting every frame and Set Interpolation mode so it will not glitter when game is playing.

And Finally, the main script of the system, player controller will have varieties of variables and functions that controls the player.


Player Controller (Script)
Script
PlayerController
Is In Dialogue
Is Control Disabled
Camera Setting
Camera Follow Go
CameraFollowObject
Loading Screen
None (Game Object)
Health Setting
Health
0
Max Health
100
Blood Spurt
Blood Spurt
Hit Flash Speed
10
Timeto Heal
1
Heart Fill Rect Transform
heart (Rect Transform)
Max Width
100
Damage Flash
Player (Damage Flash)
Mana Setting
Mana
0
Mana Drain Speed
0.02
Mana Gain
0.05
Mana Bar Rect Transform
mana (Rect Transform)
Max Mana Height
160
Mana Change Speed
6
Max Mana
1
Full Mana Y
160
Empty Mana Y
-100
Spell Setting
Mana Spell Cost
1
Time Between Cast
0.5
Spell Damage
10000
Down Spell Force
2
Side Spell Fireball
FireBall
Up Spell Explosion
Up Spell Explosion
Down Spell Fireball
Down Spell Fireball
Horizontal Movement Setting
Walkspeed
12
Player Scale
0.5
Vertical Movement Setting
Jump Force
35
Jump Buffer Frames
8
Coyote Time
0.1
Max Air Jumps
1


Ground Check Setting
Ground Check Point
Ground CheckPoint (Transform)
Ground Check Y
0.2
Ground Check X
0.4
What Is Ground
Ground
Dash Variable Setting
Dash Speed
50
Dash Time
0.2
Dash Cool Down
0.35
Dash Effect
Dash Effect
Attack Setting
Side Attack Transform
Side Attack Transform (Transform)
Up Attack Transform
Up Attack Transform (Transform)
Down Attack Transform
Down Attack Transform (Transform)
Side Attack Area
X 5
Y 3.5
Up Attack Area
X 3.5
Y 4.5
Down Attack Area
X 3.5
Y 4.5
Attackable Layer
Attackable
Damage
5
Slash Effect
Slash Effect
Attackable Item Mask
Attackable
Attack Range
1
Attack Cooldown
Attack Rate
1
Next Attack Time
0
Attack Cooldown
0.25
Recoil Setting
Recoil X Steps
2
Recoil Y Steps
2
Recoil X Speed
5
Recoil Y Speed
5
Jumpzoom Setting
First Jump Zoom Factor
1.2
Second Jump Zoom Factor
1.5
Save Ground CP
Player (Save Ground CP)
Speed Boost Active
SoundEffects
Jump Sound Effect
JumpSoundEffect (Audio Source)
Dash Sound Effect
DashSoundEffect (Audio Source)
Attack Sound Effect
AttackSoundEffect (Audio Source)
Heal Sound Effect
None (Audio Source)
Spell Sound Effect
SpellCastSoundEffect (Audio Source
On Ground Sound Effect
None (Audio Source)
Walking Effect
WalkingSoundEffect (Audio Source)


Player State List (Script)
Script
PlayerStateList
Jumping
Dashing
Recoiling X
Recoiling Y
Looking Right
Invincible
Healing
Casting
Cutscene
Alive
Save Ground CP (Script)
Script
SaveGroundCP
What Is Check Point
CheckPoint
Player Collect (Script)
Script
PlayerCollect
Gem Count Text
GemCount (Text Mesh Pro UGUI)
Key Count Text
JadeCount (Text Mesh Pro UGUI)
Stone Count Text
StoneCount (Text Mesh Pro UGUI)
Diamond Count Text
DiamondCount (Text Mesh Pro UGU
Collect Sound
CoinSoundEffect (Audio Source)
Player Teleport (Script)
Script
PlayerTeleport
Teleport Text
TeleportMessage (Text Mesh Pro U
Capsule Collider 2D
Edit Collider
Material
None (Physics Material 2D)
Is Trigger
Used By Effector
Offset
X -0.58456 Y -0.18601
Size
X 1.520135 Y 5.07778
Direction
Vertical
Info

2. Health and Mana Management

- **Logic Flow**:
  - When the player takes damage or uses healing items, the **health** variable is updated.
  - The health is **clamped** to make sure it doesn't drop below 0 or exceed the player's max health.
  - After the health value changes, the **UI is updated** (e.g., health bar or heart icons) to reflect the new value.
  - Similarly, for **mana**, when spells are cast or mana is replenished, the mana variable is updated, clamped, and the **mana bar UI** is refreshed to show the new amount.

```
3 references
public float Health
{
    get { return health; }
    set
    {
        if (health != value)
        {
            health = Mathf.Clamp(value, 0f, maxHealth);
            UpdateHeartUI(); // Update the heart UI to reflect the change.
            //Debug.Log($"health updated to: {health}");
        }
    }
}
```

```
14 references
public float Mana
{
    get { return mana; }
    set
    {
        mana = Mathf.Clamp(value, 0, 1);
        UpdateManaUI(mana); // Update the UI with the current mana percentage
        //Debug.Log($"Mana updated to: {mana}");
    }
}
```

- **Purpose**: The health and mana management ensures that the player's stats are maintained within appropriate limits and that the user interface remains synchronized with the game state.

---

3. Ground Detection (Grounded Check)

- **Logic Flow**:
  - The player's character needs to know whether it's on the ground to allow actions like **jumping** or **dashing**.
  - Several **ray casts** are projected downward from the player's position.
  - If any of these rays hit the ground, the player is considered **grounded**.
  - If grounded, jumping is enabled, and certain animations and mechanics, like double-jumping, are reset.

```
19 references
public bool Grounded()
{
    if (Physics2D.Raycast(groundCheckPoint.position, Vector2.down, groundCheckY, WhatIsGround)
        || Physics2D.Raycast(groundCheckPoint.position + new Vector3(groundCheckX, 0, 0), Vector2.down, groundCheckY, WhatIsGround)
        || Physics2D.Raycast(groundCheckPoint.position + new Vector3(-groundCheckX, 0, 0), Vector2.down, groundCheckY, WhatIsGround))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

- **Purpose**: This mechanism is used to allow the player to jump, dash, or perform other ground-based actions only when they are grounded, preventing unintended jumps in mid-air unless explicitly allowed (like a double jump).

---

4. Movement (Walking)

- **Logic Flow**:
  - The player provides **input** (e.g., pressing the arrow keys or using the joystick).
  - This input is translated into horizontal movement by **updating the velocity** of the player character.
  - The character's **walking animation** is triggered based on movement, and **sound effects** (footsteps) play when the character is moving.
  - When no input is provided, the character's velocity is set to zero, and the **idle animation** is triggered.

```
1 reference
private void Move()
{
    if (isInDialogue || isControlDisabled)
    {
        rb.velocity = new Vector2(0, rb.velocity.y); // Stop horizontal movement
        anim.SetBool("Walking", false); // Ensure the walking animation is not playing
        return; // Skip the rest of the method
    }

    // Use currentMovementInput to set the velocity
    float moveX = currentMovementInput.x * walkspeed;
    float moveY = rb.velocity.y;
    rb.velocity = new Vector2(moveX, moveY);

    // Update the animator with the walking state
    anim.SetBool("Walking", moveX != 0 && Grounded());

    // Check if the player is walking
    bool isWalking = moveX != 0 && Grounded(); // Declare isWalking here
    anim.SetBool("Walking", isWalking);

    // Play/Stop walking sound based on movement
    if (isWalking && !WalkingEffect.isPlaying)
    {
        WalkingEffect.Play();
    }
    else if (!isWalking && WalkingEffect.isPlaying)
    {
        WalkingEffect.Stop();
    }
}
```

- **Purpose**: This system handles the basic movement of the player, allowing them to move left and right with fluid transitions between walking and idle states.

---

5. Jumping & Double Jumping

- **Logic Flow**:
    - **Jump Input**: When the player presses the jump button, the game checks if the character is grounded.
    - If grounded, the player's **vertical velocity** is increased to initiate the jump.
    - If the player holds the jump button, the jump can be **extended**, allowing for variable jump heights.
    - If the player is **falling**, they might still be able to jump if the game allows for **coyote time** (a short grace period after walking off a ledge).
    - For **double jumping**, the player can perform a second jump if they haven't already done so in the air, even without being grounded.

```
1 reference
void Jump()
{
    if (isInDialogue || isControlDisabled) return; // Skip if in dialogue or controls are disabled
                                                   // Jump logic is now handled by OnJumpPerformed.
                                                   // The following is to handle the "letting go" of the jump button for variable jump height

    if (Input.GetButtonUp("Jump") && rb.velocity.y > 3)
    {
        pState.jumping = false;
        rb.velocity = new Vector2(rb.velocity.x, 0);
        Debug.Log("PlayerJumped");
        // jumpSoundEffect.Play(); // Commented out to prevent jump sound effect when releasing jump button
    }

    anim.SetBool("Jumping", !Grounded());
}
```

```
1 reference
void UpdateJumpVariable()
{
    if (Grounded())
    {
        pState.jumping = false;
        coyoteTimeCounter = coyoteTime;
        airJumpCounter = 0;



    }
    else
    {
        coyoteTimeCounter -= Time.deltaTime;
    }

    if (Input.GetButtonDown("Jump"))
    {
        jumpbufferCounter = jumpBufferFrames;
        //WalkingEffect.Stop();
        //jumpSoundEffect.Play();

        //WalkingEffect.Stop();


    }
    else
    {
        jumpbufferCounter--;
    }
}
Unity Message | 0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("MovingPlatform"))
    {
        currentPlatform = collision.gameObject;
        lastPlatformPosition = currentPlatform.transform.position;
    }
}

Unity Message | 0 references
private void OnCollisionExit2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("MovingPlatform"))
    {
        currentPlatform = null;
    }
}
```

- **Purpose**: This gives players more control over their jumps, with flexibility for different jump heights and the ability to double jump, adding fluidity to the platforming experience.

---

6. Dashing

- **Logic Flow**:
  - When the player presses the **dash button**, the player's movement is quickly propelled in the **facing direction**.
  - **Gravity is disabled** temporarily during the dash to prevent the player from falling mid-dash.
  - The player also becomes **invincible** during the dash, allowing them to avoid enemies or obstacles.
  - Once the dash completes, the **cooldown timer** begins, preventing the player from spamming the dash repeatedly.

```
1 reference
private IEnumerator Dash()
{
    canDash = false;
    pState.dashing = true;
    dashCoroutine = StartCoroutine(MakeInvincible(dashTime, false)); // Make the player invincible during dash without flashing effect
    anim.SetTrigger("Dashing");
    rb.gravityScale = 0;
    int _dir = pState.lookingRight ? 1 : -1;
    rb.velocity = new Vector2(_dir * dashSpeed, 0);
    if (Grounded()) Instantiate(dashEffect, transform);

    WalkingEffect.Stop();
    dashSoundEffect.Play();

    yield return new WaitForSeconds(dashTime);
    rb.gravityScale = gravity;
    pState.dashing = false;
    yield return new WaitForSeconds(dashCoolDown);
    canDash = true;
}
```

- **Purpose**: Dashing allows for quick bursts of movement, making it useful for dodging attacks or navigating obstacles, while the cooldown period ensures the mechanic is not overused.

---

7. Attacking in Four Directions

- **Logic Flow**:
  - The player can attack by pressing the **attack button** and giving a **directional input** (up, down, or none for sideways).
  - Based on the direction:
    - A **side attack** is performed when no directional input is provided.
    - An **upward attack** happens if the player presses up while attacking.

        - A **downward attack** occurs if the player is airborne and presses down while attacking.
    - When attacking, the game checks the position of nearby enemies or objects and applies **damage** to them if they're within range.
    - If certain attacks (like downward attacks) are performed, the player might experience a **recoil effect**, moving them slightly in the opposite direction.

```
2 references
void Attack()
{
    if (isInDialogue || isControlDisabled) return; // Skip attack logic if in dialogue or controls are disabled

    timeSinceAttack += Time.deltaTime;
    if (attack && timeSinceAttack >= timeBetweenAttack)
    {
        timeSinceAttack = 0;
        anim.SetTrigger("Attacking");

        // Update the Animator's Speed parameter even during the attack
        anim.SetFloat("Speed", Mathf.Abs(xAxis)); // Add this line

        Vector3 effectScale = new Vector3(1f, 1f, 1f); // Define the desired scale for the slash effect

        // Perform the attack based on the vertical axis and grounded state
        if (yAxis == 0 || (yAxis < 0 && Grounded()))
        {
            Hit(SideAttackTransform, SideAttackArea, ref pState.recoilingX, recoilXSpeed);
            float effectAngle = pState.lookingRight ? 0f : 180f;
            SlashEffectAtAngle(slashEffect, effectAngle, SideAttackTransform.position, effectScale);
            Debug.Log("Side Attack!");
            PlayAttackSound();
            CheckForAttackableItems(SideAttackTransform);
        }
        else if (yAxis > 0)
        {
            Hit(UpAttackTransform, UpAttackArea, ref pState.recoilingY, recoilYSpeed);
            SlashEffectAtAngle(slashEffect, 80, UpAttackTransform.position, effectScale);
            Debug.Log("Up Attack!");
            PlayAttackSound();
            CheckForAttackableItems(UpAttackTransform);
        }
        else if (yAxis < 0 || !Grounded())
        {
            Hit(DownAttackTransform, DownAttackArea, ref pState.recoilingY, recoilYSpeed);
            SlashEffectAtAngle(slashEffect, -90, DownAttackTransform.position, effectScale);
            Debug.Log("Down Attack!");
            PlayAttackSound();
            CheckForAttackableItems(DownAttackTransform);
        }
    }

    // Set the Idle parameter based on the xAxis value and whether the player is grounded
    anim.SetBool("Idle", xAxis == 0 && Grounded() && !pState.dashing);
}
```

- **Purpose**: This provides flexibility in combat, allowing the player to attack enemies in different directions, enhancing the strategic aspect of the game.

---

8. Spell Casting in Directions And Healing

- **Logic Flow**:
    - When the player presses the **spell button** and gives a **directional input**, a spell is cast in that direction (left, right, up, or down).
    - Each spell costs a certain amount of **mana**. If the player doesn't have enough mana, the spell will not be cast.

- The spell is then **projected** in the desired direction (e.g., a fireball moves to the side, or an explosion happens above).
- The **UI is updated** to reflect the reduced mana.
- Alternatively, Player can heal instead of spell casting
-

```
1 reference
void CastSpell()
{
    // Check if mana is full and other conditions are met before casting
    if (Input.GetButtonDown("Cast") && castOrHealTimer <= 0.05f && timeSinceCast >= timeBetweenCast && Mana >= manaSpellCost )
    {
        Debug.Log($"Mana before cast: {Mana}");
        Debug.Log($"Mana cost for casting: {manaSpellCost}");

        // Deduct the mana spell cost from the current mana
        Mana -= manaSpellCost;

        Debug.Log($"Mana after cast: {Mana}");

        pState.casting = true;
        timeSinceCast = 0;
        StartCoroutine(CastCoroutine());
    }
    else
    {
        timeSinceCast += Time.deltaTime;
    }

    // Disable down spell on ground
    if (Grounded())
    {
        downSpellFireball.SetActive(false);
    }

    // If down spell is active, force player down until ground
    if (downSpellFireball.activeInHierarchy)
    {
        rb.velocity += downSpellForce * Vector2.down;
    }
}
```

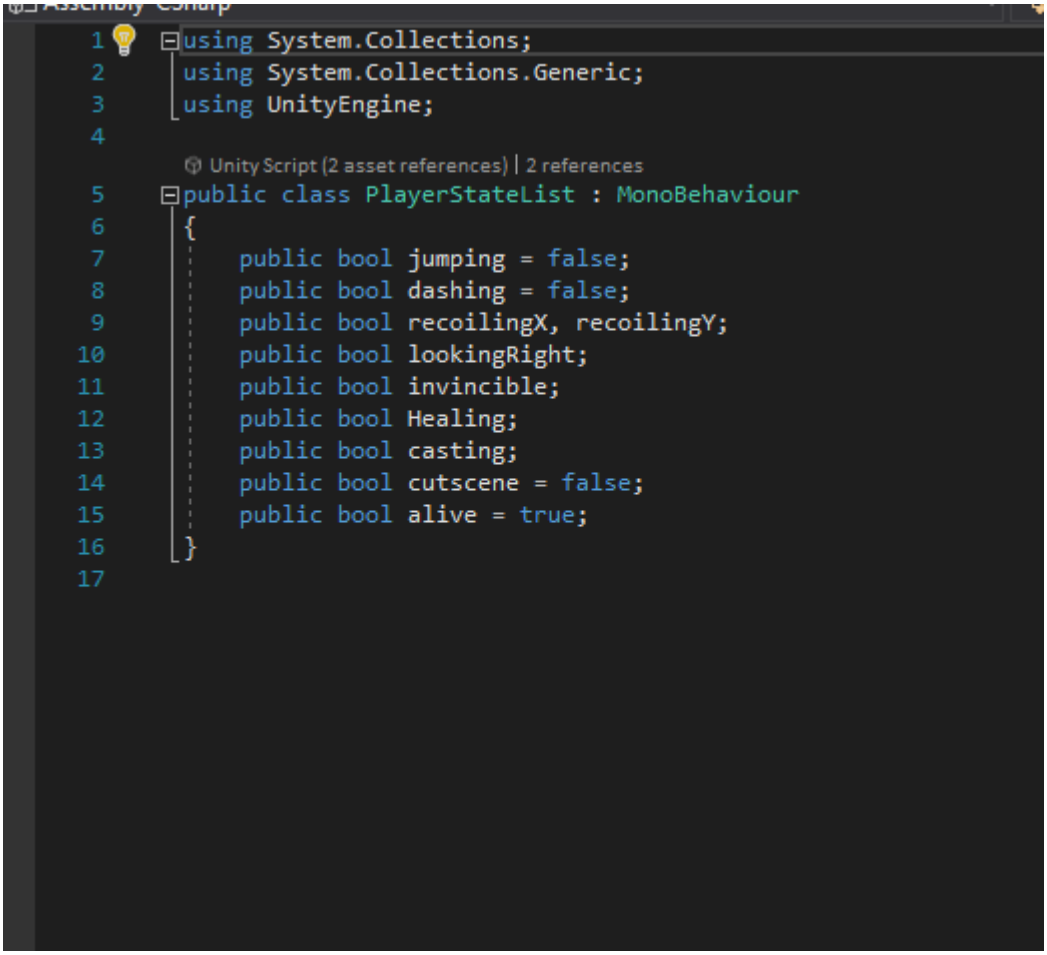
```
2 references
IEnumerator CastCoroutine()
{
    Debug.Log("Spell casting coroutine started.");
    anim.SetBool("Casting", true);
    yield return new WaitForSeconds(0.15f); //based on casting animtion, two parts, prep and cast state, take on frame on time 00:15 , frame time can change varies

    // Play spell sound effect
    spellSoundEffect.Play();

    //side cast
    if (yAxis == 0 || (yAxis < 0 && Grounded()))
    {
        GameObject _fireball = Instantiate(sideSpellFireball, SideAttackTransform.position, Quaternion.identity);

        //fireball
        if (pState.lookingRight)
        {
            _fireball.transform.eulerAngles = Vector3.zero; //if facing right, fireball continues as per normal
        }
        else
        {
            _fireball.transform.eulerAngles = new Vector2(_fireball.transform.eulerAngles.x, 180);
            //if not facing right, rotate the effect 180 degree
        }

        pState.recoilingX = true;
    }

    //up cast
    else if (yAxis > 0)
    {
        Instantiate(upSpellExplosion, transform);
        rb.velocity = Vector2.zero;
    }

    //down cast
    else if (yAxis < 0 && !Grounded())
    {
        downSpellFireball.SetActive(true);
    }

    //Mana = manaSpellCost;
    yield return new WaitForSeconds(0.35f);
    anim.SetBool("Casting", false);
    pState.casting = false;

}
```

- **Purpose**: Spells allow for ranged attacks, which can target enemies at different angles. The system balances combat with the mana pool, making players manage their resources carefully.

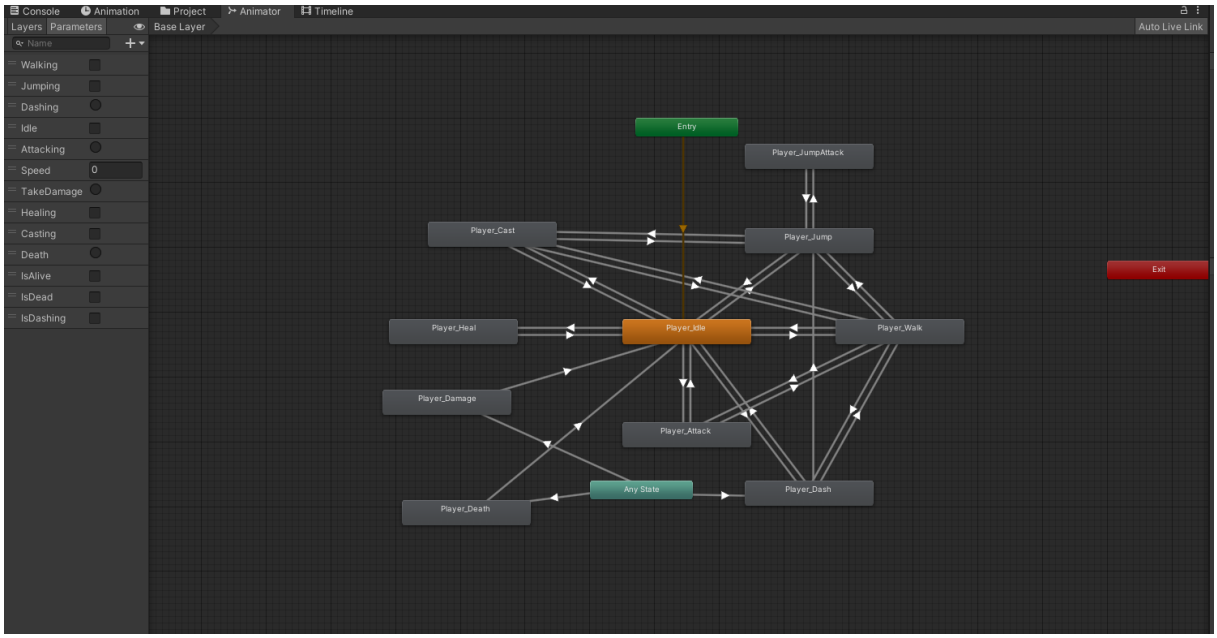
---

9. Interaction Between States

- **Logic Flow**:
  - The player's ability to perform actions like **moving, jumping, attacking**, or **casting spells** depends on their current state:
    - If the player is grounded, they can **jump or dash**.
    - If they are in the air, they can perform **air attacks or double jumps**.
    - If they are dashing, **other actions may be disabled** (e.g., casting spells) until the dash finishes.
  - The game continuously monitors the player's **state** and only allows actions that are appropriate for that state.
  - Certain external factors, like entering **dialogue mode** or **cutscenes**, may completely disable player control temporarily.
  - Player state lists are states to decide which animations to play according to the action player makes.



- **Purpose**: This ensures that the player's actions are contextual, allowing them to transition smoothly between different states (e.g., from walking to jumping) while preventing invalid actions (like dashing in mid-air unless allowed).



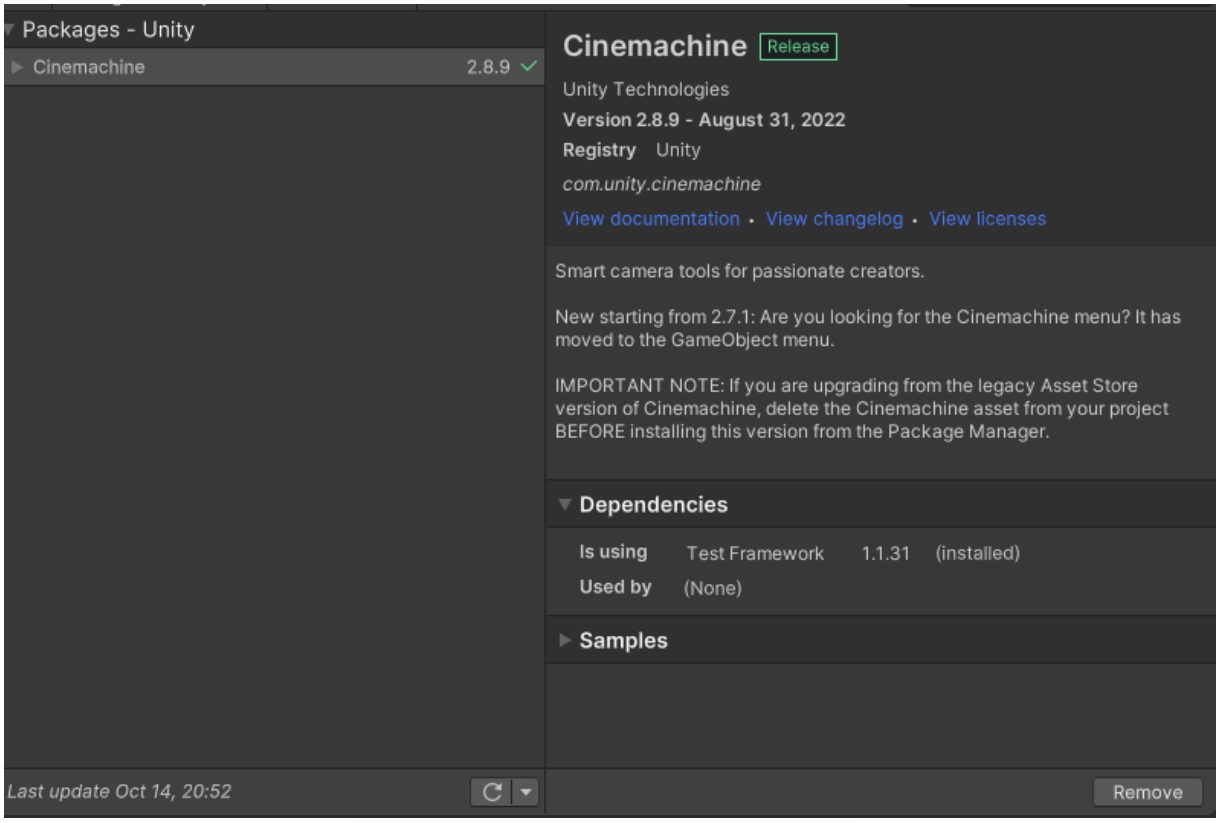
The Figure above show how states are transitioned between one after another by Booleans , conditions, and other factors.

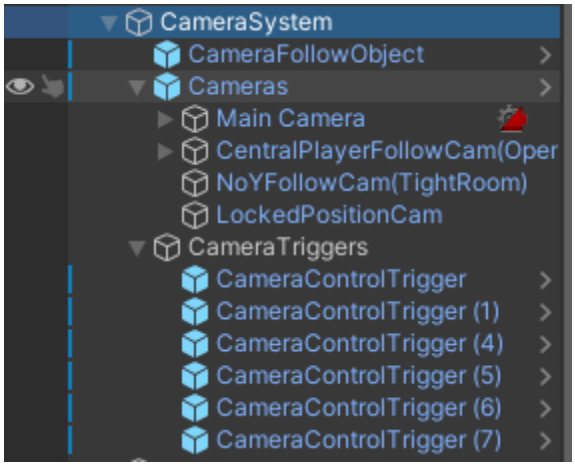
4.5 Camera System

1. Camera Setup

Camera panning refers to shifting the camera's position horizontally or vertically, adjusting the view relative to a tracked object. In this script, panning is implemented using the **CinemachineFramingTransposer**, which controls how the camera tracks the player or other objects. The main logic is handled by the `PanCameraOnContact()` and `PanCamera()` methods.

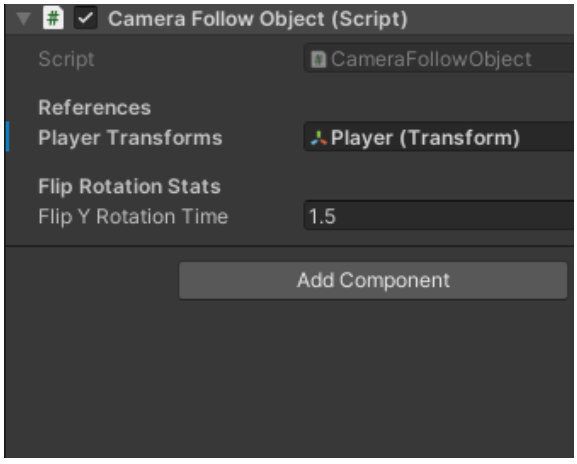
In this system, we are going to use a package library in Unity called Cine machine. Cine machine is ideal for those who wants to create Camera effect just like in movies and Industry level games and animations. In default Unity's camera system, the camera can only follow the player in static way and no special effects like camera shakes, camera cut out, camera delay can be done. But those efforts are easily done with Cine machine with just a few steps on a daily basis by developers. Thus, we install the package for our camera system.



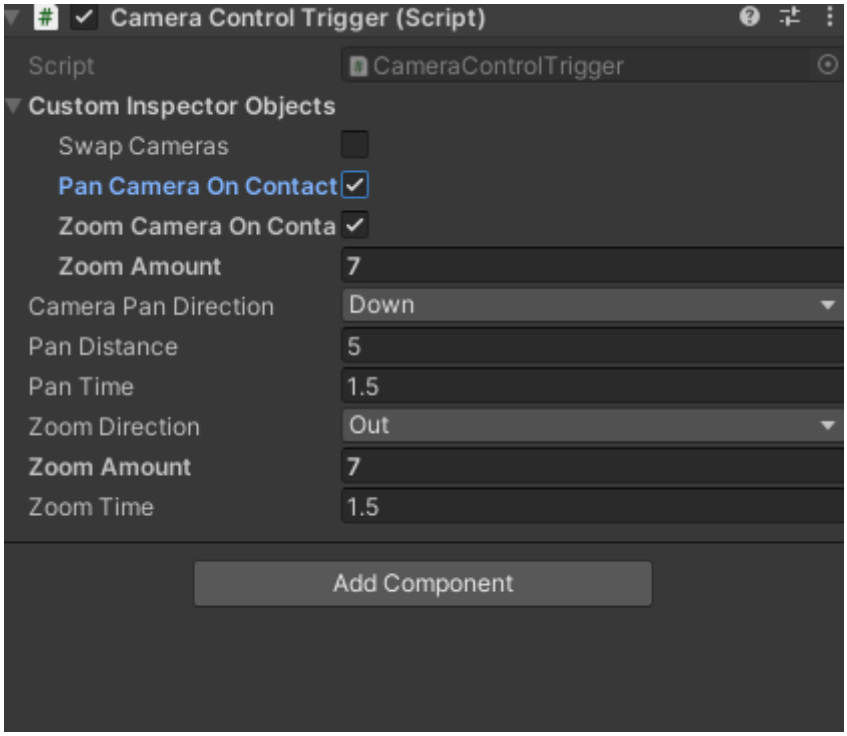


The Camera system is setup in this order.

1. **Camera follow object** : The object that will follow the player and the camera assigned to it.
2. **Cameras**: Collections of different types of camera for different purposes.
3. **Camera Triggers**: Area based trigger points where player enters, following effects can happen.
4. Pan in four directions, zoom in or out, mix of both.



The Camera Follow script is assigned to camera follow object and the camera trigger script is assigned to objects with trigger point areas.



Once the trigger script is attached, the developer can tweak the value of pan direction, pan amount, zoom amount and how many time it takes to complete those effect how fast or how slow.

They can also choose to have only pan effect, only zoom in or out effect or both.

This approach is a one time solution for developers that they don't have to go change value in their code files.

1. PanCameraOnContact Method:

- **Input Parameters:**
  - `panDistance`: The distance the camera should move (in Unity units).
  - `panTime`: How long the camera should take to complete the pan (in seconds).
  - `panDirection`: The direction in which to pan (up, down, left, right).
  - `panToStringPos`: A boolean flag indicating whether the camera should return to its original position.

```
2 references
public void PanCameraOnContact(float panDistance, float panTime, PanDirection panDirection, bool panToStringPos)
{
    if (_panCameraCoroutine != null)
    {
        StopCoroutine(_panCameraCoroutine); // Ensure any existing panning coroutine is stopped before starting a new one
    }
    _panCameraCoroutine = StartCoroutine(PanCamera(panDistance, panTime, panDirection, panToStringPos));
}
```

- **Explanation:**
  - This method starts the panning coroutine (`PanCamera()`) after stopping any existing panning to prevent conflicting movements.

- It helps in controlling camera panning upon events such as player movement or triggers in the game.

2. PanCamera Coroutine:

- **Flow:**
  - It calculates the start (`startingPos`) and end (`endPos`) positions for panning based on the camera's current tracked object offset and the direction (`panDirection`) provided.
  - Depending on the **PanDirection**:
    - `Up`: Moves the camera upward (`Vector2.up`).
    - `Down`: Moves the camera downward (`Vector2.down`).
    - `Left`: Moves the camera to the left (`Vector2.left`).
    - `Right`: Moves the camera to the right (`Vector2.right`).
  - If **panToStringPos** is true, it pans the camera back to its original tracked offset (`_startingTrackedObjectOffset`).
- **Calculations:**
  - Uses **linear interpolation (Lerp)** to smoothly move the camera between the starting position and the end position over the `panTime`.
  - The interpolation is performed by updating the camera's **m_TrackedObjectOffset** property, which adjusts the camera's tracking of the player.
- **Panning Execution:**
  - The panning happens gradually over `panTime`. Each frame, the camera moves a small portion of the total distance based on how much time has passed (`elapsedTime / panTime`).
  - The **Lerp** function calculates the intermediate position between `startingPos` and `endPos` for each frame, producing a smooth panning effect.
- **Completion:**
  - Once the elapsed time reaches `panTime`, the camera finishes its movement, and the panning ends, setting `isPanning` to false.

```
private IEnumerator PanCamera(float panDistance, float panTime, PanDirection panDirection, bool panToStringPos)
{
    isPanning = true; // Indicate that panning is in progress
    Vector2 endPos = Vector2.zero;
    Vector2 startingPos = _framingTransposer.m_TrackedObjectOffset; // Use the current offset as the starting position

    // Calculate the end position based on the pan direction and distance
    switch (panDirection)
    {
        case PanDirection.Up:
            endPos = Vector2.up * panDistance;
            break;
        case PanDirection.Down:
            endPos = Vector2.down * panDistance;
            break;
        case PanDirection.Left:
            endPos = Vector2.left * panDistance;
            break;
        case PanDirection.Right:
            endPos = Vector2.right * panDistance;
            break;
    }

    if (!panToStringPos)
    {
        endPos += startingPos; // Only modify endPos if not panning to string position
    }
    else
    {
        endPos = _startingTrackedObjectOffset; // If panning to string position, use the original stored offset
    }

    // Perform the actual panning over time
    float elapsedTime = 0f;
    while (elapsedTime < panTime)
    {
        elapsedTime += Time.deltaTime;
        Vector3 panLerp = Vector3.Lerp(startingPos, endPos, elapsedTime / panTime);
        _framingTransposer.m_TrackedObjectOffset = panLerp; // Apply the interpolated offset to the camera

        yield return null;
    }

    if (panToStringPos)
    {
        // If returning to starting position, explicitly set the offset to ensure it's accurate
        _framingTransposer.m_TrackedObjectOffset = _startingTrackedObjectOffset;
    }
    isPanning = false; // Panning is complete
}
```

---

2. Camera Zooming Logic

Camera zooming adjusts the camera's field of view (FOV), making the view closer or farther from the target object. This is managed using the **CinemachineVirtualCamera**'s `m_Lens.FieldOfView` property.

1. ZoomCamera Method:

- **Input Parameters:**
  - `amount`: The amount by which the FOV should change (i.e., the zoom level).
  - `duration`: The duration over which the zoom should happen (in seconds).
  - `zoomIn`: A Boolean flag indicating whether the camera should zoom in or out.
- **Flow:**
  - Stops any existing zoom coroutine if it's still running to prevent multiple zoom effects at once.
  - If `zoomIn` is true, the FOV is decreased (zoom in). Otherwise, it's increased (zoom out).
  - The **target FOV** is calculated by subtracting or adding the `amount` from the original FOV (`originalFOV`).

2. AdjustFOV Coroutine:

- **Flow:**
  - Similar to panning, zooming is done over time using **linear interpolation (Lerp)**.
  - The coroutine starts with the current FOV (`startFOV`) and smoothly transitions to the **target FOV** over the specified `duration`.
- **Calculations:**
  - Every frame, the **Lerp** function calculates the intermediate FOV between `startFOV` and `targetFOV`, based on how much time has elapsed (`time / duration`).
  - The FOV value is adjusted smoothly over time, resulting in a zoom effect that looks natural and fluid.

3. ResetZoom Method:

- This method resets the camera to its **original FOV** over a given `duration`. It functions similarly to `ZoomCamera`, but the target FOV is always set to the **original FOV**, restoring the default zoom level.

```
1 reference
public void ZoomCamera(float amount, float duration, bool zoomIn)
{
    if (_zoomCoroutine != null)
    {
        StopCoroutine(_zoomCoroutine); // Stop any ongoing zoom coroutine
    }
    float targetFOV = zoomIn ? originalFOV - amount : originalFOV + amount;
    _zoomCoroutine = StartCoroutine(AdjustFOV(targetFOV, duration));
}

1 reference
public void ResetZoom(float duration)
{
    if (_zoomCoroutine != null)
    {
        StopCoroutine(_zoomCoroutine); // Stop any ongoing zoom coroutine
    }
    _zoomCoroutine = StartCoroutine(AdjustFOV(originalFOV, duration));
}

3 references
private IEnumerator AdjustFOV(float targetFOV, float duration)
{
    isZooming = true; // Indicate that zooming is in progress
    float startFOV = _currentCamera.m_Lens.FieldOfView;
    float time = 0;

    while (time < duration)
    {
        time += Time.deltaTime;
        _currentCamera.m_Lens.FieldOfView = Mathf.Lerp(startFOV, targetFOV, time / duration);
        yield return null;
    }

    isZooming = false; // Zooming is complete
}
```

3. Combined Panning and Zooming

The method `StartPanAndZoom()` combines both camera panning and zooming, allowing them to occur simultaneously. Here's the logic flow:

- **Pan and Zoom Together:**
  - If there are any ongoing panning or zooming operations, they are stopped first to ensure smooth execution.
  - The method starts two coroutines simultaneously:
    - One for panning (`PanCamera()`).
    - One for zooming (`AdjustFOV()`).
  - Both operations run in parallel, giving the effect that the camera is smoothly moving and zooming in/out at the same time.

```
0 references
public void StartPanAndZoom(float panDistance, float panTime, PanDirection panDirection, bool panToStringPos, float zoomAmount, float zoomDuration, bool zoomIn)
{
    // Stop previous coroutines if they are running
    if (_panCameraCoroutine != null) StopCoroutine(_panCameraCoroutine);
    if (_zoomCoroutine != null) StopCoroutine(_zoomCoroutine);

    // Start panning and zooming simultaneously
    _panCameraCoroutine = StartCoroutine(PanCamera(panDistance, panTime, panDirection, panToStringPos));
    _zoomCoroutine = StartCoroutine(AdjustFOV(zoomIn ? originalFOV - zoomAmount : originalFOV + zoomAmount, zoomDuration));
}
```

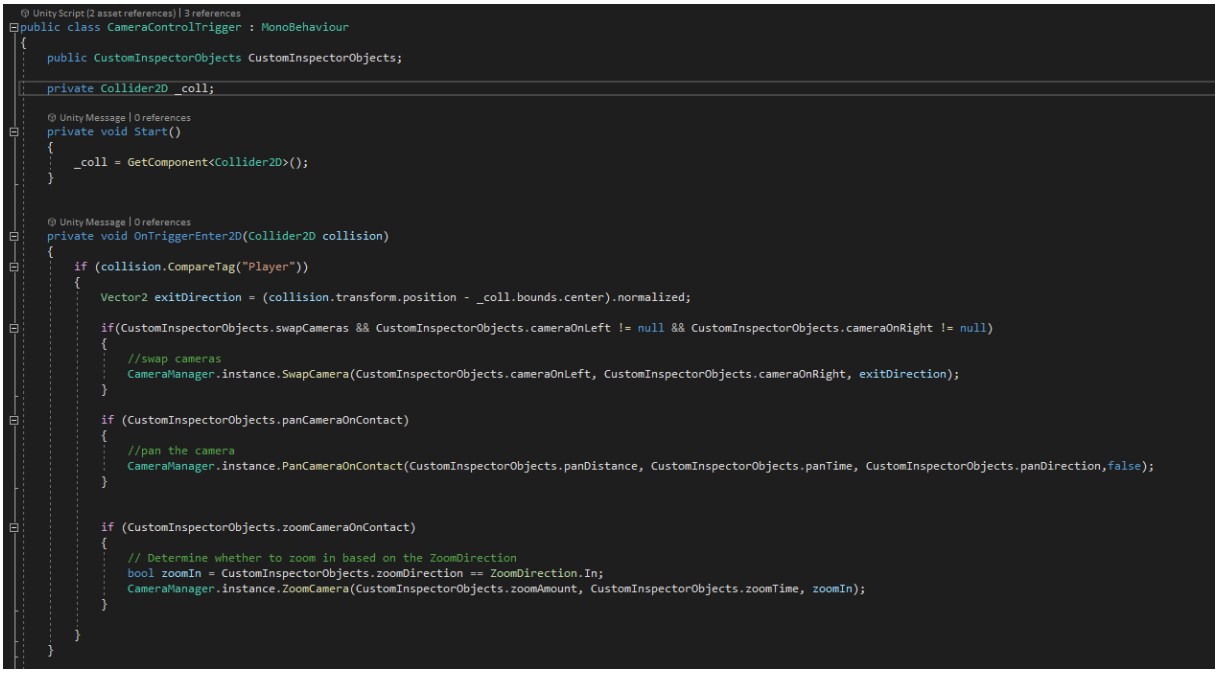
---

4. Key Takeaways:

- **Panning** adjusts the camera's tracked object offset, smoothly moving the camera along the X or Y axis over time.
- **Zooming** changes the camera's field of view (FOV), creating a zoom-in or zoom-out effect.
- **Linear interpolation (Lerp)** is used for both panning and zooming to ensure smooth transitions.
- **Coroutines** handle the timed nature of panning and zooming, allowing the operations to be spread out over multiple frames for fluid motion.

5. Camera Control Triggers

The `CameraControlTrigger` script handles camera panning and zooming when the player enters a specified trigger zone. When the player collides with the trigger (`OnTriggerEnter2D`), the script checks whether panning and zooming actions are enabled in the `CustomInspectorObjects` settings. If panning is enabled, it uses the `CameraManager` to smoothly move the camera in the specified direction over a set distance and time. Simultaneously, if zooming is enabled, the camera's field of view (FOV) is adjusted, either zooming in or out based on the configured zoom direction and amount. This allows for a seamless camera control experience as the player navigates the game world.



```
Unity Script (2 asset references) | 3 references
public class CameraControlTrigger : MonoBehaviour
{
    public CustomInspectorObjects CustomInspectorObjects;

    private Collider2D _coll;

    Unity Message | 0 references
    private void Start()
    {
        _coll = GetComponent<Collider2D>();
    }


    Unity Message | 0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            Vector2 exitDirection = (collision.transform.position - _coll.bounds.center).normalized;

            if(CustomInspectorObjects.swapCameras && CustomInspectorObjects.cameraOnLeft != null && CustomInspectorObjects.cameraOnRight != null)
            {
                //swap cameras
                CameraManager.instance.SwapCamera(CustomInspectorObjects.cameraOnLeft, CustomInspectorObjects.cameraOnRight, exitDirection);
            }

            if (CustomInspectorObjects.panCameraOnContact)
            {
                //pan the camera
                CameraManager.instance.PanCameraOnContact(CustomInspectorObjects.panDistance, CustomInspectorObjects.panTime, CustomInspectorObjects.panDirection,false);
            }


            if (CustomInspectorObjects.zoomCameraOnContact)
            {
                // Determine whether to zoom in based on the ZoomDirection
                bool zoomIn = CustomInspectorObjects.zoomDirection == ZoomDirection.In;
                CameraManager.instance.ZoomCamera(CustomInspectorObjects.zoomAmount, CustomInspectorObjects.zoomTime, zoomIn);
            }

        }
    }
```

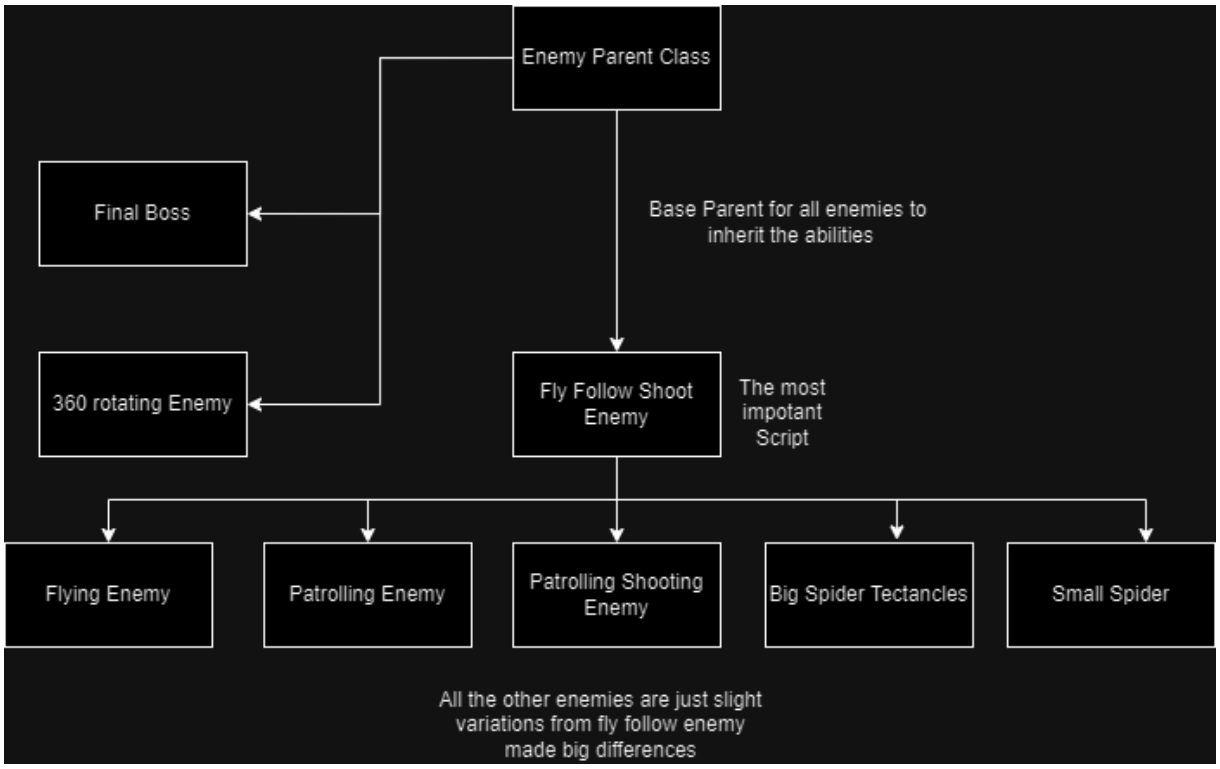
4.6 Enemies System

1. Core Concept

The Enemy System may have a lots of enemies but it is the most easiest , simplest, and reusable system in all 6 major functions. All Enemies comes from Parent Enemy Class. And so far, we have

1. Fly follow shoot enemy
2. Fly enemy
3. Walking enemy
4. Walking shooting enemy
5. Patrolling enemy
6. 360-degree rotating enemy
7. Spider Big
8. Spider Small
9. Traps

But They all are variation, of directions of travel, type of traveling (ground or air) and asset picture changes from Fly Follow Shoot Enemy. So in this chapter, we will only cover the first enemy, fly follow shoot enemy.



2. Core Fields and References

The `Enemy` class defines several key fields that are essential for enemy behavior. The `health` field stores the enemy's remaining health, which determines when the enemy is defeated. The `speed` field controls how fast the enemy can move, while the `damage` field specifies how much damage the enemy inflicts on the player upon contact. The `Rigidbody2D` component (`rb`) is used to manage the physics interactions of the enemy, allowing for movement and collision handling. Additionally, a reference to the `PlayerController` (`player`) enables the enemy to interact with the player, such as dealing damage or detecting the player's position.

```
[SerializeField] protected float health;
[SerializeField] protected float speed;
[SerializeField] public float damage;
public Rigidbody2D rb;
protected PlayerController player;
```

3. Enemy States Enum

The `EnemyStates` enumeration encapsulates the different states that the enemy can be in, such as `Idle`, `Crawler_Idle`, `Following`, and various states for different enemy types like `Bat` and `BossFight`. This structure allows for organized state management, enabling the enemy to

change behaviors based on its current state. For example, an enemy in the `Following` state would engage the player, while an enemy in `Idle` might remain stationary. The `currentEnemyState` variable tracks the enemy's current state, providing a basis for its behavior and animations.

```
protected enum EnemyStates
{
    Idle,
    Crawler_Idle,
    Crawler_Flip,
    Following,
    Bat_Idle,
    Bat_Chase,
    BossFight_Stage1,
    BossFight_Stage2,
    BossFight_Stage3,
    BossFight_Stage4
}

protected EnemyStates currentEnemyState = EnemyStates.Idle;
```

4. State Management Property

The property `GetCurrentEnemyState` serves as a mechanism to encapsulate the logic for state transitions within the `Enemy` class. When the current state is updated, it checks if the new state differs from the existing state. If it does, it triggers the `ChangeCurrentAnimation` method to adjust the enemy's animations accordingly. This encapsulation ensures that any state change is consistently handled, promoting cleaner code and easier maintenance. This design allows derived classes to override the behavior while still benefiting from the base functionality.

```
protected virtual EnemyStates GetCurrentEnemyState
{
    get { return currentEnemyState; }
    set
    {
        if (currentEnemyState != value)
        {
            currentEnemyState = value;
            ChangeCurrentAnimation();
        }
    }
}
```

5. Start Method

The `Start` method initializes important components of the enemy. It fetches and assigns references to the `Rigidbody2D`, which is crucial for physics interactions, and obtains a reference to the `PlayerController` instance, which enables the enemy to track and react to the player. This method also contains error handling to check if the player controller exists in the scene. By setting these foundational components in `Start`, the enemy is prepared for interactions and movements that occur later in the game loop.

```
protected virtual void Start()
{
    rb = GetComponent<Rigidbody2D>();
    player = PlayerController.Instance;
}
```

6. Update Method

The `Update` method is called once per frame and serves as the core loop for managing enemy behavior. It invokes `UpdateEnemyStates`, which can be overridden by derived classes to implement state-specific logic, and `CheckHealth`, which monitors the enemy's health. If the

```
protected virtual void Update()
{
    UpdateEnemyStates();
    CheckHealth();
}
```

enemy's health falls to zero or below, it triggers death logic. This method ensures that the enemy continually checks for necessary state updates and health management, maintaining responsiveness in gameplay.

7. Health Check and Death Handling

The `CheckHealth` method assesses whether the enemy's health has dropped to zero. If so, it sets the health to zero (to prevent negative health) and invokes the `OnEnemyDeath` event to notify any listeners (such as game managers) that the enemy has died. It then calls the `Death` method, which handles the destruction of the enemy game object after a specified delay. This structure allows for flexible death handling, such as triggering animations or effects upon death before removing the enemy from the scene.

```
protected virtual void CheckHealth()
{
    if (health <= 0)
    {
        health = 0;
        OnEnemyDeath?.Invoke(gameObject);
        Death(0f);
    }
}

protected virtual void Death(float destroyTime)
{
    Destroy(gameObject, destroyTime);
}
```

8. Damage Handling

The `EnemyHit` method is responsible for processing damage dealt to the enemy. It reduces the enemy's health based on the damage received, checks if the enemy is currently recoiling from a hit, and applies a force in the opposite direction of the hit to simulate a recoil effect. This method also integrates a screen shake effect for feedback during hits, enhancing the player's experience. This damage processing logic is critical for maintaining a dynamic and interactive combat system within the game.

```
public virtual void EnemyHit(float _damageDone, Vector2 _hitDirection, float _hitForce)
{
    health -= _damageDone;
    rb.AddForce(-_hitForce * _hitDirection); // Apply recoil
}
```

9. Collision Detection

The `OnCollisionStay2D` method detects collisions with other game objects, specifically targeting the player. If the enemy collides with the player, it retrieves the `PlayerController` component from the collided object and inflicts damage based on the enemy's defined damage value. This logic enables enemies to consistently harm the player during direct contact, supporting various enemy behaviors like melee attacks. The implementation ensures that the game responds correctly to player-enemy interactions.

```
protected virtual void OnCollisionStay2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Player"))
    {
        collision.gameObject.GetComponent<PlayerController>()?.TakeDamage(damage);
    }
}
```

10. State Update and Animation Change

The `UpdateEnemyStates` method serves as a placeholder for derived classes to implement their specific state management logic, enabling the enemy to behave differently based on its current state. The `ChangeCurrentAnimation` method is called whenever the state changes, allowing for seamless transitions between animations that reflect the enemy's actions. The `ChangeState` method updates the enemy's state while leveraging the `GetCurrentEnemyState` property to ensure that animation changes are properly triggered. This architecture provides flexibility for creating diverse enemy behaviors and animations while maintaining a consistent framework for state management.

```
protected virtual void UpdateEnemyStates() { }

protected virtual void ChangeCurrentAnimation() { }

protected void ChangeState(EnemyStates _newState)
{
    GetCurrentEnemyState = _newState; // Update enemy state
}
```

The `FollowingEnemy` function in the `FollowingShootingEnemy` class plays a pivotal role in defining the behavior of this specific enemy type, extending the capabilities provided by the

base `Enemy` class. Here's a detailed explanation of how the `FollowingEnemy` function works and how it contributes to creating a new type of enemy:

```
1 reference
private void FollowingEnemy()
{
    if (player == null) return; // If player is null, exit early

    float distanceFromPlayer = Vector2.Distance(player.position, enemyTransform.position);

    if (distanceFromPlayer < lineOfSite && distanceFromPlayer > shootingRange)
    {
        // Move towards the player
        enemyTransform.position = Vector2.MoveTowards(enemyTransform.position, player.position, moveSpeed * Time.deltaTime);

        // Determine the direction of movement
        //float xAxis = player.position.x - enemyTransform.position.x;
        //bool movingRight = xAxis > 0;

        // Flip the enemy object based on movement direction
        //FlipObject(movingRight);

        // Flip the enemy to always face the player
        FlipTowardsPlayer();
    }
    else if (distanceFromPlayer <= shootingRange && Time.time >= nextFireTime)
    {
        // Shoot at the player
        Instantiate(bullet, bulletPos.transform.position, Quaternion.identity);
        nextFireTime = Time.time + fireRate;

        // Ensure the enemy is facing the player when shooting
        FlipTowardsPlayer();
    }
}
```

1. **Distance Calculation**:
    - The function starts by calculating the distance between the enemy and the player using `Vector2.Distance(player.position, enemyTransform.position)`. This distance is crucial as it determines the enemy's actions: whether to move closer or shoot.
2. **Behavior Based on Distance**:
    - The enemy's behavior is dictated by its proximity to the player, which is defined by two thresholds: `lineOfSite` and `shootingRange`.
        - **Chasing the Player**: If the player is within the `lineOfSite` but outside the `shootingRange`, the enemy moves towards the player. This is done using `Vector2.MoveTowards`, which smoothly interpolates the enemy's position towards the player at a speed defined by `moveSpeed`.
        - **Shooting**: If the player is within `shootingRange` and the enemy is ready to fire (checked using `nextFireTime`), the enemy instantiates a bullet at its `bulletPos` location. After firing, it updates `nextFireTime` to regulate the firing rate based on `fireRate`.
3. **Orientation**:
    - The `FlipTowardsPlayer` method is called in both scenarios to ensure the enemy visually faces the player, enhancing the gameplay experience by making the enemy appear aware of the player's position.

11. Contribution to Enemy Class Structure

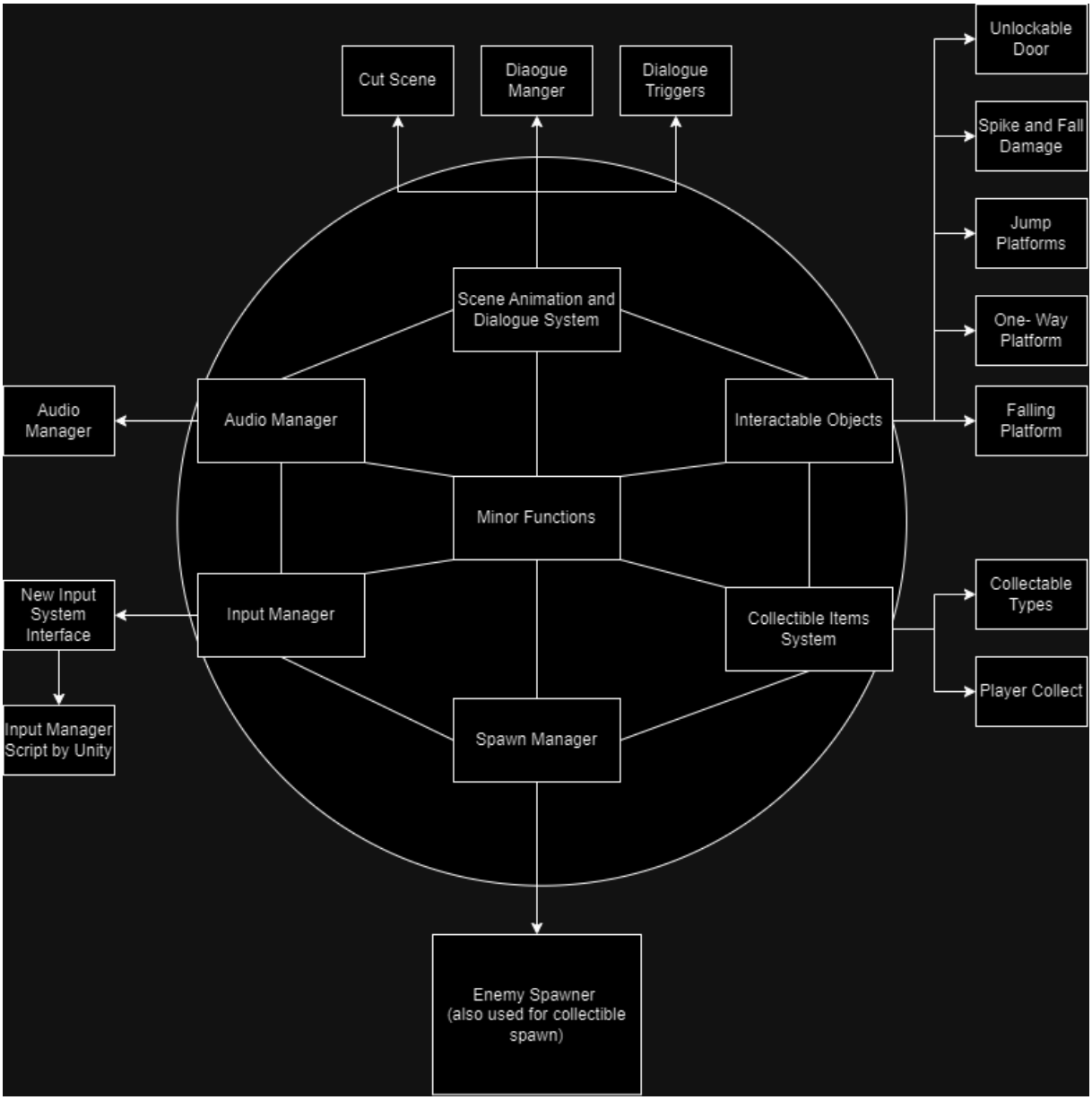
The `FollowingShootingEnemy` class derives from the `Enemy` base class, allowing it to inherit common enemy functionalities while adding specialized behavior. Here's how it establishes a foundation for other enemy types:

- **Base Functionality**: The `FollowingShootingEnemy` class utilizes the inherited methods from the `Enemy` class, such as `Start`, `Update`, and `EnemyHit`. This ensures that all enemies share fundamental characteristics, such as health management, destruction logic, and animation handling.
- **Specialized Behavior**: By implementing the `FollowingEnemy` function, this class introduces new behaviors specific to shooting and following mechanics. This means that other enemy types can inherit from this class if they require similar chasing and shooting capabilities, thereby promoting code reusability.
- **Modularity and Extensibility**: Other enemy classes can inherit from `FollowingShootingEnemy` to create variations with slight modifications. For example:
    - A **MeleeEnemy** could extend `FollowingShootingEnemy` and override the shooting logic to implement a melee attack when within range.
    - A **StealthEnemy** could inherit from `FollowingShootingEnemy`, adding logic to become invisible while moving towards the player or shooting only from cover.
- **Behavior Overrides**: Any derived enemy class can override the `FollowingEnemy` function to customize movement patterns, attack styles, or decision-making processes while still retaining the core enemy structure. This approach ensures that all enemies share essential traits while allowing for diverse gameplay experiences.

4.7 Boss Fight AI System

4.8 Minor Functions

1. Scene Animation and Dialogue System
2. Interactable Objects
3. Collectible Item System
4. Spawning Manager
5. Input Manager
6. Audio Manager



4.9 Scene Animation and Dialogue System

In interactive games, cutscenes play a crucial role in enhancing narrative delivery and player engagement. The cutscene system implemented in the project provides a method for integrating cinematic sequences that can convey important story elements, develop characters, and provide pivotal moments within the gameplay. This system consists of the **CutScene.cs** script, which manages the playback of cutscenes triggered by player interactions within the game environment.

The **Cut Scene** script is responsible for controlling the playback of cinematic sequences. It handles the activation of cutscenes when players enter designated areas and offers functionality to skip the cutscene if desired.

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

public class CutScene : MonoBehaviour
{
    [SerializeField] private PlayableDirector playableDirector;
    private bool isCutScenePlaying = false; // Track if the cutscene is playing

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            playableDirector.Play();
            isCutScenePlaying = true; // Mark that the cutscene is playing
            GetComponent<BoxCollider2D>().enabled = false; // Disable the collider to prevent retriggering
        }
    }

    private void Update()
    {
        // Check if the cutscene is playing and if the space bar is pressed
        if (isCutScenePlaying && Input.GetKeyDown(KeyCode.Space))
        {
            SkipCutscene(); // Skip the cutscene if space bar is pressed
        }
    }

    private void SkipCutscene()
    {
        // Skip to the end of the timeline and evaluate
        playableDirector.time = playableDirector.duration;
        playableDirector.Evaluate(); // Evaluate to skip to the end properly

        OnCutsceneEnd(); // Call function to handle end of the cutscene
    }

    private void OnCutsceneEnd()
    {
        // Mark the cutscene as finished
        isCutScenePlaying = false;

        // Resume player control or handle any other necessary logic here
        // For example, if you disabled player movement during the cutscene, re-enable it here.

        Debug.Log("Cutscene ended, player can move again.");
    }
}
```

Core Components:

Playable Director: **The Playable Director component is crucial for managing the playback of timelines created in Unity. This component allows for the execution of animations, audio, and other cinematic elements.**

isCutScenePlaying: This boolean flag tracks whether a cutscene is currently in progress. This is important for managing the player's input and ensuring the game behaves correctly during cinematic moments.

OnTriggerEnter2D(Collider2D collision):This method is triggered when the player enters a specific collider set up in the game world. It checks if the object colliding with the trigger, is the player.

If it is, the cutscene playback is initiated using playableDirector.Play(), and the isCutScenePlaying flag is set to true. This prevents retriggering the cutscene by disabling the collider.

Update():The Update method runs every frame and checks if a cutscene is currently playing. It listens to the space bar input, allowing players to skip the cutscene if they desire. This responsiveness improves player engagement by granting control over the pacing of narrative delivery.

SkipCutscene():This method allows players to skip the cutscene by moving to the end of the timeline. By setting playableDirector.time to the cutscene's total duration and calling playableDirector.Evaluate(), the cutscene effectively concludes, allowing the game to resume.

OnCutsceneEnd():This method is called when the cutscene finishes, either through normal progression or player input. It resets the isCutScenePlaying flag to false and handles any necessary gameplay logic, such as re-enabling player movement.

System Behavior:

Starting a Cutscene: When the player enters the trigger zone, the cutscene begins. The OnTriggerEnter2Dmethod initiates playback and ensures that the cutscene can only be triggered once by disabling the collider.

Ending a Cutscene: Once the cutscene is finished, the OnCutsceneEnd method is executed, resetting necessary flags and allowing the player to resume control. This ensures a smooth transition back to gameplay.

Player Interaction:

Skipping Cutscenes: Players can skip the cutscene at any time by pressing the space bar. This feature adds flexibility and allows players to control their experience, particularly useful for players who may wish to expedite the narrative.

Input Management: The system manages player input effectively during cutscenes, ensuring that actions such as movement are disabled while the cinematic is playing. This prevents unintended interactions that could disrupt the cutscene experience.

Summary

The cutscene system in the game provides a structured and immersive way to integrate cinematic sequences into gameplay. By utilizing the **Cut Scene** script, the system efficiently manages the triggering, playback, and skipping of cutscenes.

Dialogue System

In this game, dialogue systems are essential for conveying storylines and character interactions. The dialogue system implemented in the project provides a structured way to handle conversations between characters, presenting the dialogue with both visual and textual elements. This system consists of two main

scripts: **DialogueManager.cs** and **DialogueTrigger.cs**, working together to handle the initiation, display, and progression of dialogues.

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

public class DialogueManager : MonoBehaviour
{
    public Image ActorImage;
    public TextMeshProUGUI actorName;
    public TextMeshProUGUI messageText;
    public RectTransform backgroundBox;

    private Message[] currentMessages;
    private Actor[] currentActors;
    private int activeMessage = 0;
    public static bool isActive = false;

    public float typingSpeed = 0.05f; // Speed of typing effect in seconds per character
    private Coroutine typingCoroutine; // Store the reference to the typing coroutine

    // New variable to enable or disable skipping
    public bool canSkipDialogue = true;

    void Update()
    {
        // Check for input if skipping is allowed and dialogue is active
        if (canSkipDialogue && isActive && Input.anyKeyDown)
        {
            SkipCurrentMessage();
        }
    }

    public void OpenDialogue(Message[] messages, Actor[] actors)
    {
        currentMessages = messages;
        currentActors = actors;
        activeMessage = 0;
        isActive = true;
        Debug.Log("Started conversation! Loaded messages: " + messages.Length);
        backgroundBox.gameObject.SetActive(true); // Show the dialogue UI
        StartCoroutine(DisplayMessageWithTypingEffect());
    }
```

Dialogue Manager.cs

The DialogueManager script is responsible for managing how dialogue is displayed on the screen. It controls the flow of messages, the timing of when messages appear, and the visual elements related to the dialogue, such as character portraits and names.

Core Components:

UI Elements:

Actor Image (**ActorImage**)**:** This image displays the portrait of the character currently speaking. It updates with each new message to reflect the correct actor.

Actor Name (**actorName**)**:** Displays the name of the character currently speaking. This text changes dynamically as different actors speak during the dialogue.

Message Text (**messageText**)**:** This field contains the dialogue text itself, updating with each new message. It uses **TextMeshProUGUI** to render the text in a clean and flexible way.

Dialogue Flow**:** The dialogue consists of multiple messages stored in the **currentMessages** array. Each message contains the text the character will say and other relevant data, such as the duration for which the message should be displayed. The system keeps track of the current message using the **activeMessage** variable.

Typing **Effect**: To add realism, the system uses a typing effect where the dialogue appears letter by letter rather than all at once. This effect is managed by the **TypeMessage** coroutine. It gradually builds the message text over time by iterating over each character and adding it to the **messageText** field, with a small delay between **characters**. This delay is controlled by the **typingSpeed** variable, allowing developers to adjust the speed at which the text appears.

Progressing Through Messages: The function **Next Message** is responsible for moving from one message to the next. When the player presses any key or the message's display duration expires, this function is called, advancing the dialogue to the next message. If all messages have been displayed, the dialogue ends, and the dialogue UI is hidden. The ability to skip a message is provided through the **SkipCurrentMessage** function, which allows players to skip the typing effect and move directly to the next message. This gives players more control over the pacing of the dialogue.

```
44
45      IEnumerator DisplayMessageWithTypingEffect()
46      {
47          while (isActive)
48          {
49              DisplayMessage();
50              yield return new WaitForSeconds(currentMessages[activeMessage].duration);
51              NextMessage();
52          }
53      }
54
55      void DisplayMessage()
56      {
57          if (activeMessage < currentMessages.Length)
58          {
59              Message messageToDisplay = currentMessages[activeMessage];
60              Actor actorToDisplay = currentActors[messageToDisplay.actorId];
61              actorName.text = actorToDisplay.name;
62              ActorImage.sprite = actorToDisplay.sprite;
63
64              // Stop any previous typing coroutine before starting a new one
65              if (typingCoroutine != null)
66              {
67                  StopCoroutine(typingCoroutine);
68              }
69
70              typingCoroutine = StartCoroutine(TypeMessage(messageToDisplay.message));
71          }
72      }
73
74      IEnumerator TypeMessage(string message)
75      {
76          messageText.text = "";
77          foreach (char letter in message.ToCharArray())
78          {
79              messageText.text += letter;
80              yield return new WaitForSeconds(typingSpeed);
81          }
82      }
```

Message Duration: Each message has a specified duration, set in the Message class. This duration determines how long the message will remain on the screen before moving to the next one, unless the player chooses to skip it. The **DisplayMessageWithTypingEffect** coroutine controls this timing by waiting for the specified duration before calling **NextMessage.**

```
84      public void NextMessage()
85      {
86          activeMessage++;
87          if (activeMessage < currentMessages.Length)
88          {
89              StopAllCoroutines(); // Stop any running coroutines before starting a new message
90              StartCoroutine(DisplayMessageWithTypingEffect());
91          }
92          else
93          {
94              Debug.Log("Conversation ended");
95              isActive = false;
96              backgroundBox.gameObject.SetActive(false); // Hide the dialogue UI
97          }
98      }
99
100     // Updated function to skip the current message and go to the next one
101     public void SkipCurrentMessage()
102     {
103         // Stop the current typing coroutine
104         if (typingCoroutine != null)
105         {
106             StopCoroutine(typingCoroutine);
107         }
108
109         // Immediately jump to the next message
110         NextMessage();
111     }
112 }
```

System Behavior

Starting a Dialogue**:** The **OpenDialogue** function is called to start a dialogue session. This function takes two array messages (the lines of dialogue) and actors (the characters who are speaking). It initializes the system by setting the first message and making the dialogue UI visible. The DisplayMessageWithTypingEffect coroutine is then started to handle the display of each message.

Ending a Dialogue**:** Once all messages have been displayed, the **isActive** flag is set to false, and the UI is hidden. This ensures a smooth transition back to normal gameplay.

Player Interaction:

Skipping Messages**:** Players can skip messages by pressing any key, which is called the **SkipCurrentMessagefunction**. This interrupts the typing effect and jumps directly to the next message. This feature is especially useful for players who want to speed through dialogue they've already seen or prefer a faster pace.

Coroutine Control**:** The system makes heavy use of coroutines to manage the timing of the typing effect and message transitions. The **StopCoroutine** function is used to ensure that no two typing effects overlap, maintaining a smooth and error-free experience.

DialogueTrigger.cs

The **DialogueTrigger** script is responsible for starting dialogues in the game world. It allows specific objects (such as NPCs or interactive objects) to trigger dialogues when the player interacts with them.

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

public class DialougueTrigger : MonoBehaviour
{
    public Message[] messages;
    public Actor[] actors;

    public void StartDialogue()
    {
        FindObjectOfType<DialogueManager>().OpenDialogue(messages, actors);
    }
}

[System.Serializable]
public class Message
{
    public int actorId;
    public string message;
    public float duration; // Duration to display this message
}


[System.Serializable]
public class Actor
{
    public string name;
    public Sprite sprite;
}
```

Main Functions:

Message and Actor Arrays**:** DialogueTrigger stores arrays of Message and Actor objects. These arrays contain the lines of dialogue and the actors who speak to them. When a dialogue is triggered, these arrays are passed to the DialogueManager to begin the conversation.

Triggering the Dialogue**:** The **StartDialogue** function is called when a player interacts with an object that should start a dialogue. This function locates the DialogueManager in the scene and calls its OpenDialogue function, passing in the relevant messages and actors.

Message and Actor Classes

Message Class: Each Message object contains the dialogue text, the actor who will speak the message (**actorId**), and the duration for which the message should be displayed. This structure allows the dialogue to be tailored to different characters and situations.

Actor Class: The Actor class represents a character in the game, containing the actor's name and portrait (sprite). These are displayed alongside the dialogue text to provide a visual representation of who is speaking.

Summary

The dialogue system in the game provides a flexible, modular way to manage conversations between characters. By separating the dialogue logic into DialogueManager and DialogueTrigger, the system is easily reusable for different interactions throughout the game. The dialogue system is used for missions texts, tutorials, and character interactions.

4.10 Interactable objects

In this section, we focusing on how different elements such as falling platforms, jump platforms, and other interactive features enhance gameplay. These objects not only contribute to the game's dynamic and immersive environment but also add layers of strategy and challenge for the player. Below, we will examine the core mechanics of various interactable objects, detailing their behavior and purpose within the project.

1. Falling Platforms

Falling platforms are a crucial gameplay element designed to create temporary hazards for the player. These platforms initially remain stationary when the player steps on them, but after a short delay, they fall, forcing the player to move quickly to avoid falling with them.

- **Key Mechanism**: The platform waits for a set delay after the player lands on it before it starts to fall. The platform remains in place until the player steps on it, and then the platform falls at a predetermined speed. Once the platform falls, it can reset itself to its original position, making it available for reuse.
- **Behavior**:
    - The platform stays still when untouched.
    - It begins to fall a few moments after the player steps on it.
    - After falling, it resets itself when it reaches a specific reset zone, returning to its original position and ready for another fall.

This creates both a time-sensitive challenge for the player and adds dynamic level design where the environment is unpredictable.

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

public class FallingPlatform : MonoBehaviour
{
    [SerializeField] private float delayBeforeFall = 0.5f; // Time in seconds before the platform starts to fall
    [SerializeField] private float fallSpeed = 2f; // Speed of the fall

    private Rigidbody2D rb;
    private Vector3 originalPosition;
    private bool isFalling = false;

    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        originalPosition = transform.position; // Store the original position of the platform
        rb.isKinematic = true; // Initially, the platform doesn't fall
    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        // Check if the player has walked on the platform
        if (collision.gameObject.CompareTag("Player") && !isFalling)
        {
            StartCoroutine(FallAfterDelay());
        }
    }

    private IEnumerator FallAfterDelay()
    {
        isFalling = true;
        yield return new WaitForSeconds(delayBeforeFall);

        rb.isKinematic = false; // Platform starts falling
        rb.velocity = new Vector2(0, -fallSpeed);
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        // Check if the platform has collided with the designated trigger for resetting
        if (other.CompareTag("ResetTrigger") && isFalling)
        {
            ResetPlatform();
        }
    }

    private void ResetPlatform()
    {
        rb.velocity = Vector2.zero;
        rb.isKinematic = true;
        transform.position = originalPosition;
        isFalling = false; // The platform can now fall again if the player touches it
    }
}
```

2. Jump Platforms (Jump Pads)

Jump pads are interactable physics objects that propel the player upwards when they step on them. These elements are usually placed to help the player reach higher platforms or to navigate through tricky areas.

- **Key Mechanism**: Upon collision with the player, the jump pad applies an upward force, launching the player into the air. The strength of this jump can

be controlled by the force applied in the code, allowing designers to adjust the height of the player's jump.

- **Behavior**:
  - The player collides with the jump pad.
  - An upward force is applied, propelling the player into the air.
  - The jump force can be adjusted to fit specific game requirements or level designs.

This mechanic introduces vertical movement in the game, making the navigation of levels more dynamic. Players must learn to use jump pads strategically to progress through areas that cannot be reached by normal jumping alone.

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

public class JumpPads : MonoBehaviour
{
    [SerializeField] private float bounce = 20f;

    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            collision.gameObject.GetComponent<Rigidbody2D>().AddForce(Vector2.up * bounce,ForceMode2D.Impulse);
        }
    }
}
```

3. Spikes and Fall Damage

Spikes are a common obstacle in platformer games, causing damage or instant death if the player comes into contact with them. Fall damage, on the other hand, penalizes the player for falling from great heights, adding a level of risk to their movements.

- **Spikes**: The spikes are static hazards that, when touched by the player, will trigger a damage event. This could either reduce the player's health or cause instant death, depending on the game's design.
- **Fall Damage**: When the player falls from a significant height, they may incur damage or other penalties. This mechanic encourages players to plan their movements carefully to avoid taking unnecessary risks.

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

public class FallDamage : MonoBehaviour
{
    [SerializeField] private float damageAmount = 10f; // Amount of damage to deal on fall
    private SaveGroundCP safeGroundCheckPointSaver;


    private void Start()
    {
        safeGroundCheckPointSaver = GameObject.FindGameObjectWithTag("Player").GetComponent<SaveGroundCP>();
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            // Try to get the PlayerController component on the collided object
            PlayerController playerHealth = collision.GetComponent<PlayerController>();

            // If the component is found, call the TakeDamage method with isFallDamage set to true
            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damageAmount); // Specify true for isFallDamage parameter

                //warp the player to the saveground checkpoint
                safeGroundCheckPointSaver.WarpPlayerToSafeGround();
            }
        }
    }

}
```

4. One-Way Platforms

One-way platforms allow players to jump up through them but prevent them from falling back down. This mechanic adds a layer of complexity to movement, giving the player more options for vertical navigation.

- **Key Mechanism**: One-way platforms only block downward movement. The player can jump up through the platform from below but will be stopped by the platform when falling back down onto it.
- **Behavior**:
    - The player can jump from beneath and pass through the platform.
    - Once above the platform, the player cannot fall through it.

This mechanic allows for strategic placement of platforms, adding flexibility to level design and requiring the player to think carefully about how they navigate each section.

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

public class OneWayCollisionPlatform : MonoBehaviour
{
    private PlatformEffector2D effector2D;
    private float waitTime;


    // Start is called before the first frame update
    private void Start()
    {
        waitTime = 3f;
        effector2D = GetComponent<PlatformEffector2D>();
    }

    // Update is called once per frame
    private void Update()
    {
        OneWayCollisionController();
    }

    private void OneWayCollisionController()
    {
        if(PlayerController.Instance.pState.jumping == true)
        {
            effector2D.rotationalOffset = 0f;
        }

        waitTime -= Time.deltaTime;
        if (waitTime < 0f)
        {
            effector2D.rotationalOffset = 180f;
            waitTime = 3f;
        }
    }
}
```

5. Unlockable Doors

Unlockable doors are used to restrict access to certain areas until the player completes specific tasks, such as collecting a key or solving a puzzle. Once the condition is met, the door opens, allowing the player to proceed.

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

public class DoorUnlocker : MonoBehaviour
{
    public CollectibleType requiredItemType;
    public int requiredItemCount = 1;
    public float moveUpDistance = 5f; // Distance to move up
    public float moveSpeed = 2f; // Speed of the move
    [SerializeField] private AudioSource unlockSound;
    public GameObject requirementPanel; // Assign this in the Inspector
    public TextMeshProUGUI messageUI; // Use Text if you're not using TextMeshPro
    [SerializeField] private GameObject collectiblePrefab; // The prefab of the collectible to spawn
    [SerializeField] private Transform[] spawnPoints; // Array of spawn points for the collectibles
    private bool hasSpawnedCollectibles = false; // To ensure collectibles are spawned only once
    private bool isDoorUnlocked = false; // Add this line to declare the variable
    private bool isUnlocking = false; // Indicates that the unlocking process has started

```

- **Key Mechanism**: The door remains locked until a specific event is triggered, such as the player collecting a key or activating a switch. Once the condition is satisfied, the door unlocks and opens, allowing the player to pass through.
- **Behavior**:
    - Initially, the door blocks the player from progressing.
    - Once the player completes the required task (e.g., finding a key), the door unlocks.
    - The door can either remain open permanently or close after a set time or event.

```
38      private void OnTriggerEnter2D(Collider2D collision)
39      {
40          if (collision.CompareTag("Player"))
41          {
42              PlayerCollect playerCollect = collision.gameObject.GetComponent<PlayerCollect>();
43              if (playerCollect != null)
44              {
45                  // Check if the door is already unlocked
46                  if (isDoorUnlocked || isUnlocking)
47                  {
48                      // If the door is unlocked, do nothing or perform another action
49                      return; // Exit the method early
50                  }
51
52                  if (playerCollect.HasItems(requiredItemType, requiredItemCount))
53                  {
54                      playerCollect.UseItems(requiredItemType, requiredItemCount);
55                      StartCoroutine(UnlockDoor());
56                      isUnlocking = true; // Add this line to indicate that unlocking has started
57                  }
58                  else
59                  {
60                      int currentItemCount = playerCollect.GetItemCount(requiredItemType);
61                      if (currentItemCount == 0 && !hasSpawnedCollectibles)
62                      {
63                          SpawnCollectibles();
64                          hasSpawnedCollectibles = true;
65                      }
66
67                      int itemsNeeded = requiredItemCount - currentItemCount;
68                      messageUI.text = currentItemCount == 0 ?
69                          $"You need {requiredItemCount} {requiredItemType}s to unlock this door." :
70                          $"You have {currentItemCount}. You need {itemsNeeded} more {requiredItemType}s to unlock this door.";
71                      requirementPanel.SetActive(true);
72                      StartCoroutine(HideMessageAfterDelay(3f));
73                  }
74              }
75          }
76      }
```

Unlockable doors are essential for controlling the player's progress through levels, ensuring that they complete specific tasks before advancing.

Conclusion

Interactable physics objects such as falling platforms, spikes, jump pads, one-way platforms, and unlockable doors are integral to creating an engaging and challenging gaming experience. Each of these objects introduces new mechanics that the player must learn to navigate, adding depth to both the gameplay and level design. Through careful implementation and fine-tuning, these elements contribute to a more immersive and rewarding experience, keeping players engaged while providing a dynamic environment that reacts to their actions.

By leveraging these interactable elements, our project enhances its overall complexity, requiring the player to think quickly, move strategically, and use the environment to their advantage.

4.11 Collectible Items System

1. Using Directives and Enum Definition

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro; // Make sure to use TextMeshPro

// Define an enum for collectible types
public enum CollectibleType
{
    Gem,
    Jade,
    Stone,
    Diamond
}
```

Explanation

- **Using Directives**: The `using` statements at the top include necessary namespaces:
  - `System.Collections` and `System.Collections.Generic` are included for handling collections like lists or arrays if needed.
  - `UnityEngine` is the core namespace for Unity functionalities.
  - `TMPro` is included for using TextMesh Pro, which allows for advanced text rendering in Unity.
- **Enum Definition**: The `CollectibleType` enum defines the different types of collectibles that can be collected in the game (Gem, Jade, Stone, Diamond). This enum allows for clear and type-safe referencing of collectible types throughout the code.

2. Class and Serialized Fields

```
public class PlayerCollect : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI gemCountText;
    [SerializeField] private TextMeshProUGUI keyCountText;
    [SerializeField] private TextMeshProUGUI stoneCountText;
    [SerializeField] private TextMeshProUGUI diamondCountText;
    [SerializeField] private AudioSource collectSound;

    private int gemCount = 0;
    private int jadeCount = 0;
    private int stoneCount = 0;
    private int diamondCount = 0;
```

Explanation

- **Class Definition**: The `PlayerCollect` class derives from `MonoBehaviour`, making it a Unity script that can be attached to game objects.
- **Serialized Fields**:
  - The `TextMeshProUGUI` variables are used to reference UI text elements in the game that display the count of each collectible type. These are exposed to the Unity Inspector because of the `[SerializeField]` attribute, allowing developers to assign the UI elements directly from the Unity Editor.
  - `AudioSource collectSound` is used to play a sound effect when a collectible is gathered, enhancing the player's feedback.
- **Count Variables**: Four integer variables are defined to keep track of the number of each collectible type that the player has collected. They are initialized to zero.

3. Trigger Event Handling

```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("CollectibleItem"))
    {
        // Assume the collectible item has a component that specifies its type
        CollectibleItem collectibleItem = collision.gameObject.GetComponent<CollectibleItem>();
        if (collectibleItem != null)
        {
            switch (collectibleItem.itemType)
            {
                case CollectibleType.Gem:
                    gemCount++;
                    gemCountText.text = gemCount.ToString();
                    break;
                case CollectibleType.Jade:
                    jadeCount++;
                    keyCountText.text = jadeCount.ToString();
                    break;
                case CollectibleType.Stone:
                    stoneCount++;
                    stoneCountText.text = stoneCount.ToString();
                    break;
                case CollectibleType.Diamond:
                    diamondCount++;
                    diamondCountText.text = diamondCount.ToString();
                    break;
            }

            collectSound.Play();
            Destroy(collision.gameObject); // Destroy the collectible item after collecting
        }
    }
}
```

Explanation

- **Trigger Detection**: The `OnTriggerEnter2D` method is automatically called when the player enters a collider marked as a trigger. The `collision` parameter contains information about the collider that was entered.
- **Collectible Detection**: The script checks if the collided object has the tag `CollectibleItem`. If it does, the script attempts to get the `CollectibleItem` component, which presumably contains the collectible's type.
- **Switch Case for Counting**: The switch statement evaluates the `itemType` of the collectible:
    - For each collectible type (Gem, Jade, Stone, Diamond), the respective count variable is incremented, and the corresponding UI text element is updated to reflect the new count.
- **Feedback and Destruction**: After collecting the item:
    - The `collectSound` is played to provide audio feedback to the player.
    - The collected item is destroyed using `Destroy(collision.gameObject)`, removing it from the game.

4. Checking Item Counts

```
public bool HasItems(CollectibleType itemType, int count)
{
    switch (itemType)
    {
        case CollectibleType.Gem:
            return gemCount >= count;
        case CollectibleType.Jade:
            return jadeCount >= count;
        case CollectibleType.Stone:
            return stoneCount >= count;
        case CollectibleType.Diamond:
            return diamondCount >= count;
        default:
            return false;
    }
}
```

Explanation

- **Item Availability Check**: The `HasItems` method allows other parts of the code to check whether the player has a certain number of a specific collectible type.
- **Switch Case Logic**: The switch statement checks the `itemType` and compares the corresponding count variable against the requested `count`. It returns `true` if the player has enough items; otherwise, it returns `false`.

5. Using Items

```
public void UseItems(CollectibleType itemType, int count)
{
    switch (itemType)
    {
        case CollectibleType.Gem:
            gemCount = Mathf.Max(0, gemCount - count);
            gemCountText.text = gemCount.ToString();
            break;
        case CollectibleType.Jade:
            jadeCount = Mathf.Max(0, jadeCount - count);
            keyCountText.text = jadeCount.ToString();
            break;
        case CollectibleType.Stone:
            stoneCount = Mathf.Max(0, stoneCount - count);
            stoneCountText.text = stoneCount.ToString();
            break;
        case CollectibleType.Diamond:
            diamondCount = Mathf.Max(0, diamondCount - count);
            diamondCountText.text = diamondCount.ToString();
            break;
    }
}
```

Explanation

- **Item Usage Logic**: The `UseItems` method allows other parts of the code to consume or use a specific number of a given collectible type. This is useful for crafting, opening doors, or other interactions.
- **Switch Case for Usage**: The switch statement checks the `itemType` and decreases the corresponding count by the specified `count`. The `Mathf.Max(0, ...)` ensures that the count never goes below zero.
- **UI Update**: After modifying the count, the corresponding UI text element is updated to reflect the new count.

6. Getting Item Count

```
public int GetItemCount(CollectibleType itemType)
{
    switch (itemType)
    {
        case CollectibleType.Gem:
            return gemCount;
        case CollectibleType.Jade:
            return jadeCount;
        case CollectibleType.Stone:
            return stoneCount;
        case CollectibleType.Diamond:
            return diamondCount;
        default:
            Debug.LogError("Unknown item type.");
            return 0;
    }
}
```

Explanation

- **Retrieve Item Count**: The `GetItemCount` method allows other parts of the code to query the number of a specific collectible type the player currently has.
- **Switch Case for Retrieval**: The switch statement checks the `itemType` and returns the corresponding count variable. If an unknown item type is passed, it logs an error and returns zero.

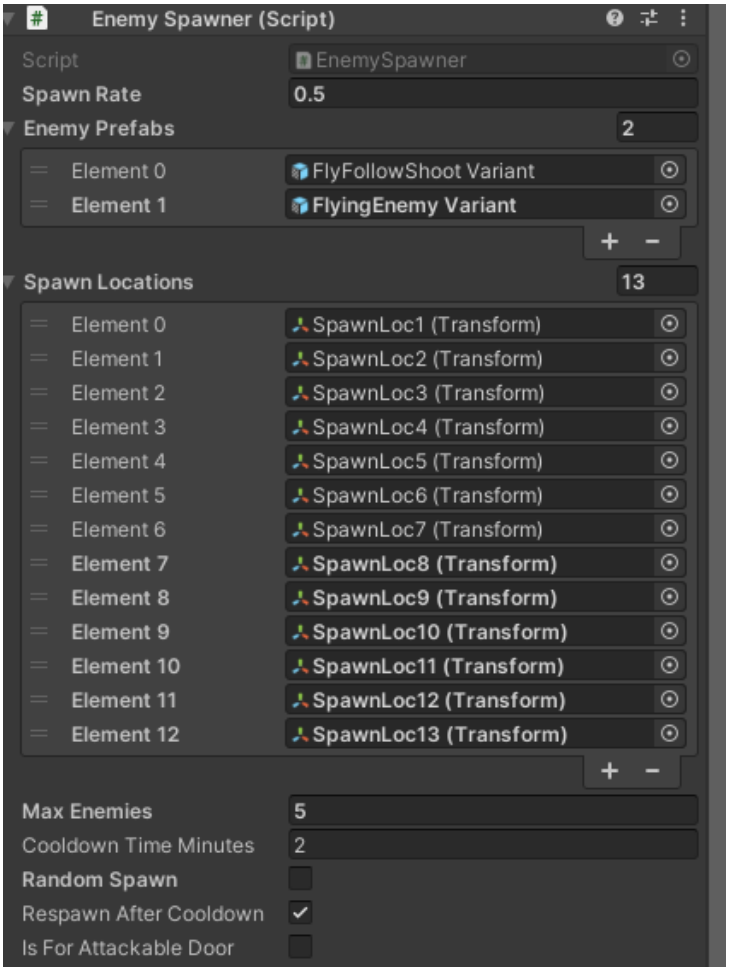
Conclusion

The `PlayerCollect` class effectively manages the collection of various collectible items in the game. By using trigger detection, it responds to player interactions with collectible items, updates item counts, and provides feedback through UI updates and sound effects. The methods for checking, using, and retrieving item counts enable other game systems to interact with the player's inventory.

4.12 Spawning Manager

Spawning system is a crucial system that works with enemy and items to spawn for player to achieve missions across the game system. The well built multi- system can be used for not

only enemies and collectible items, but also any other objects developers want to spawn. This system works together with enemy system, collectible items system and locked doors.



1. Spawn Rate is the spawn speed of the enemy.
2. Unlimited amount of spawn location can be added in the Spawn Locations Array.
3. Max number of enemies to spawn in those locations.
4. After maximum numbers spawned, we will cool down the system for as long as we want and then there will be two choices.
5. We can respawn the enemies as in “Wave Attack” or we can use it as one-time spawn.
6. Random Spawn is used when we want to spawn random in those array locations as it is less predictable.
7. Is For Attackable Door attribute is used for two choice or door missions. We want the door to open after we killed all enemies which works with one-time spawn. We want to open the door ourselves little by little while the enemies keep spawning until we finally opened the door.

Part 1: Class Declaration and Variables

```
public class EnemySpawner : MonoBehaviour
{
    public static EnemySpawner Instance { get; private set; }

    [SerializeField] private float spawnRate = 1f;
    [SerializeField] private GameObject[] enemyPrefabs;
    [SerializeField] private Transform[] spawnLocations;
    [SerializeField] private int maxEnemies = 10;
    [SerializeField] private float cooldownTimeMinutes = 5f;
    [SerializeField] private bool randomSpawn = true;
    [SerializeField] private bool respawnAfterCooldown = true;
    [SerializeField] private bool isForAttackableDoor = false;

    private bool canSpawn = false;
    private int currentEnemyCount = 0;
    private bool cooldownActive = false;
    private int currentSpawnIndex = 0;
    private float CooldownTimeSeconds => cooldownTimeMinutes * 60f;
    private bool doorOpened = false; // New variable to track if the door has been opened

    private List<GameObject> spawnedEnemies = new List<GameObject>();
    public event System.Action OnAllEnemiesDefeated;
}
```

Explanation:

- **Singleton Pattern**: `public static EnemySpawner Instance { get; private set; }` allows for a single instance of the `EnemySpawner` class to be accessed throughout the game.
- **Serialized Fields**: Attributes like `[SerializeField]` expose private variables in the Unity Inspector. This includes:
    - `spawnRate`: The time interval (in seconds) between enemy spawns.
    - `enemyPrefabs`: An array of enemy prefab GameObjects that can be spawned.
    - `spawnLocations`: An array of possible locations where enemies can spawn.
    - `maxEnemies`: The maximum number of enemies that can exist simultaneously.
    - `cooldownTimeMinutes`: Time (in minutes) to wait before respawning after all enemies are defeated.
    - `randomSpawn`: A boolean to decide if enemies should spawn randomly.
    - `respawnAfterCooldown`: Indicates whether to respawn enemies after the cooldown period.
    - `isForAttackableDoor`: A flag to determine if this spawner is for an attackable door.
- **Instance Variables**:
    - `canSpawn`: Tracks if the spawner is currently able to spawn enemies.

- `currentEnemyCount`: The number of enemies currently spawned.
- `cooldownActive`: Indicates if a cooldown period is currently active.
- `currentSpawnIndex`: Index for the next spawn location.
- `CooldownTimeSeconds`: A calculated property to convert minutes to seconds.
- `doorOpened`: Tracks if the door has been opened.
- `spawnedEnemies`: A list to keep track of spawned enemies.
- `OnAllEnemiesDefeated`: An event to notify when all enemies are defeated.

Part 2: Trigger Methods

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (doorOpened) return; // Prevent spawning if the door is already opened

    if (other.CompareTag("Player") && !cooldownActive && currentEnemyCount == 0)
    {
        canSpawn = true;
        StartCoroutine(Spawner());
    }
}

private void OnTriggerExit2D(Collider2D other)
{
    if (other.CompareTag("Player"))
    {
        canSpawn = false;
        StopCoroutine(Spawner());
    }
}
```

Explanation:

- **OnTriggerEnter2D**:
  - This method is called when another collider enters the trigger collider attached to the spawner object.
  - It checks if the door is already opened. If it is, it exits early.
  - If the player enters the trigger and no enemies are present, it starts the spawning process by setting `canSpawn` to true and starting the `Spawner` coroutine.
- **OnTriggerExit2D**:
  - This method is called when another collider exits the trigger collider.
  - If the exiting collider is the player, it sets `canSpawn` to false and stops the `Spawner` coroutine, halting enemy spawning.

Part 3: Door Management Methods

```
public void DoorOpened()
{
    doorOpened = true; // Set door as opened
    StopSpawning();    // Stop spawning new enemies
    IsNotTrigger();    // Disable spawner trigger functionality
    DestroySpawner();  // Destroy the spawner game object to prevent further spawning
}

// Define a new method to destroy the spawner game object
private void DestroySpawner()
{
    Destroy(gameObject); // Destroy the spawner game object
}

// Define the IsNotTrigger method
public void IsNotTrigger()
{
    StopSpawning();
    canSpawn = false;  // Ensure spawner no longer triggers
}
```

Explanation:

- **DoorOpened**:
  - This method is called when the door is opened, marking the door as opened, stopping enemy spawning, and disabling the spawner's trigger functionality. Finally, it destroys the spawner object to prevent any further spawning.
- **DestroySpawner**:
  - This method simply destroys the spawner GameObject. It can be called from other methods, such as when the door opens.
- **IsNotTrigger**:
  - This method stops enemy spawning and ensures that the spawner does not trigger anymore.

Part 4: The Spawner Coroutine

```
3 references
private IEnumerator Spawner()
{
    WaitForSeconds wait = new WaitForSeconds(spawnRate);

    while (canSpawn && (isForAttackableDoor || currentEnemyCount < maxEnemies))
    {
        if (currentEnemyCount >= maxEnemies && !isForAttackableDoor)
        {
            yield break;
        }

        yield return wait;

        int randPrefabIndex = randomSpawn ? UnityEngine.Random.Range(0, enemyPrefabs.Length) : 0;
        GameObject enemyToSpawn = enemyPrefabs[randPrefabIndex];
        Transform spawnLocation = GetNextSpawnLocation();
        GameObject spawnedEnemy = Instantiate(enemyToSpawn, spawnLocation.position, Quaternion.identity);
        spawnedEnemies.Add(spawnedEnemy);
        spawnedEnemy.GetComponent<Enemy>().OnEnemyDeath += OnEnemyDeath;
        currentEnemyCount++;
    }

    if (isForAttackableDoor && respawnAfterCooldown && !doorOpened) // Ensure cooldown only happens if the door isn't opened
    {
        cooldownActive = true;
        yield return new WaitForSeconds(CooldownTimeSeconds);
        cooldownActive = false;
        StartCoroutine(Spawner());
    }
}
```

Explanation:

- **Coroutine Logic**:
  - This coroutine manages the enemy spawning process.
  - It waits for the specified `spawnRate` before attempting to spawn a new enemy.
  - The while loop checks two conditions: `canSpawn` must be true, and either `isForAttackableDoor` is true or the current enemy count is less than the maximum allowed. If the maximum number of enemies is reached and it's not for an attackable door, it exits the coroutine.
- **Random Enemy Selection**:
  - If `randomSpawn` is enabled, it randomly selects an enemy prefab from the `enemyPrefabs` array.
- **Spawning Logic**:
  - It gets the next spawn location using `GetNextSpawnLocation()`, instantiates the enemy prefab, adds it to the `spawnedEnemies` list, and subscribes to the `OnEnemyDeath` event.
- **Cooldown Logic**:
  - If the spawner is for an attackable door and respawning after cooldown is allowed, it activates the cooldown, waits for the cooldown period, then restarts the spawning process.

Part 5: Get Next Spawn Location

```
private Transform GetNextSpawnLocation()
{
    Transform spawnLocation = spawnLocations[currentSpawnIndex];
    currentSpawnIndex = (currentSpawnIndex + 1) % spawnLocations.Length;
    return spawnLocation;
}
```

Explanation:

- **GetNextSpawnLocation**:
    - This method retrieves the next spawn location in a circular manner. After returning a spawn location, it increments the `currentSpawnIndex`, wrapping around to the start of the array if necessary.

Part 6: Enemy Death Handling

```
private void OnEnemyDeath(GameObject enemy)
{
    spawnedEnemies.Remove(enemy);
    currentEnemyCount--;

    if (spawnedEnemies.Count == 0 && !isForAttackableDoor)
    {
        OnAllEnemiesDefeated?.Invoke();  // Notify that all enemies are defeated
        DoorOpened();  // Call DoorOpened to stop spawning
    }
}
```

Explanation:

- **OnEnemyDeath**:
    - This method is called when an enemy dies. It removes the enemy from the `spawnedEnemies` list and decrements the `currentEnemyCount`.
    - If all enemies have been defeated and it's not for an attackable door, it invokes the `OnAllEnemiesDefeated` event and calls `DoorOpened()` to indicate the door can now be opened.

Part 7: Stop Spawning

```
public void StopSpawning()
{
    canSpawn = false;
    StopAllCoroutines();
}
```

Explanation:

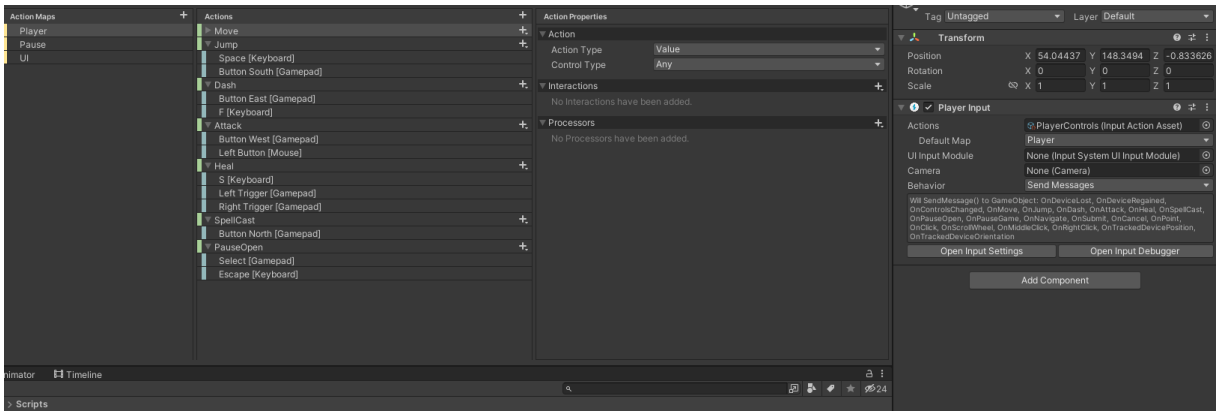
- **StopSpawning**:
  - This method sets `canSpawn` to false, effectively halting any further enemy spawning, and stops all coroutines running on this GameObject, including the `Spawner()` coroutine.

Summary

The `EnemySpawner` class controls the spawning of enemies in a game, allowing for flexible spawning mechanics such as cooldowns, max limits, and random selection of enemy types. It tracks whether the door has been opened and manages events related to enemy defeat. The use of coroutines allows for efficient waiting and timing management.

4.13 Input Manager

The Input Manager is all about setting up Unity new input system interface and connecting to the object that accepts input like Player and Menu System.



We use Actions Map to connect to the keys of the laptop , pc and external controllers. In Figure below, you can see the control keys assigned for player actions and pause menu.

This code is an example of how to handle player inputs using Unity's **new Input System**. The new Input System replaces the legacy input system and allows developers to create more robust and customizable control schemes. Here's a breakdown of the code and its usage:

```
Unity Message | 0 references
void OnEnable()
{
    SceneManager.sceneLoaded += OnSceneLoaded;
    playerControls.Enable();
    playerControls.Player.Jump.performed += OnJumpPerformed;
    playerControls.Player.Dash.performed += OnDashPerformed;
    playerControls.Player.Attack.performed += OnAttackPerformed;
    playerControls.Player.Heal.performed += OnHealPerformed;
    playerControls.Player.Heal.canceled += OnHealCanceled;
    //playerControls.Player.SpellCast.performed += OnSpellCastPerformed;
    playerControls.Enable();
}

Unity Message | 0 references
void OnDisable()
{
    SceneManager.sceneLoaded -= OnSceneLoaded;
    playerControls.Disable();
    playerControls.Player.Jump.performed -= OnJumpPerformed;
    playerControls.Player.Dash.performed -= OnDashPerformed;
    playerControls.Player.Attack.performed -= OnAttackPerformed;
    playerControls.Player.Heal.performed -= OnHealPerformed;
    playerControls.Player.Heal.canceled -= OnHealCanceled;
    //playerControls.Player.SpellCast.performed -= OnSpellCastPerformed;
    playerControls.Disable();

}
// Callbacks from the new input system
2 references
private void OnHealPerformed(InputAction.CallbackContext context)
{
    isHealing = true; // Start healing
}

2 references
private void OnHealCanceled(InputAction.CallbackContext context)
{
    isHealing = false; // Stop healing
}

2 references
private void OnAttackPerformed(InputAction.CallbackContext context)
{
    // Perform the attack logic
    Attack();
    // Play attack sound effect
    //attackSoundEffect.Play();
}
```

Key Components

1. **Player Input Handling**
2. **Event Registration and Deregistration**
3. **Input Callbacks and Actions**

1. Handling Input Events When the Object is Active

```
void OnEnable()
{
    SceneManager.sceneLoaded += OnSceneLoaded;
    playerControls.Enable();
    playerControls.Player.Jump.performed += OnJumpPerformed;
    playerControls.Player.Dash.performed += OnDashPerformed;
    playerControls.Player.Attack.performed += OnAttackPerformed;
    playerControls.Player.Heal.performed += OnHealPerformed;
    playerControls.Player.Heal.canceled += OnHealCanceled;
    playerControls.Enable();
}
```

In the first function that runs when the script becomes active:

- It sets up a listener so that whenever a new scene is loaded, a special function will be triggered to handle necessary setup tasks.
- It activates the player's input controls, making them responsive to player actions.
- Then, it attaches specific functions to the player's different actions, such as jumping, dashing, attacking, and healing. Each action is linked to its respective function so that when a player performs an action (like pressing a button), the game will respond appropriately.
- It also allows for an additional setup where healing starts when a button is pressed and stops when the button is released.

2. Cleaning Up Input Events When the Object is Disabled

```
void OnDisable()
{
    SceneManager.sceneLoaded -= OnSceneLoaded;
    playerControls.Disable();
    playerControls.Player.Jump.performed -= OnJumpPerformed;
    playerControls.Player.Dash.performed -= OnDashPerformed;
    playerControls.Player.Attack.performed -= OnAttackPerformed;
    playerControls.Player.Heal.performed -= OnHealPerformed;
    playerControls.Player.Heal.canceled -= OnHealCanceled;
    playerControls.Disable();
}
```

In the second function, which runs when the script becomes inactive or the object is destroyed:

- It removes the listener for when a new scene loads, as the object doesn't need to respond to it anymore.
- It disables the player's input controls so that no more input actions will be registered.
- Just like before, it disconnects all the specific actions (jumping, dashing, attacking, healing, etc.) from their respective functions. This prevents actions from triggering functions that are no longer relevant.

```
private void OnHealPerformed(InputAction.CallbackContext context)
{
    isHealing = true; // Start healing
}

private void OnHealCanceled(InputAction.CallbackContext context)
{
    isHealing = false; // Stop healing
}
```

```
private void OnAttackPerformed(InputAction.CallbackContext context)
{
    // Perform the attack logic
    Attack();
    // Play attack sound effect
    //attackSoundEffect.Play();
}
```

3. Explanation of Unity's Input System Usage

This code demonstrates a basic usage of Unity's new Input System, which involves:

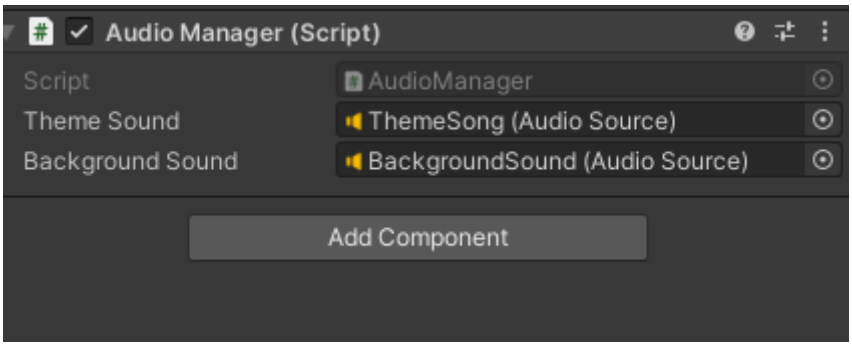
- **Input Action Maps**: The `playerControls` object is an instance of a generated class from the Input System's action map. This class organizes different actions (e.g., Jump, Dash, Attack, Heal) into one cohesive input scheme.
- **Input Action Phases**: Actions have different phases, such as `.performed` (when an input happens) and `.canceled` (when an input ends). This allows for continuous actions like holding down a heal button or performing quick actions like a dash or jump.
- **Input Event Registration**: The code listens for input events in `OnEnable()` by subscribing to specific action events. It ensures those event listeners are removed in

`OnDisable()` to avoid any lingering input actions when the player object or script is disabled.

Summary

- **Flexible Input Handling**: By using Unity's Input System, you can create modular, reusable input logic with phase-based callbacks.
- **State Management**: The code effectively handles stateful input actions, such as starting and stopping healing.
- **Automatic Input Configuration**: The Input System allows you to manage inputs across different devices (keyboard, controller, etc.) without changing the core logic.
- **Future Key Binding Development**: In Some games, they allow users to set control keys and change them. For example, originally, the attack key may be key “A” but user can change to key “B”. This approach is very important for this step if we want to allow our users to customize keys.

4.14 Audio Manager



Class Declaration and Variables

```
public class AudioManager : MonoBehaviour
{
    public AudioSource themeSound;
    public AudioSource backgroundSound;
```

- **Class Definition**: The class `AudioManager` inherits from `MonoBehaviour`, which allows it to be attached to a GameObject in Unity and take advantage of Unity's event functions (like `Start`, `Update`, etc.).
- **Public Variables**:
  - `themeSound`: This `AudioSource` is intended for the main theme music or sound of the game. It's public, which means it can be assigned through the Unity Inspector.
  - `backgroundSound`: Another `AudioSource` for ambient or background sounds. It is also public for Inspector assignment.

Start Method

```
void Start()
{
    // Play theme sound loop
    themeSound.loop = true;
    themeSound.Play();

    // Play background sound loop
    backgroundSound.loop = true;
    backgroundSound.Play();
}
```

- **Start Method**: This method is called when the script instance is being loaded. It's a good place to initialize values or start processes.
- **Looping Audio**:
  - `themeSound.loop = true;` sets the theme sound to loop continuously.
  - `themeSound.Play();` starts playing the theme sound.
  - Similarly, `backgroundSound.loop = true;` and `backgroundSound.Play();` ensure that the background sound also plays in a loop.

Public Methods for Audio Control

```
public void DisableThemeSound()
{
    themeSound.Pause();
}

public void EnableThemeSound()
{
    themeSound.UnPause();
}
```

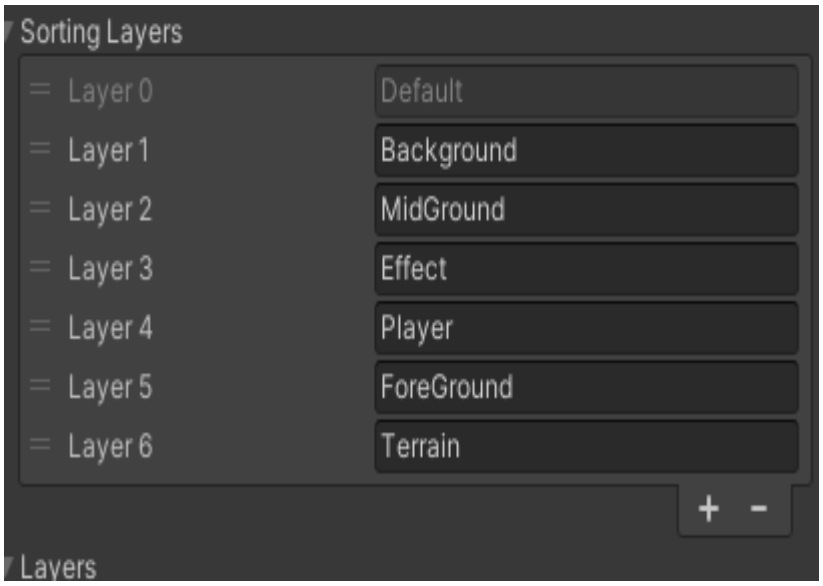
- **DisableThemeSound**: This method pauses the playback of the `themeSound` audio. It allows the game to temporarily stop the theme sound without losing its playback position.
- **EnableThemeSound**: This method resumes playback of the `themeSound` audio from where it was paused. This is useful for implementing game states where the theme sound should be temporarily silenced (like in menus or during certain gameplay events).

Summary

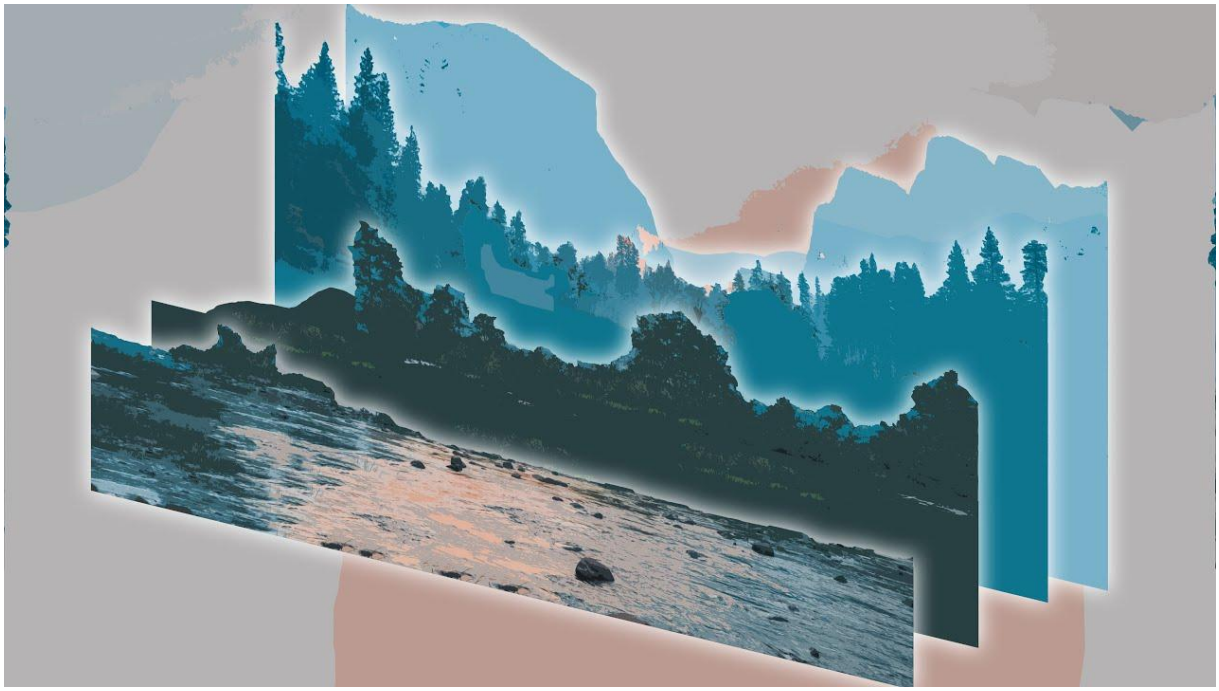
We did not go very advance in Audio management department as it is not in our scope. For basic sound players like player sound effects, theme song, background noises effects, this system is more than enough to give the result we want.

4.15 Special Effects and Level Design

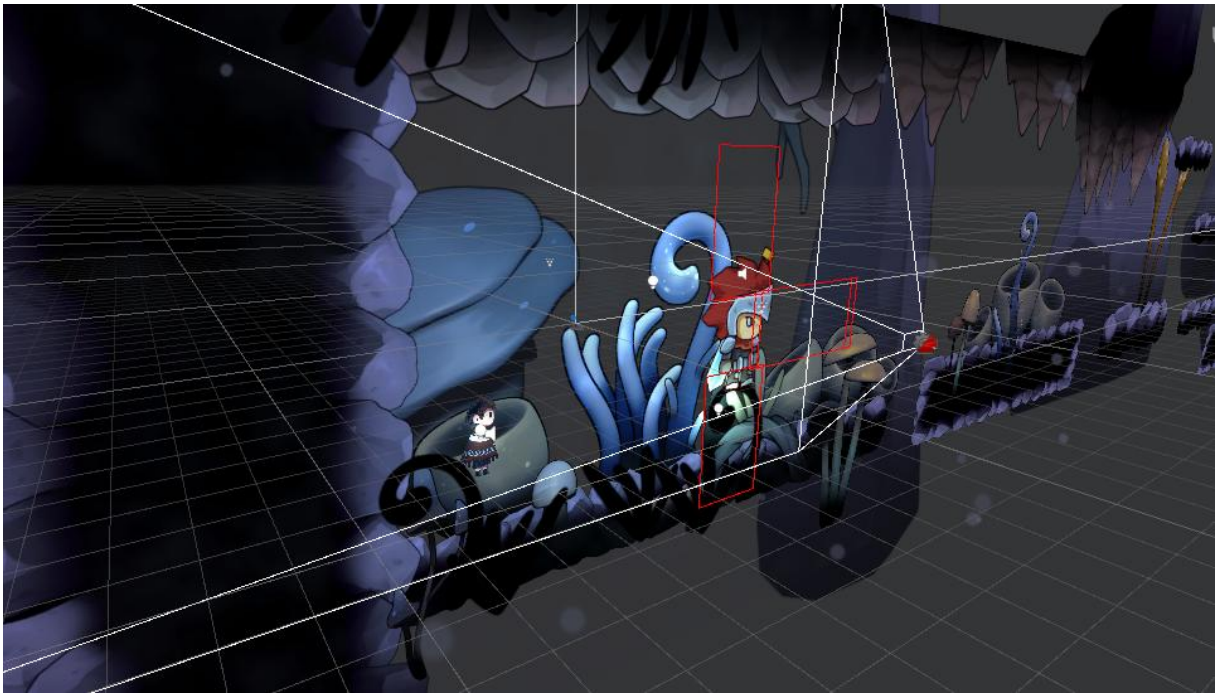
Parallel Background Effect: Our level design involves layering the platforms in different layers dividing into foreground, midground and background. Each layers is sub-divided into index of 01, 02, 03 and we position them in the number orders. For example, background 00 set position as 5, 01 as 10 , 02 as 15.

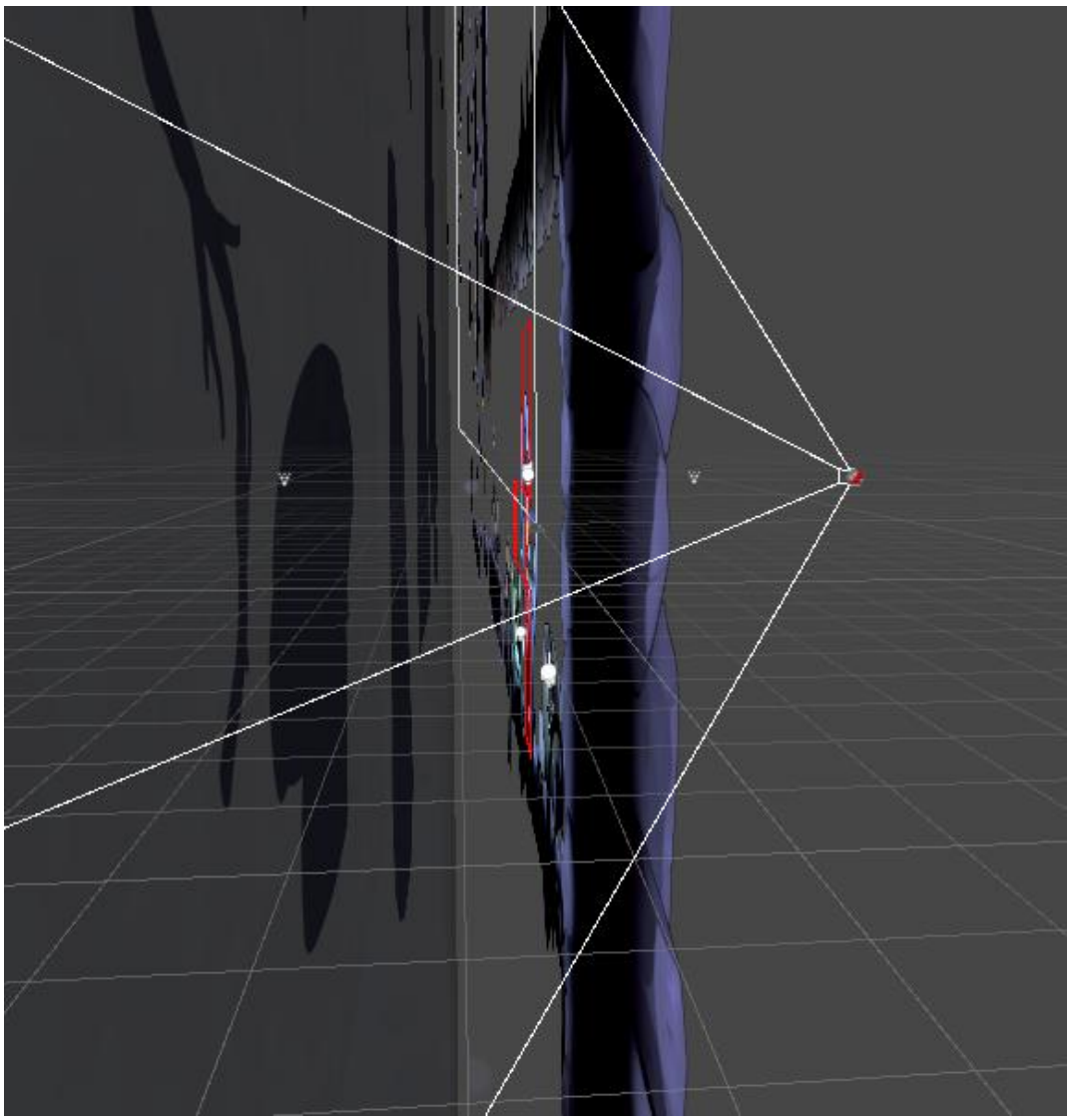


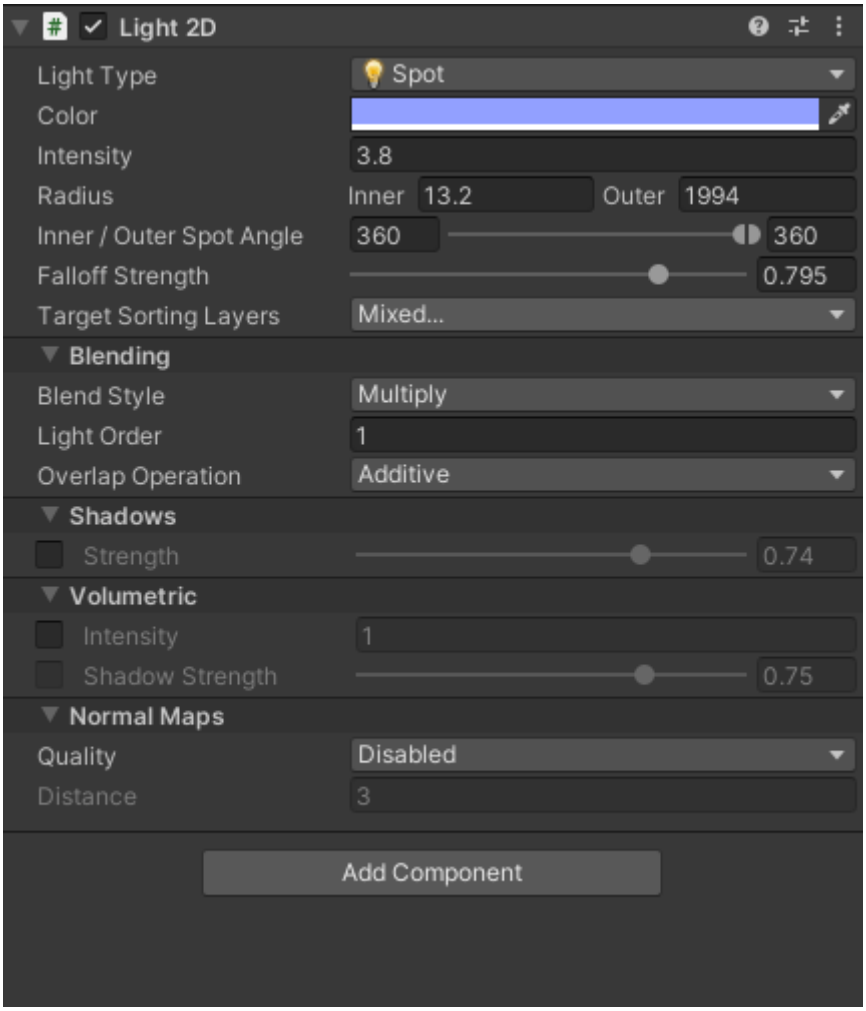
Each Layer object is assigned to different Sorting Layers. This is like paper stacking. The first paper with a drawing will show the brightest and the second layers will appear blurry. Unity's layer system help the camera identify how to differentiate the layers of the platforms



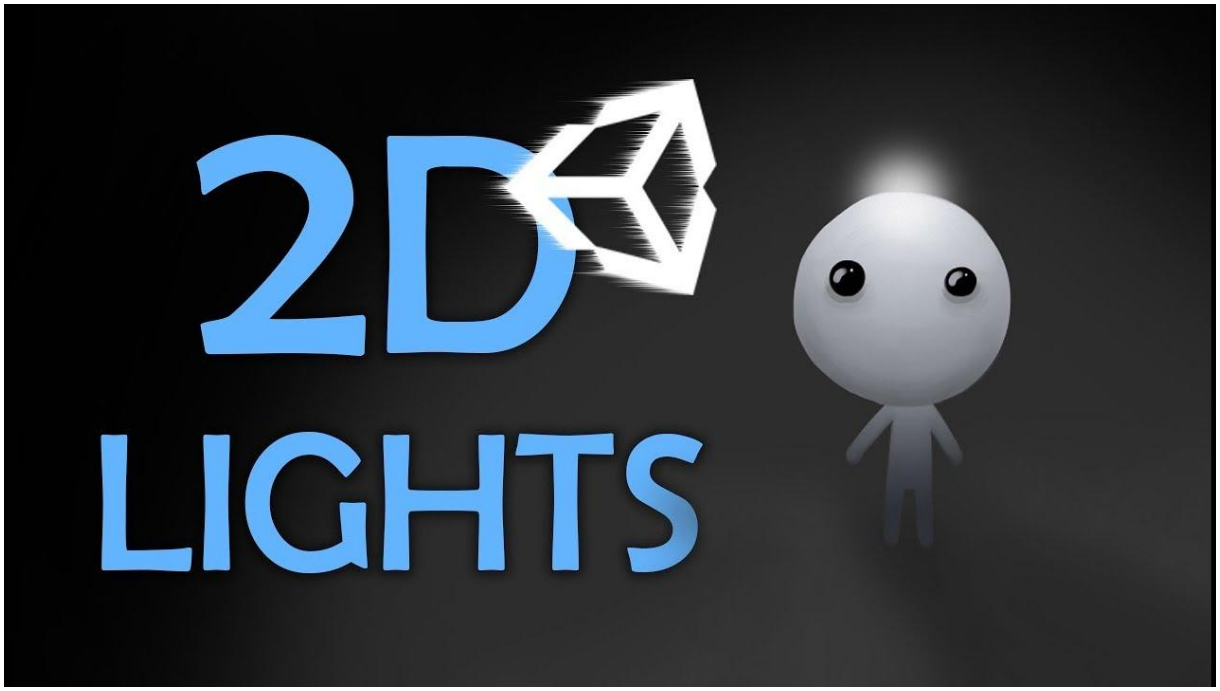
Parallax Background Effect is having multiple stacks of layers of image together and moving around your eyes to make it look like 3D. A Common technique used in animations by animators especially in 2D animation project. When an scene is static without the parallax effect, it is not alive as it should be and the view looks flat. This simple technique trick the camera that capture the scene give depts to the background as it moves around. Same principle happens to the eyes of human. By using this technique, we can trick our brain to feel more depths in the view we are looking.



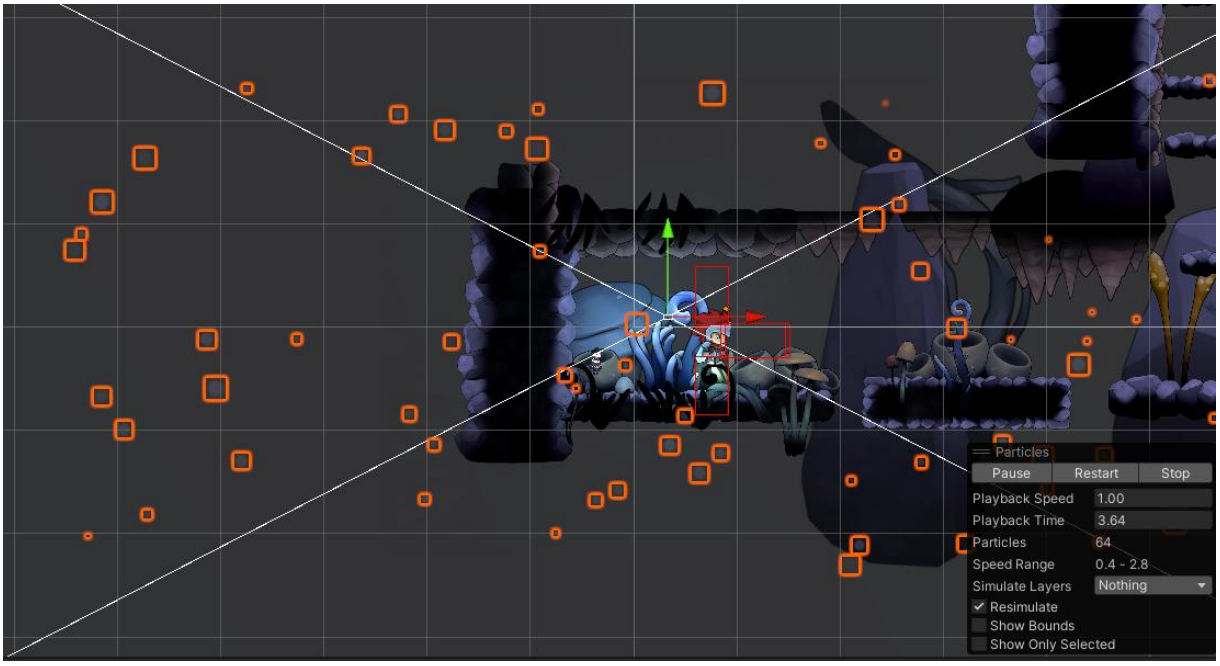


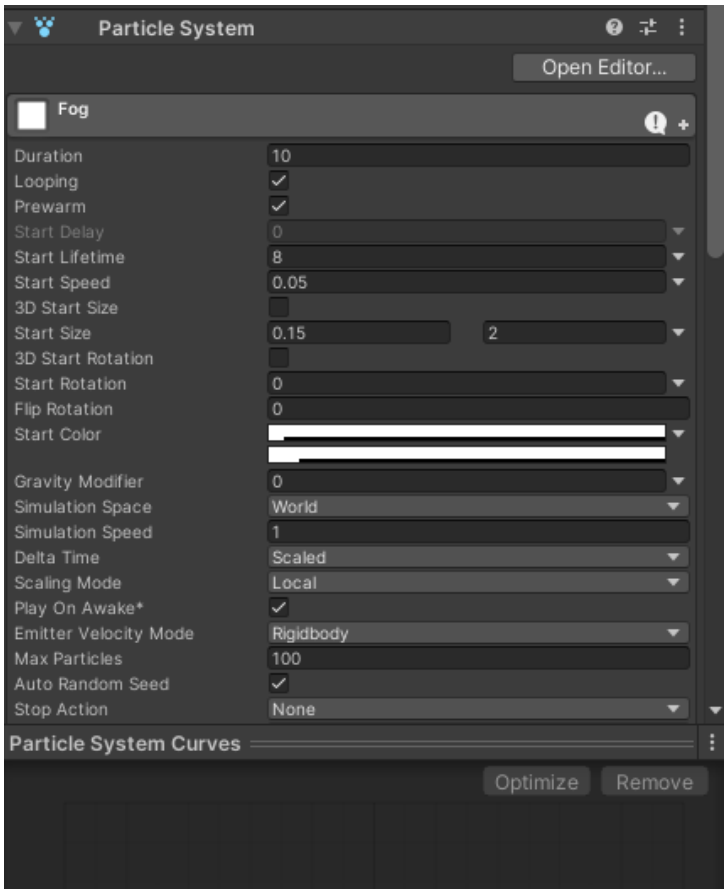


We use Unity 2d UDP package to create a 2D game with 3D light effects in our project. It is handy and easy to use. We can choose types of light from the component and assigned in the hierarchy. Chose which layer we want to get effected in "Target Sorting Layers". All the objects in those chosen layers will be effected by the light and have light and shadows.



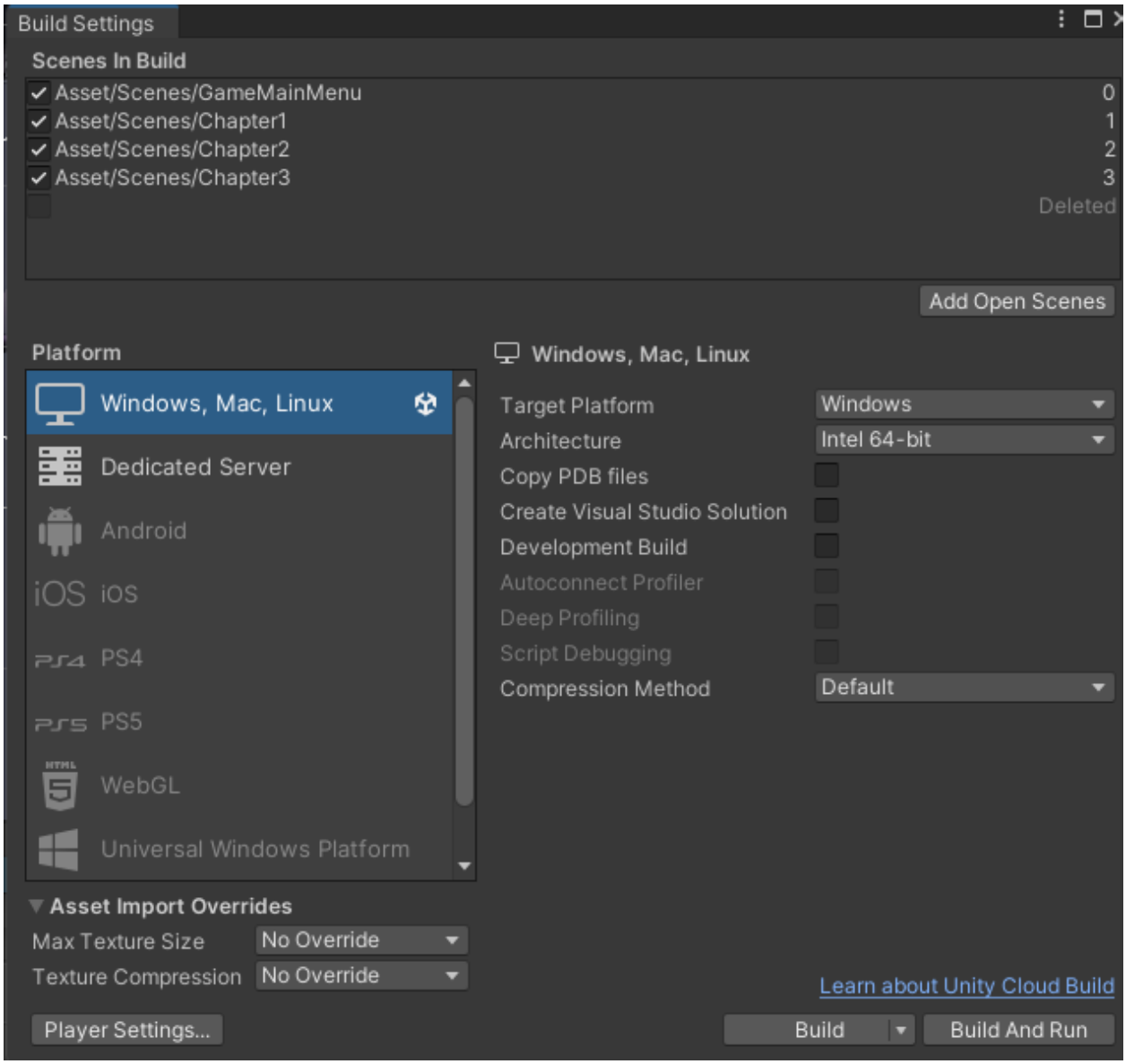
Particle effects in Unity are easy enough to go creative and make simple but appealing effects. In our project , our biggest use of particle effects is the “Fog effect” which when player move around the scene, user can see the circle shaped fogs in different sizes moving around.





We can create the size and the shape of particles in the particle system component. We can set duration time, looping the particle animations, life time of each particle, starting small and getting big or from big to small and disappearance, maximum numbers of particles in the scene , and auto random seed for generating particles at random places in the scene. This process is all about creativity. After we get the desired result, we append the particle object under the camera class so the effect move along with the camera since the camera follows player.

Chapter 5: Build and Deployment



Before we publish the game, we need to build it in unity, In the build setting, we can add the list order of level scenes and choose our platform, in our case, window is ideal for our high graphic game. And then, we upload in our platform choice itch.io which is free and the game is finally ready. We can always upload different versions of the game as part of the development.

ALostBoyVersion1.0

More information

Updated	Mar 21, 2024
Published	Mar 18, 2024
Status	Released
Platforms	Windows
Author	lucasMin
Genre	Adventure, Platformer
Tags	Singleplayer

Download

Download Now Name your own price

Click download now to get access to the following files:

AlostBoybuild1.0.rar 144 MB

Comments

Write your comment...

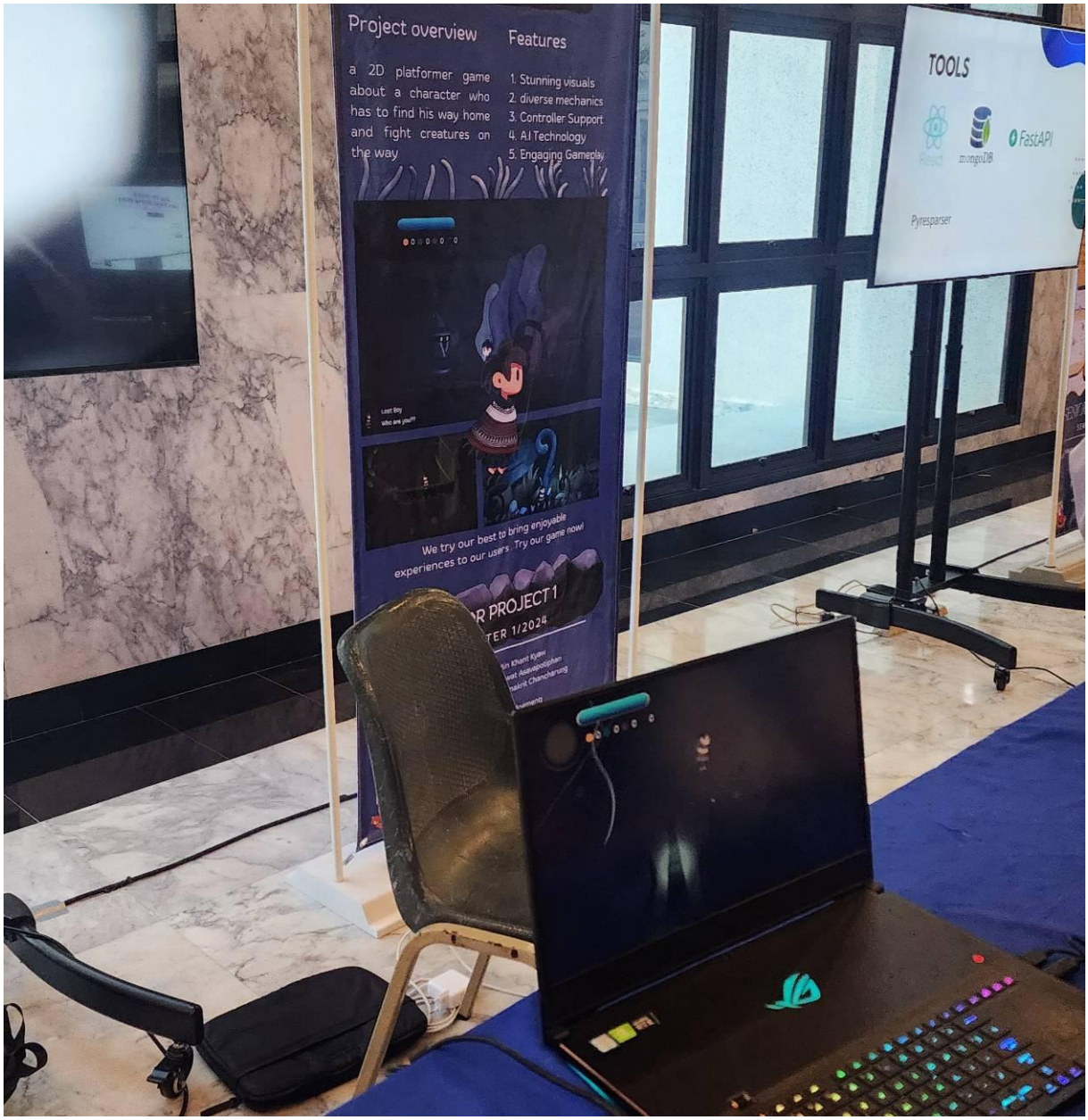
Chapter 6: Limitations and Emphasis

Limitation and Difficulties

One of the limitations of making a game project is the resources of assets. If we go for relying on the online assets, our game won't be original and will be just another Super Mario game. But if we go full original, it takes too much time and effort. This is why even in a small game dev team, there are developers, game artists, game designers, animators and sound designers for each role. When trying to cover all those roles in a 3-person team, there will always be difficulties. Secondly, time is our greatest enemies. With given time, we believe we can implement more functions and features.

Advantages

However, we do have a lot of advantages with our project. Our project is built from scratch to the end. It is a very solid foundation for everyone who want to take our project as template and use it in their similar project ideas. All the systems and functions are expandable and reusable as we chose the harder and more stable way over easy methods to make it easier in the long run. Most of the techniques used in game industry are covered in the project and we can always keep updating our project with version controls software. We also did a lot of user tests on our project exhibition day and get feedback from different types of people with different opinions. Based on those feedbacks, we can always keep fixing bugs and issues as game development is not a one-stop project. There will always be some one finding weakness in the game to overcome the system.



Chapter 7: Conclusion

Not just summarise what previous chapters, but conclude whether you proposed method successfully solve the problem and suggestion of what can be the next stage (future work)

For Conclusion, Did our method solve the problems

References

[1] Peter Waiganjo Wagacha. Instance-Based Learning: *k*-Nearest Neighbour, 2003

[2] Tom Mitchell. *Machine Learning*. MIT Press and McGraw-Hill, 1997.

[3] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, UK, 1995. ISBN: 0 19 853864 2.

[4] *Tom Gruber <gruber@ksl.stanford.edu>*Short answer: An ontology is a specification of a conceptualization.

[5] Uche Ogbuji is a consultant and co-founder of Fourthought Inc., a consulting firm specializing in XML solutions for enterprise knowledge management applications. Boulder, Colorado, USA.