The Crystal Mines - A 2D Metroidvania Game

NOO181604

SUPERVISOR: CATHERINE NOONAN
SECOND READER: JOACHIM PIETSCH

JOSHUA SEYMOUR (IADT STUDENT)

YEAR 4 2023
DL836 BCS (HONS) IN CREATIVE COMPUTING

Abstract

The goal of this project is to learn about 2D game design.

The game was developed in the Unity engine and coded using the C# (pronounced see sharp) coding language. The steps the author went through to create the game were: Research, Requirements, Design, Implementation, and Testing.

Acknowledgements

I would like to thank my project supervisor Catherine Noonan who gave me plenty of reassurance and brilliant advice throughout the duration of this project.

I would like to thank my project secondary reader Joachim Pietsch for providing useful feedback at the Interim presentation.

I would like to thank my friends and family for supporting me throughout the project.

Declaration

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

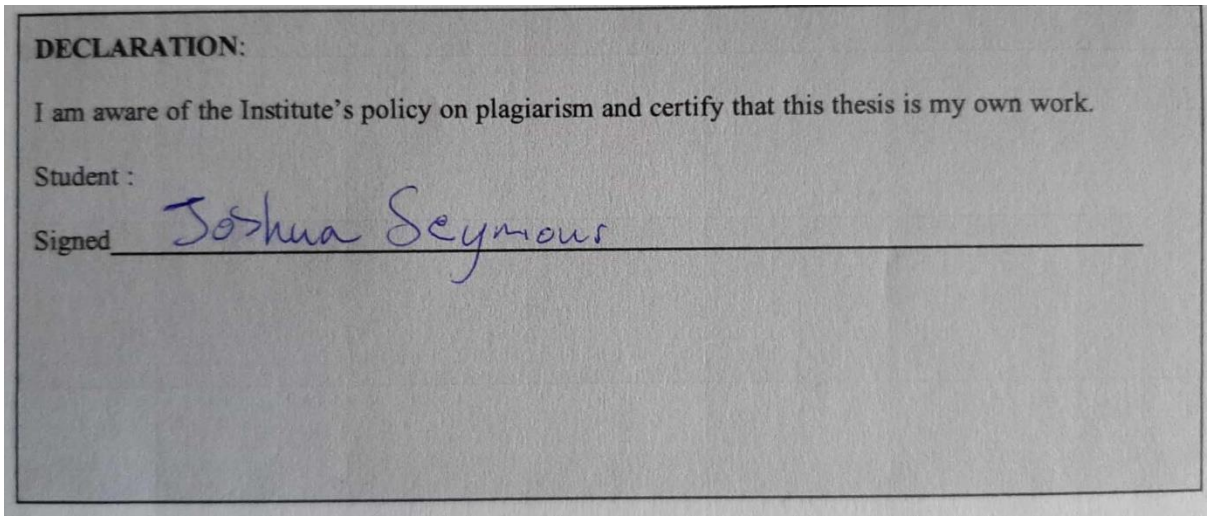
WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not leave copies of your own files on a hard disk where they can be accessed by others. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.



Failure to complete and submit this form may lead to an investigation into your work.

Table of Contents

Abstract 1

Acknowledgements 2

Declaration 3

1 Introduction 1

2 Research 2

2.1 Inheritance 2

2.2 Introduction 3

2.2.1 What is User Experience (UX)? 3

2.2.2 Why is the User's experience important? 3

2.3 Good video game characters 4

2.4 Flow theory 5

2.5 UX Design Process 6

2.5.1 What is UX design 6

2.5.2 The Effect of good UX design 6

2.5.3 How UX design is implemented 7

2.6 Conclusion 8

3 Requirements 9

3.1 Similar games 9

3.1.1 Advantages and disadvantages of both games 14

3.1.2 Desired Aspects 14

3.2 Survey 14

3.3 Persona 15

3.4 Functional Requirements 16

3.5 Non-Functional Requirements 16

3.6 Software 16

3.6.1 Advantages of Unity 16

3.6.2 Disadvantages of Unity 17

3.6.3 Advantages of Unreal Engine 17

3.6.4 Disadvantages of Unreal Engine 17

3.7 Paper Prototype 18

3.8 Story board Paper Prototype 20

3.8 Potential Difficulties 21

3.9 Conclusion 21

4 Design Chapter 22

4.1 System Architecture 22

User Flow 22
4.2 User Interface Design 26
4.2.1 Inspiration 33
4.3 Proposed Map Layout 36
4.3.1 The Crystal Village 1 36
4.3.2 The Crystal Village 2 37
4.3.3 The Crystal Mines 38
4.3.4 The Crystal Mines Boss 41
5 Implementation Chapter 43
5.1 Introduction 43
5.2 IsWalled() function 43
5.3 IsGrounded() Function 44
5.4 WallSlide() Function 46
5.5 Flip() Function 47
5.5.1 Flip() Function Update 47
5.6 Player Movement 49
5.6.1 Fire 50
5.6.2 Jump 51
5.6.3 Dash 51
5.7 Fire() Function 52
5.7.1 Detecting enemies 53
5.7.2 Damaging enemies 53
5.8 Knockback.cs 56
5.9 Move() Function 58
5.9.1 Updated Move() Function 59
5.10 Player Jump 61
510.1 Jump() function Updated 62
5.10.2 Coyote Time Jump 64
5.10.3 WallBounce() Coroutine function or Wall Jump 66
5.10.4 WallJumpCooldown() Coroutine function 69
5.10.5 Double Jump 70
5.11 Dash() Function 72
5.11.1 Dash() Coroutine Function 72
5.12 Interact() Function 74
5.12.1 CheckInteractions() Function 75
5.12.2 NPCInteractable.cs 76

5.12.3 ResetText() Function 77
5.12.4 Typing() Function 77
5.12.5 ResetText() Function 77
5.12.6 OnTriggerEnter2D() & OnTriggerExit2D() Function 78
5.12.7 Updated NPCInteractable.cs 79
5.13 One Way platforms 81
5.13.1 OneWayPlatform.cs script 81
5.13.2 OnCollisionEnter2D() Function 82
5.13.3 OnCollisionStay2D() Function 82
5.13.4 DisableCollision() Function 83
5.14 RespawnScript.cs 85
5.14.1 CheckpointScript.cs 86
5.15 PauseScript.cs 87
5.15.1 PauseCheck() Function 88
5.15.2 PauseGame() Function 89
5.15.3 ResumeGame() Function 89
5.15.4 Updated “PauseScript Update()” function 91
5.15.5 PauseGameDeath() Function 91
5.15.6 RestartGame() Function 92
5.15.7 Updated PauseScript.cs Start() Function 92
5.16 EnemyScript.cs 93
5.17 MainMenu.cs 94
5.18 SettingsMenu.cs 95
5.18.1 SetResolution() 95
5.18.2 SetQuality() Function 98
5.18.3 SetVolume() 99
5.18.4 Updated SetVolume() Saves between scenes. 101
5.19 AudioManager.cs. 103
5.19.1 Play sound Functions 105
5.20 BgMusic.cs 106
5.21 Inheritance 107
5.21.1 Character.cs 108
5.21.2 Player.cs 113
5.21.3 NPC.cs 117
5.21.4 Enemy.cs 118
5.22 SceneController.cs 122

5.23 LoadNextArea.cs 124
5.24 HealthBar.cs 128
5.25 Tile map 129
5.26 Level Creation 131
5.26.1 The Crystal Village 1 131
5.26.2 The Crystal Village 2 131
5.26.3 The Crystal Mines 1 132
5.26.4 The Crystal Mines 2 Boss room 133
5.27 Sprite sheet 134
5.28 Camera 137
5.28.1 Cinemachine settings 137
5.28.2 CameraFollowObject.cs 138
5.29 Parallax 139
5.29.1 Parallax.cs 140
5.30 Animation 143
6 Testing Chapter 147
6.1 Usability Testing 147
6.2 The Objective of Testing 147
6.3 The Tasks 147
6.4 The Results of Testing 147
6.6 Feature Testing 148
7 Project Management 153
7.1 Introduction 153
7.2 Phases 153
7.2.1 Proposal 153
7.2.2 Research 153
7.2.3 Requirements 153
7.2.4 Design 153
7.2.5 Implementation 153
7.3 SCRUM Methodology 154
7.4 Project Management Tools 154
7.4.1 Trello 154
7.4.2 Journal/Notes 154
7.4.3 GitHub 154
7.5 Reflection 154
7.5.1 Deferral 154

7.5.2 Your views on the project 154
7.5.3 Working with a supervisor 155
7.5.4 Conclusion 155
8 Business Opportunities 156
8.1 Paid Download 156
9 Conclusion 157
10 References 158
10.1 Research 158
10.2 Requirements 159
10.3 Implementation Assets 159

1 Introduction

This report is divided into sections to explain the process used to create the 2D game. The first chapter is the **Research** chapter, this goes into detail on Inheritance, design process, and how the background research was done. The second chapter is the **Requirements** chapter. This lays out the requirements the author set for themselves. The third chapter is the **Design** chapter. This explains the design process, where the author got their inspiration. The fourth chapter is the **Implementation** chapter**.** This goes into detail on how the project was assembled. The fifth chapter is the **Testing** chapter. This goes into detail on feedback from participants who tested the game and how / if their feedback was implemented.

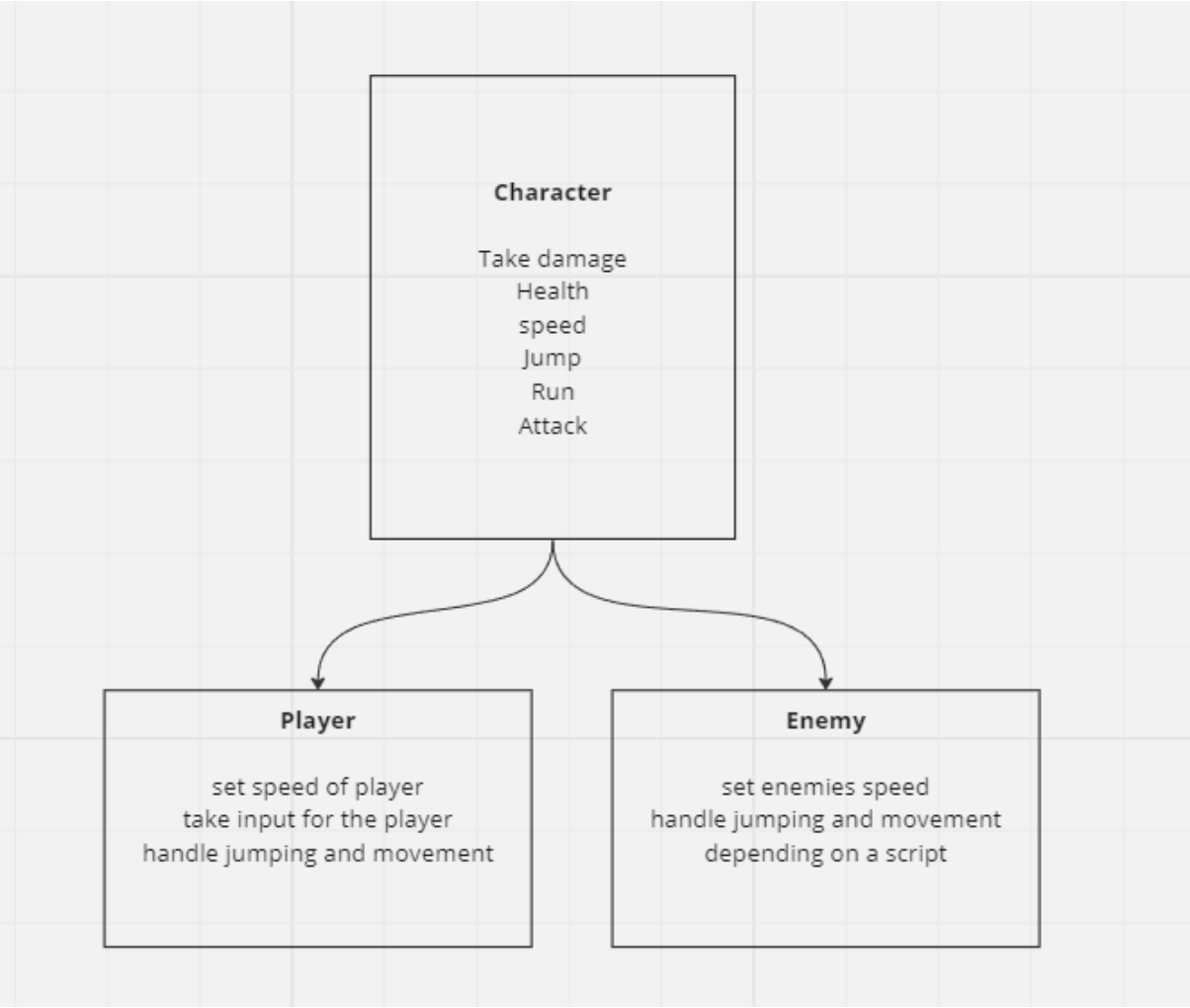
2 Research

This chapter goes into detail on research the author has done before starting this project.

2.1 Inheritance

C# uses Inheritance to reduce repeated code. An example of how Inheritance would work is shown below.

In this example the “Character” class is the parent class, and the “Player” and “Enemy” classes inherit functions and variables from “Character”. This means if the “Player” class uses the same function as the “Enemy” class, it can be removed from both and only added to the Character class. The child class can choose to change the parent class functions or keep them the same. *(Hejlsberg et al.)*



The Author wrote a literature review to look at why a user’s experience is important, what makes the game enjoyable and how enjoyment can be achieved through a design process.

2.2 Introduction

For this literature review I was focused on looking at UX in general, I am interested in how good UX is achieved. I hope to implement what I research into my final project. I want to mainly focus on what is UX and what makes it enjoyable, to do this, I will look at readings that are or could be related to video games so that I can apply what I learned to my final project. In this literature review I will be researching topics surrounding UX which consists of, What User Experience is and why it is important, how creating good characters can make the UX more enjoyable for video games and how flow theory helps with UX. Also, I will be looking into the UX design process and seeing What is UX design, The effect of good UX design and how it is implemented.

2.2.1 What is User Experience (UX)?

UX is a design procedure that is focused on making catered experiences with any technology, product, and service. Interactions with these experiences can be physical, mental, or mechanical. UX includes all the touch points a user has with the product, service or technology, these points include experiences such as, finding, purchasing, using, repairing / maintaining the product, service, or technology *(Rosenzweig, E., 2015)*. How a user interacts with and experiences a product is their User Experience.

2.2.2 Why is the User's experience important?

The UX design process focuses on placing the user in the centre and creating everything around them, so when the product is fully created, the experience will be instinctive, accommodating and hopefully enjoyable *(Nichols & Chesnut, 2014)*. It is important to focus on the UX design before the product design as it ensures the customer's experience is the priority and assures the product won't have to be re-developed after. Good UX design can improve people's day to day lives in big and small ways.

For a company, having good UX is important because the users are the ones purchasing and using the product. A product with a good UX influences the user to have a positive view of the overall product, which generates customer loyalty and brings back a customer for other products created by the company. A bad experience with a product influences the user's attitude towards the product as it is likely that the product has wasted the users time, effort, or money. When a customer has a bad experience with a product, they are much less likely to come back and try this product or related company products again. The reason for this is that our brain stores bad experiences and learns to avoid them. Bad experiences are more prominent in our mind then good experiences *(Sparks & Baumeister, 2008)*. For example, if two people are married for twenty years and they have loved each other throughout the whole relationship, one person cheating on their partner overwrites those twenty years of loving marriage *(Baumeister et al., 2001).* The disappointment of bad UX will cause unsatisfaction and overwrite the good experience causing the user to avoid the company's product. Therefore, it is important for the company to focus on designing a good UX and continue to improve it.

2.3 Good video game characters

In video games well designed characters contributed to satisfying UX. In video games wanting to like or relate to the main characters is important as it is the character the users see themselves as or is rooting for. It is beneficial for the user to care for the main character instead of dislike them and not want to play as them, because of this, usually the main characters in video games tend to be attractive. In studies it has been found that attractive people are associated with being warmer, kinder, stronger, more sensitive, more outgoing, more socially persuasive, dominant, and smarter than others. Other studies have found that having 'Babyface' features such as large eyes and pupils, small chin, high eyebrows and forehead, small nose, full lips and cheeks resemble a baby and through evolution it's been found that people tend to care more towards people with these features. 'Babyface' features are used a lot in video games when there is a character that needs to be taken care of and liked. For example, in games like "Hollow Knight" where the main character commits violent acts users are not inclined be concerned about these acts and generally brush them off because the character design makes the character forgivable (Isbister, K., 2006). Desirable character design is essential for video games as users are more inclined to connect with the characters and continue playing them, this leads to an enjoyable UX.

Stereotypes are also often used in video games. Stereotypical character designs allow users to quickly make an impression on whether the character is good or bad. Users don't have to figure out whether an approaching character is trustworthy or possibly dangerous if their character design is based on stereotypes. i.e., an enemy's threatening pose and menacing look will let the user know that they are dangerous. Quick first impressions are important in fast-paced games as the user needs to take in a lot of information quickly and make actions in case of possible danger (Isbister, K., 2006). Easy to read characters ensure that players are able to navigate their way easily through the game making their UX enjoyable.

2.4 Flow theory

A satisfying UX consists of a user that learns a product and uses it aimlessly. It has been observed that people who get immersed into activities neglect basic human needs such as food, water, and rest. In interviews of people who pursue activities for enjoyment, it has been found that the factors for entering flow are challenges or opportunities that do not exceed the persons current skills and distinct goals and instant progress feedback. Characteristics of entering the flow state include.

- High focus on present activity.
- Actively aware of current task.
- Unaware of surroundings.
- Able to predict future occurrences.
- Disregard of time.
- Activity is personally rewarding; the process is more rewarding than the end goal.

Flow state can occur in simple activities as all individual people will have unique challenges that they want to overcome. People's personal interests are subjective therefore everyone's experience of enjoyment is different. Distractions, unrealistic goals, or lack of skill can disrupt the flow state. Peoples personal interest increases growth in skills as they pursue activities that are just above their skill level. The constant progress and achievement of goals results in them repeating that activity constantly looking for new challenges within that activity, this raises their enjoyment for the activity *(Nakamura, & Csikszentmihalyi, 2009).* This amount of immersion improves the UX of the product.

2.5 UX Design Process

2.5.1 What is UX design

Design is a thought-out plan that focuses on creating a final piece of art that is user centric. Good designs are often un-noticed as we expect products to do what they are advertised to do; however, excellent, or bad designs are noticed because they are either highly appreciated or frustrating. A planned design consists of

- Point
- Line
- Form, Shape, Space
- Movement, Direction
- Colour, Value
- Pattern
- Texture
- Size

The shape and form of a product is based on the function of the product and its intended use. The form must complement the products purpose. Good product form will ensure the intended use of the product, making it intuitive to the user. Bad form will make the product less ergonomic and annoying to use. Design is a way to solve problems with an end goal in mind, it's based on concentrating on recurring problems and making life easier. Good design concentrates on solving problems that haven't been solved before in hopes of achieving innovation and overwriting previous technology *(Rosenzweig, E., 2015).*

2.5.2 The Effect of good UX design

People learn in a multitude of ways as not every person is the same, some people take in information in visual, auditory, reading and writing and kinesthetic learning styles. Every individual will interact with products in unique ways. People interact with products in ways that accommodate their needs, for example, people with physical disabilities such as visual impairment will require tools that help with vision. UX design must be aimed towards one main target audience, however, should consider people outside the main group and make their product more widely accessible.

To keep good UX design in mind a Persona can be used to guide the design in the right direction. A persona may be developed when considering a problem in mind. The design of a specified persona will help guide the product design towards the target market. When designing a persona, the characteristics that are often included are

- Demographics: Age, Gender, Education level.
- Goals
- Limitations
- Motivations
- Environment

This ensures that the product is tailored towards the intended audience answering questions like: What are they trying to achieve, what obstacles do they face, what motivates them and what conditions surround them *(Rosenzweig, E., 2015).*

2.5.3 How UX design is implemented

Once the relevant research and planning has been done and a persona has been established, the product itself can be designed and fabricated. The order of UX design process should be done in 5 stages, which are:

- Concept, sketching and flows
- Wireframes and prototyping
- Visual design and interactions
- Documentation
- Development

Concepts – With all relevant data obtained, the start of a concept can begin. Methods such as, storyboards, project briefs or project outlines help with coming up with product concepts, it can help visualise what the product will be used for or how it will be used.

Sketching – Sketching is the most efficient way to quickly visualise many variants of concepts for the final product. It allows the developer to visualise many possibilities, pick the best suited option and develop it. Seeing the sketch helps pinpoint where there might be design flaws. These flaws can then be singled out and solved before the final concept is decided on. All the final details of the finished product don't have to be included in the sketches as sketching is an early stage of concept designing and it is there for a basic visualisation of the product.

Flow – Flow is the state of moving from one stage to the next stage of a product. Every interaction will lead to another interaction in a product until the user's desired outcome has been reached. For example, every interaction on a phone brings a user to a new screen or popup that leads them further to their desired screen location.

Wireframe – Wireframes are a sketched out visual guide that show the products structure and shows the main features and elements of the product. When the wireframes are made, the flow of the product are also incorporated into the wireframe sketches. Wireframes are created first by sketching them in a low-fidelity version which will have basic necessary features. Wireframes are created to show the structure and how the most essential interactions with the product will look like. A good wireframe will become an important reference for the prototype stage.

Prototyping – Prototypes are used as a representation of what the final concept may look like, it allows its developers and users to view the potential final product and consider its feasibility. Prototypes should include all of the key elements from the concept design, this means that they are generally high-fidelity as they are made to be more interactive and more suited for the end user.

Visual Design & Interactions – Visual Design is how the final product appears to the user and how it responds when interacted with. Visual Design is created by combining UI design and graphic design. Visual Design includes important elements such as: colours, text, images, and form, these help with understanding the product visually. A consistent and understandable style needs to be created and implemented throughout the product. The UI designer is responsible for creating high quality designs that are universally readable and a layout that is easy to navigate. The interface should be clear and smooth so the user can easily tell that an interaction has taken place, for example,

when a user clicks on a desktop icon, a loading symbol appears beside the pointer to let the user know that the computer is processing.

Documentation – Documentation occurs after the design has been planned and decided on. Documentation is the process of recording all final decisions that have been made for the design of the product and organising it into different divided sections. A well organised documentation will help future developers to understand what is being requested to develop.

Development & Production – lastly the design gets sent to a development team where development begins. During the development phase the developer may send back the product for review or give feedback and receive clarification. Changes may be made during the development stage as feedback gets sent back and forth. All changes made get recorded into the documentation. The product is developed until a satisfying outcome is reached. The developed product gets quality checked and goes through usability testing. The performance of the product is measured and corrected. The product is eventually launched, and feedback is taken into consideration *(Canziba, E., 2018).*

2.6 Conclusion

In conclusion I have found that UX is a design process that is catered for specific users. Good UX design is important as it ensures that the user is the priority of the product design, and they are satisfied with the product. This guarantees that a company has not wasted time, effort, or money in making products that are unsatisfying. For video games, good character design is more likely to make a user invested in the game and become immersed increasing their UX satisfaction. UX design is a plan that focuses on creating a product for a user, good UX design takes into consideration how the product can be more suited towards a specified target audience. There are many steps taken for a UX design to be implemented, these steps make sure that the final product is catered towards the targeted audience and will be successful.

3 Requirements

This chapter will focus on the requirements to make a game. This chapter will investigate similar games. The survey will gather information on what people want from a 2D game, personas, functional requirements, non-functional requirements, a wireframe sketch of the game map, a story board of the game, what software the Author will use, and the difficulties the creator of the game will potentially encounter. The author's game will be 2D and contain an interesting environment along with a boss (hopefully multiple bosses) the player can fight. The game will have abilities that the player can find upgrades for, and it will have multiple different types of enemies and allies for the player to interact with and fight.

3.1 Similar games

Hollow Knight

Figure 1 is a screenshot of a game called *Hollow Knight (CHAN, 2022)* which the creator would like to emulate. However, the scope will be smaller as there is not a team of developers working together. In **figure 1** there are 5 skulls representing the player's health bar. Under the skulls there is a number representing the money the player has, and to the left there is a big circle that will fill up with "soul" when the player damages enemies. The player can use this "soul" to heal.

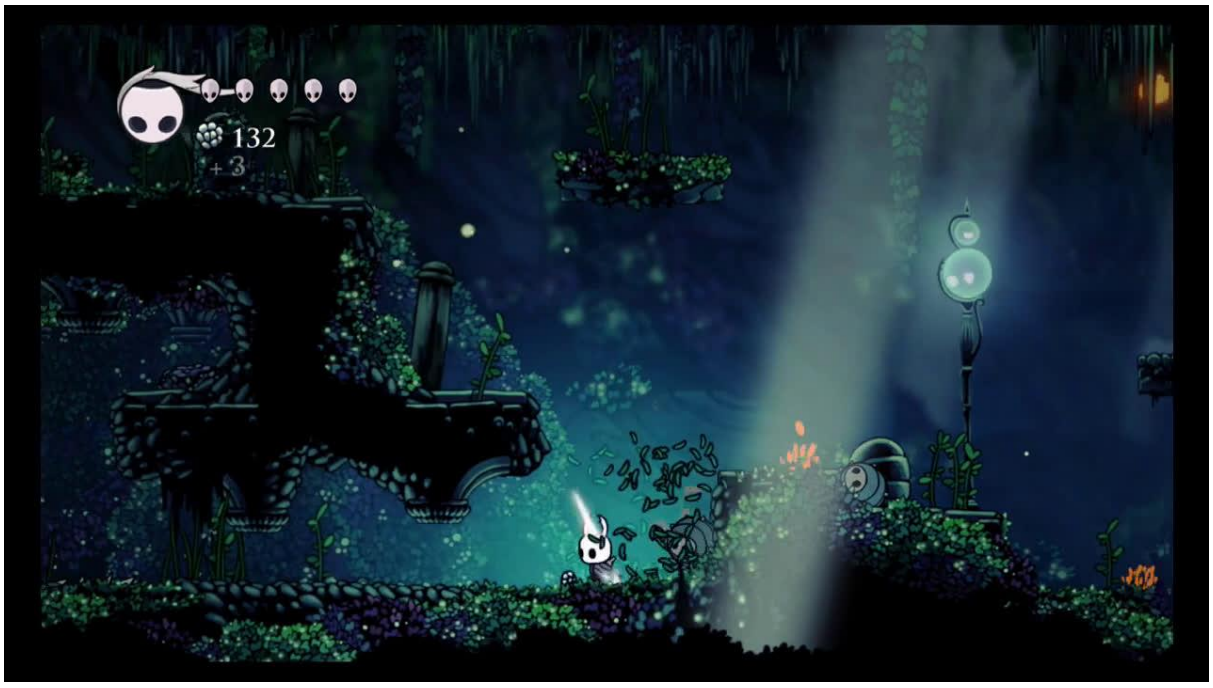


Figure 1: Screenshot of Hollow Knight game environment and UI

Figure 2 displays the different types of enemies the player can encounter in Hollow Knight *(Locke, 2022)*.

Figure 2: Screenshot of Hollow Knight game showing different types of enemies.

Figure 3 shows one of the many boss fights from Hollow Knight. Most of the boss fights in Hollow Knight have two phases. The first phase being easier and the second adding another ability or more challenge to the fight.

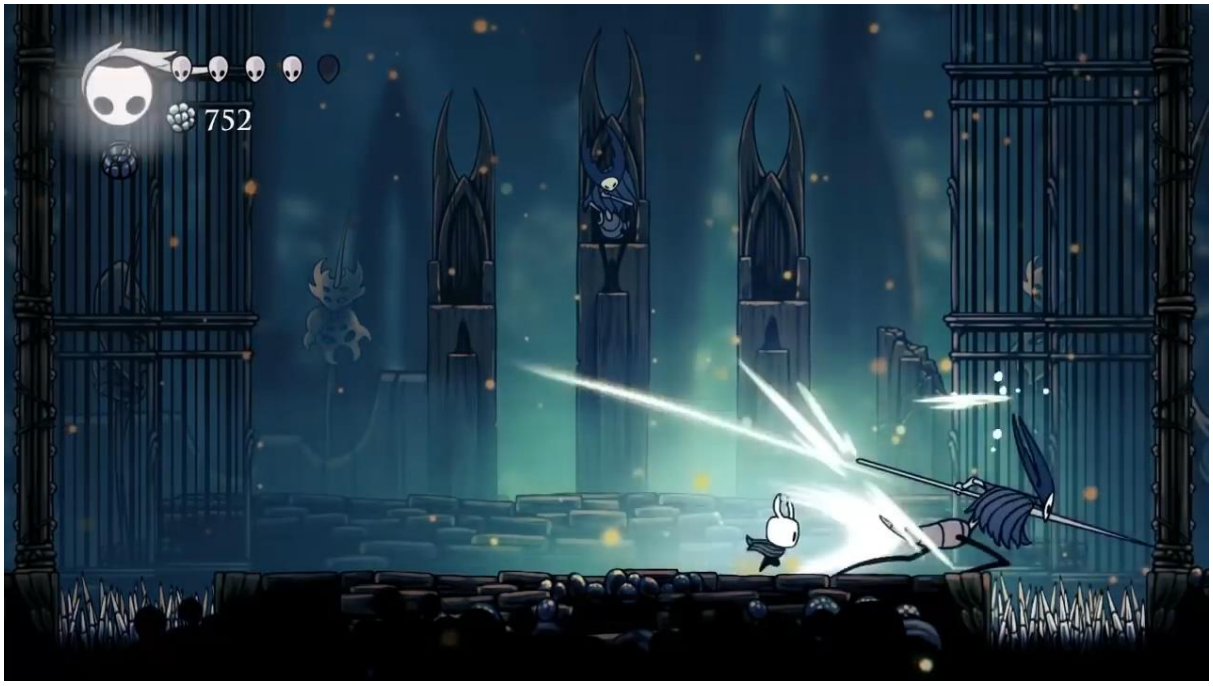


Figure 3: Screenshot of a Boss fight in Hollow Knight

Ori and the Will of the Wisps

Figure 4 and **Figure 5** are screenshots of a similar game called *Ori and the Will of the Wisps (Irwin, 2022).* The game is a 2D Metroidvania. Metroidvania is a sub-genre of action-adventure games where the player can explore a nonlinear story line while also having progression based on upgraded abilities. In this game the player encounters many incredible characters and is compelled to explore the beautiful world while following along with the story of the game.

Figure 4: Screenshot from Ori and the Will of the Wisps showing the player interacting with the creatures in the world.

Figure 5: Screenshot from Ori and the Will of the Wisps showing the player interacting with the creatures in the world.

Figure 6 shows a screenshot of the basic UI. In the top right there is the currency called spirit light, the player can buy maps and items with it. At the bottom middle of the screenshot there are the 3 abilities the player can use. (Ori and the Will of the Wisps PC & Console 2020, 2022)

Figure 6: Screenshot of Ori and the Will of the Wisps UI

Figure 7 shows a screenshot of the upgrade system. The player can acquire may upgrades that help the progression of the game and make it easier for the player. (Crego, 2020)

Figure 7: Screenshot of Ori and the Will of the Wisps Upgrade abilities

3.1.1 Advantages and disadvantages of both games

Hollow Knight advantages

- Art
  Music / Environment / Enemies / Unique Characters
- Story
  Deep lore (items and characters have a complex back-story)
  Multiple side Quests / Stories
- Exploration heavy
- Collectables

Hollow Knight Disadvantages

- Very difficult combat (frustrating for beginners)
- Story can be hidden and hard to follow.
- Too much retracing steps

Ori and the Will of the Wisps advantages

- Art
  Music / Beautiful Characters / Beautiful World
- Story
  Emotional / Captivating
- Intricate Problem-Solving Puzzles
- Exploration heavy
- Collectables

Ori and the Will of the Wisps disadvantages

- Very Difficult Beginning
- Poor Performance (FPS drops and stutters)
- Enemies can move and shoot at the player while the player is still loading into an area.
- No Mini Map + Unintuitive Menu Map
- Lots of retracing your steps
- End Game is too easy.

3.1.2 Desired Aspects

The author would like the game to be visually appealing, while having unique enemies and characters. The author wants there to be optional quests (E.g., Save certain characters and the player will be rewarded). The author also wants the mini map to be easy to use.

3.2 Survey

https://docs.google.com/forms/d/e/1FAIpQLSeGQyMOOgO1SkYDsg4qMcWu6zxPIq0sB7CGoT2GqFFmpB1UbA/viewform?usp=sf_link

This is the link to the Authors survey in which they ask the participant what aspects of 2D games they like and how Important they are to a game. The survey asks for the participants age and gender to see the demographic of people that took the survey.

3.3 Persona

Figure 8 and **figure 9** show two personas that were created to illustrate potential users of the game. In **Figure 8** Gerry likes difficult, fun, and satisfying games while not being too interested in the story or the uniqueness. In **Figure 9** Samantha is much more interested in the Story and the enjoyment of the game while being frustrated with difficult games.

A way to solve the problem of having a game that is hard but also accessible for people who don't like difficult boss fights, is to make the very hard fights optional.

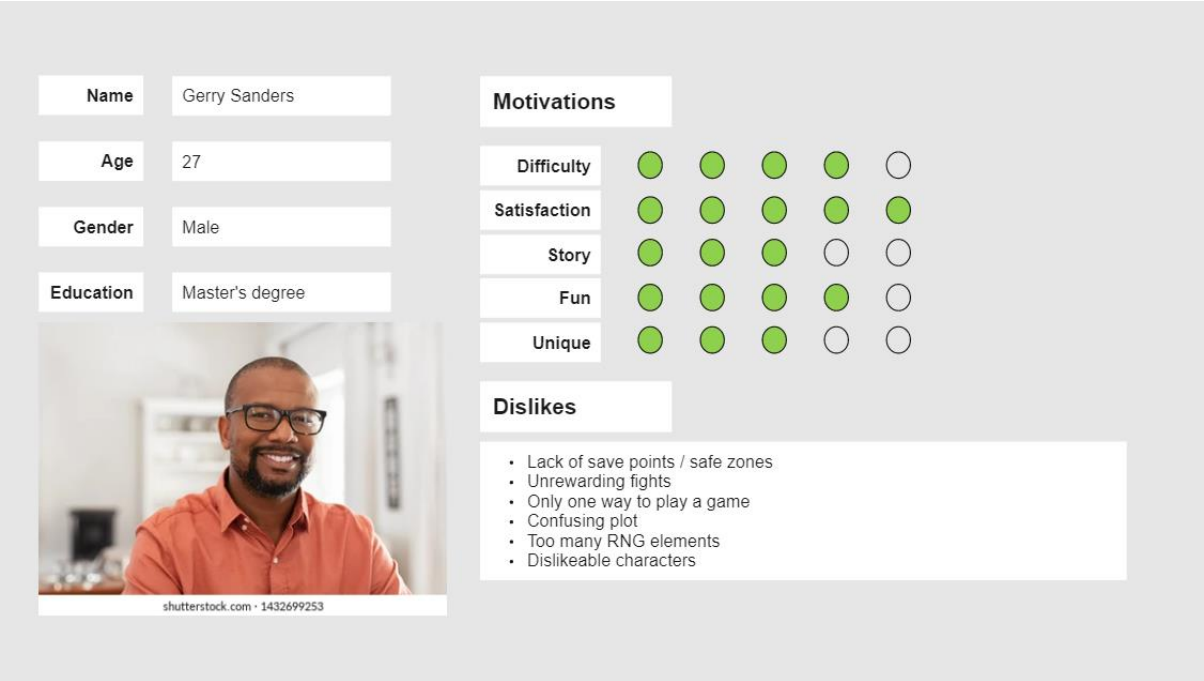


Figure 8: Persona 1

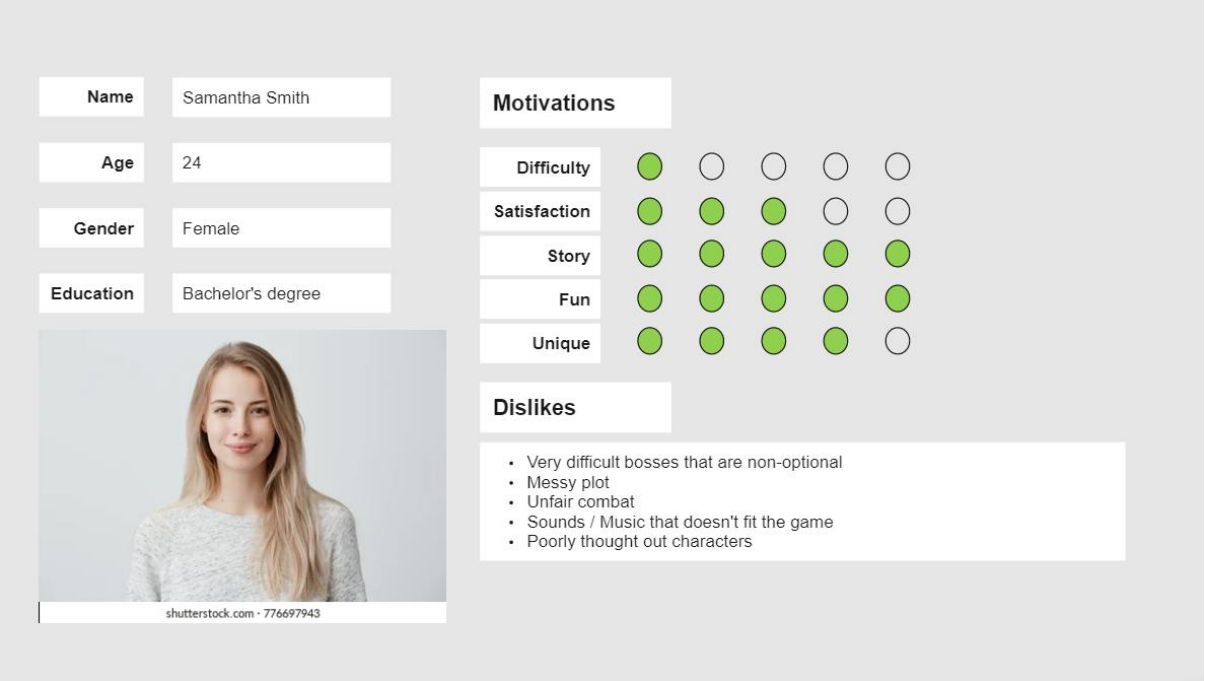


Figure 9: Persona 2

3.4 Functional Requirements

- Basic movement controls
- Health bar
- Sword attack
- Story progression
- Item pickups
- Enemies
- 3 different areas to explore (possibly 4)
- A boss fight for each area (this might be too much for the time frame)
- Autosave
- Save files.

3.5 Non-Functional Requirements

- Satisfying movement
- Pleasing Haptics
- Interactable environment + Destructible terrain

3.6 Software

The author researched Unreal Engine and Unity to compare which was the better option to use in their project. The author will be using Unity to create and run the game and visual studio code as their code editor. They will be using a physics engine.

3.6.1 Advantages of Unity

- The Author has completed 2 projects in Unity already.
- Unity Assets store has many easily accessible free assets (mainly 2D).
- Free Software

- Compatible Cross Platform Development
- C# Coding language (easier than C++, The Author has used C# before)

3.6.2 Disadvantages of Unity

- Performance-intensive
- Large Unity games take up a lot of space.
- Unity Assets store for 3D models

3.6.3 Advantages of Unreal Engine

- Real-time networking for multiplayer
- Beautiful Visual Effects
- Extensive 3D Asset libraries

3.6.4 Disadvantages of Unreal Engine

- C++ Coding Language (The author would have to learn this language)
- Harder than Unity to port to another platform

3.7 Paper Prototype

These are the Paper Prototypes the creator sketched for the map of the game.

Figure 10 shows the starting area of the game. The player learns the controls of the game here, they learn to jump and attack.

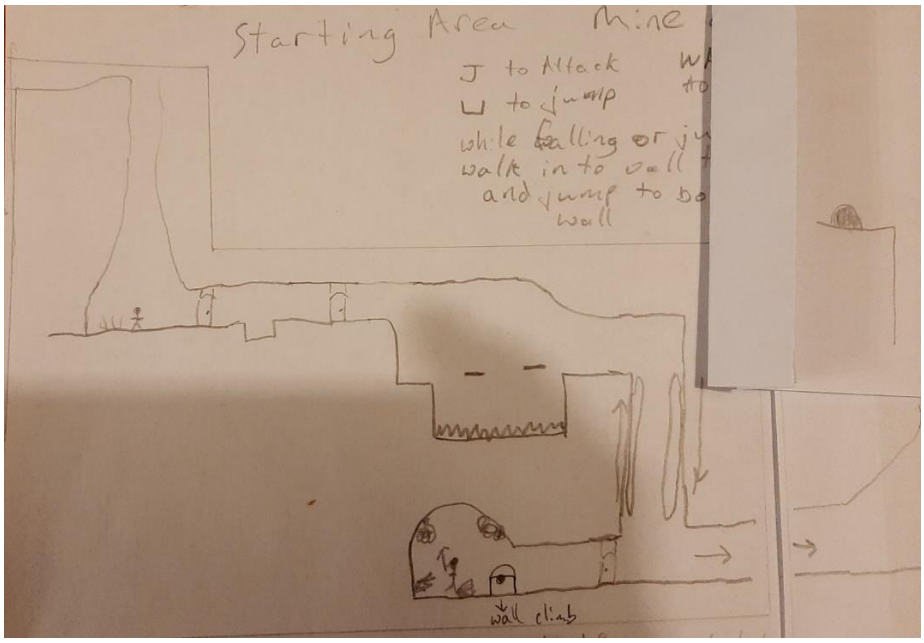


Figure 10 Starting area Paper Prototype

Figure 11 shows the mid-section of the first area where the player learns to wall jump and scale walls. Wall jumping is when the player can jump off the wall in the opposite direction to get to different areas of the map. Wall scaling is when the player slides down a wall by walking into the wall while falling, this slows the player's fall speed and allows for more time to think about where they would like to go.

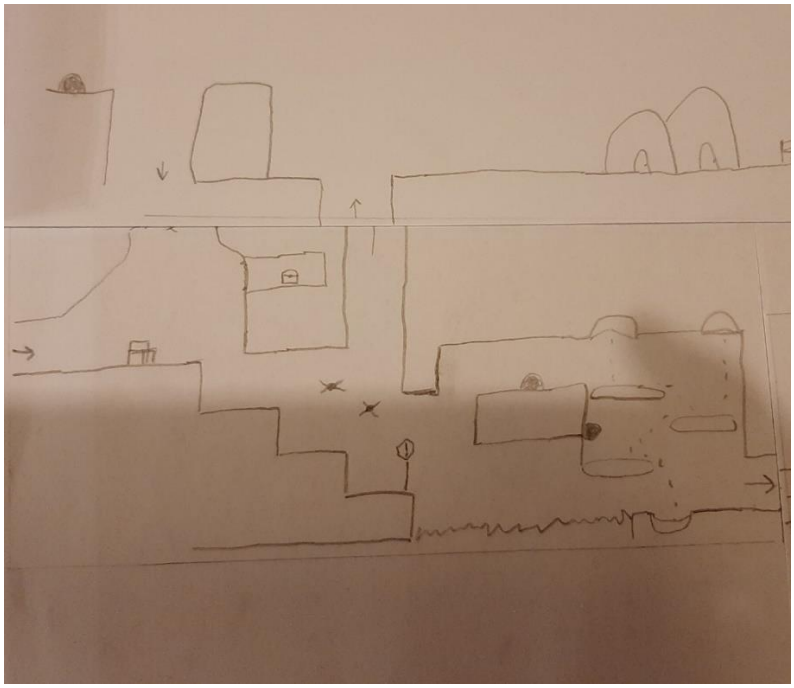


Figure 11 Starting area Paper Prototype mid-section.

Figure 12 shows the final section of the first area, the boss fight room.

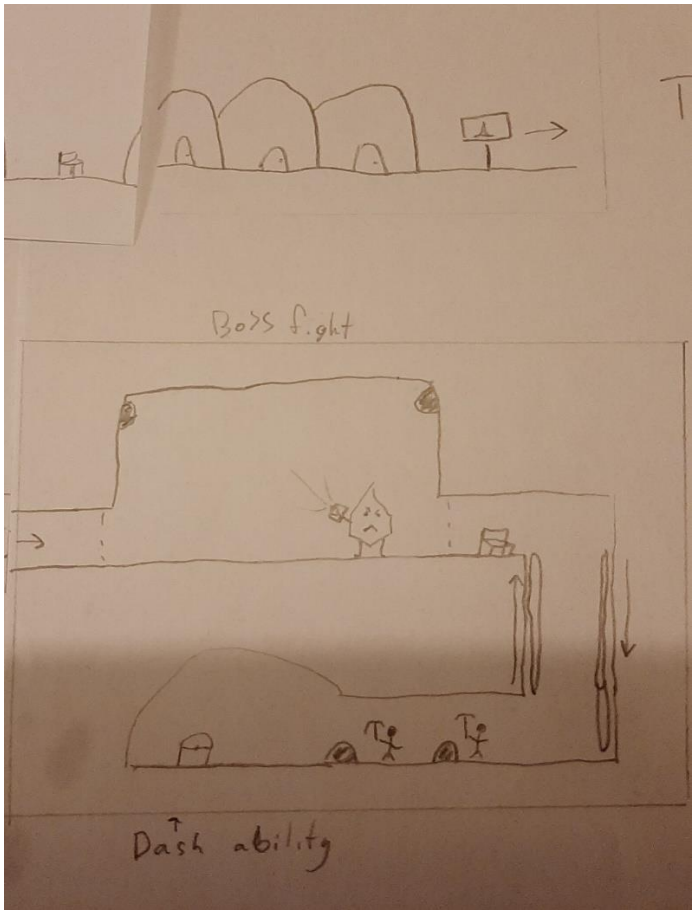


Figure 12 Paper Prototype of Boss fight of area 1

3.8 Story board Paper Prototype

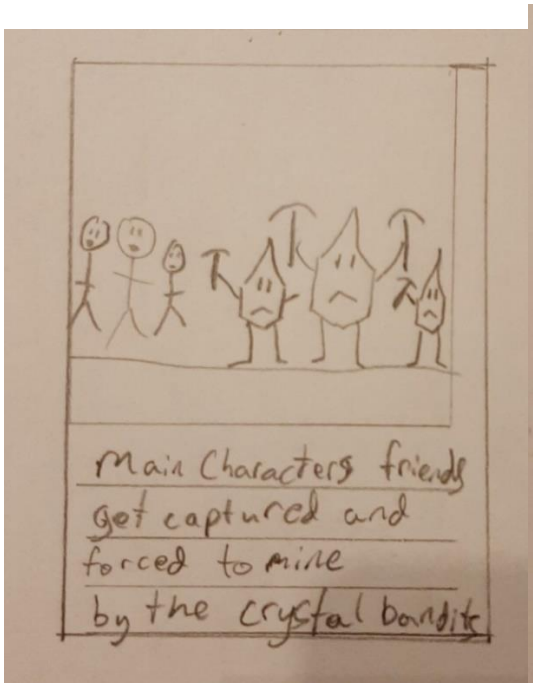


Figure 13 Story board 1

Figure 14 Story board 2

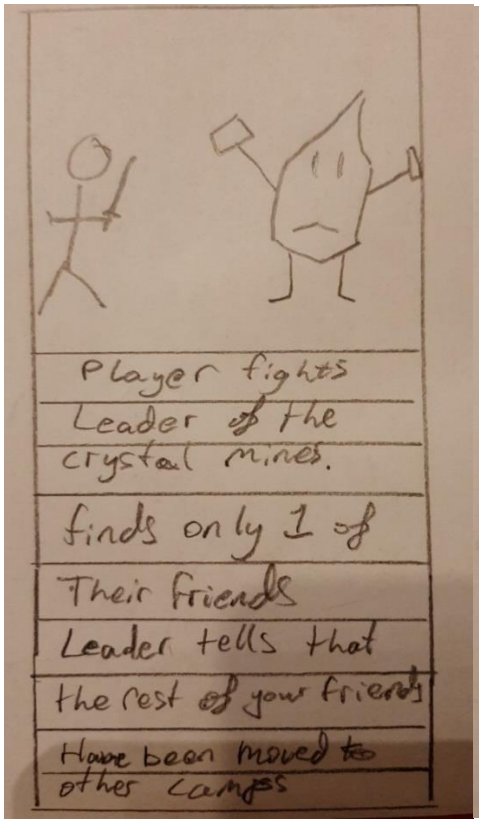


Figure 15 Story Board 3

Figure 16 Story Board 4

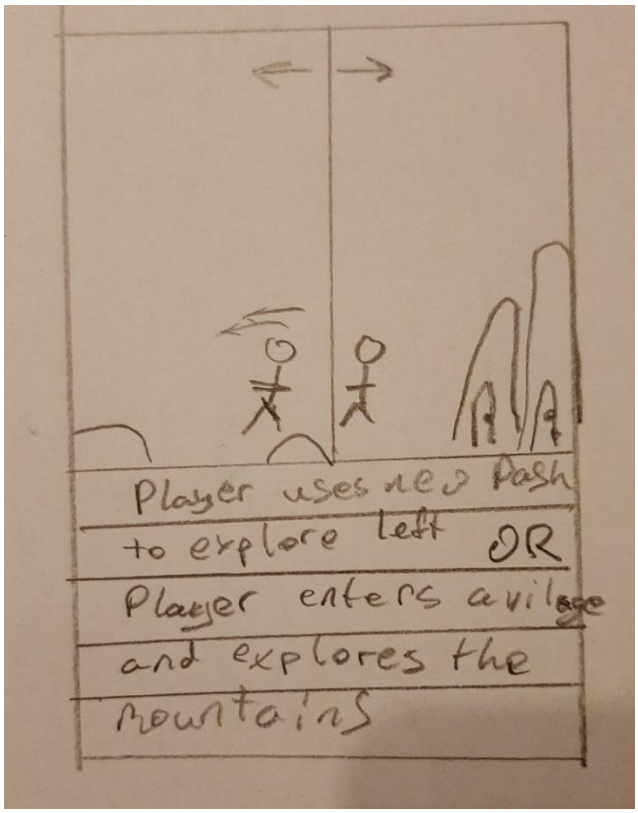


Figure 17 Story Board 5

3.8 Potential Difficulties

This is the author's first complex 2D game. They have only done very basic 2D games without any storyline or characters or item pickups. Therefore, the author could experience difficulties with coding enemies, coding how the camera follows the player and the bounds of where the camera can go. There could also be problems with save files and autosave features.

3.9 Conclusion

The author will create a 2D Metroidvania game with basic movement controls, health bars, sword attacks, story progression, item pickups, enemies, 3 different areas to explore (possibly 4), a boss fight for each area (this might be too much for the time frame), autosave features and save files.

4 Design Chapter

This chapter will focus on what is important to consider in the design process: the system architecture and the UI (User Interface). This project was developed using the Unity Engine, Visual Studio Code and Photoshop. In the System Architecture section, the user's interaction with the application is described. In the User Interface Design section, the UI sketches and high-fidelity versions of the sketches are shown. Inspiration came from similar games. In the Proposed Map Layout section, the map sketches and high-fidelity versions are shown.

4.1 System Architecture

User Flow

When designing a game, the developer must think about how the user will move through the application. The developer must understand what the user will find irritating but also what the user will like and appreciate.

Figure 1 shows how the user will interact with the main menu. The user has 3 options to choose from:

- The player can select the "Play Game" button, this starts the game at the first level.
- The player can select the "Settings" button, this opens the settings menu where the user can choose to change the audio or video settings.
- The player can select the "Quit" button, this closes the application.

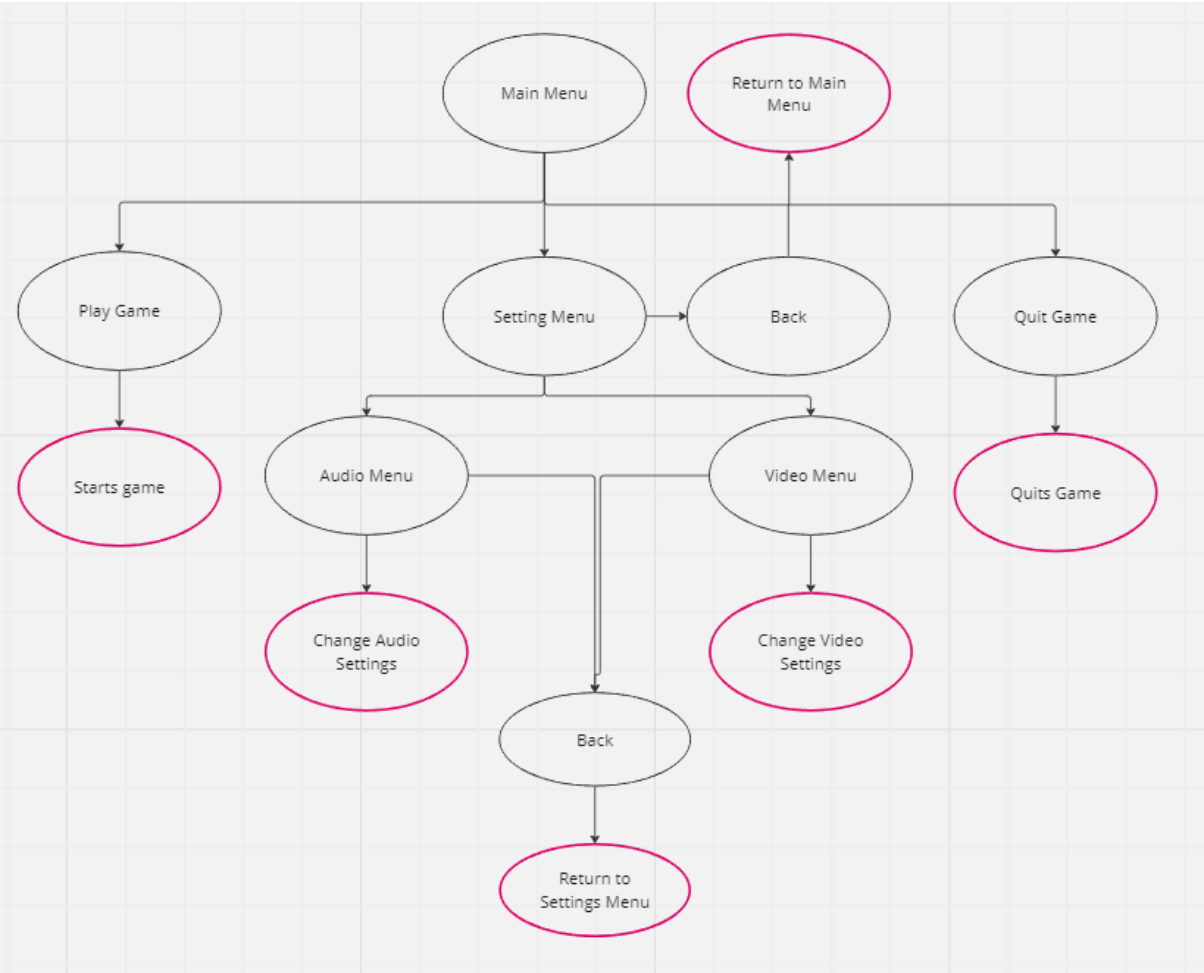


Figure 18 Main Menu User Flow Diagram

Figure 2 shows how the user will interact with the pause menu. When the player presses the “Pause” button the game is paused. The player is then shown the pause menu. The player has 3 options to choose from:

- The “Resume” button, this resumes the game and hides the pause menu.
- The “Settings” button, this displays the settings menu where the user can change the audio or video settings.
- The “Quit” button, this returns the player to the main menu.

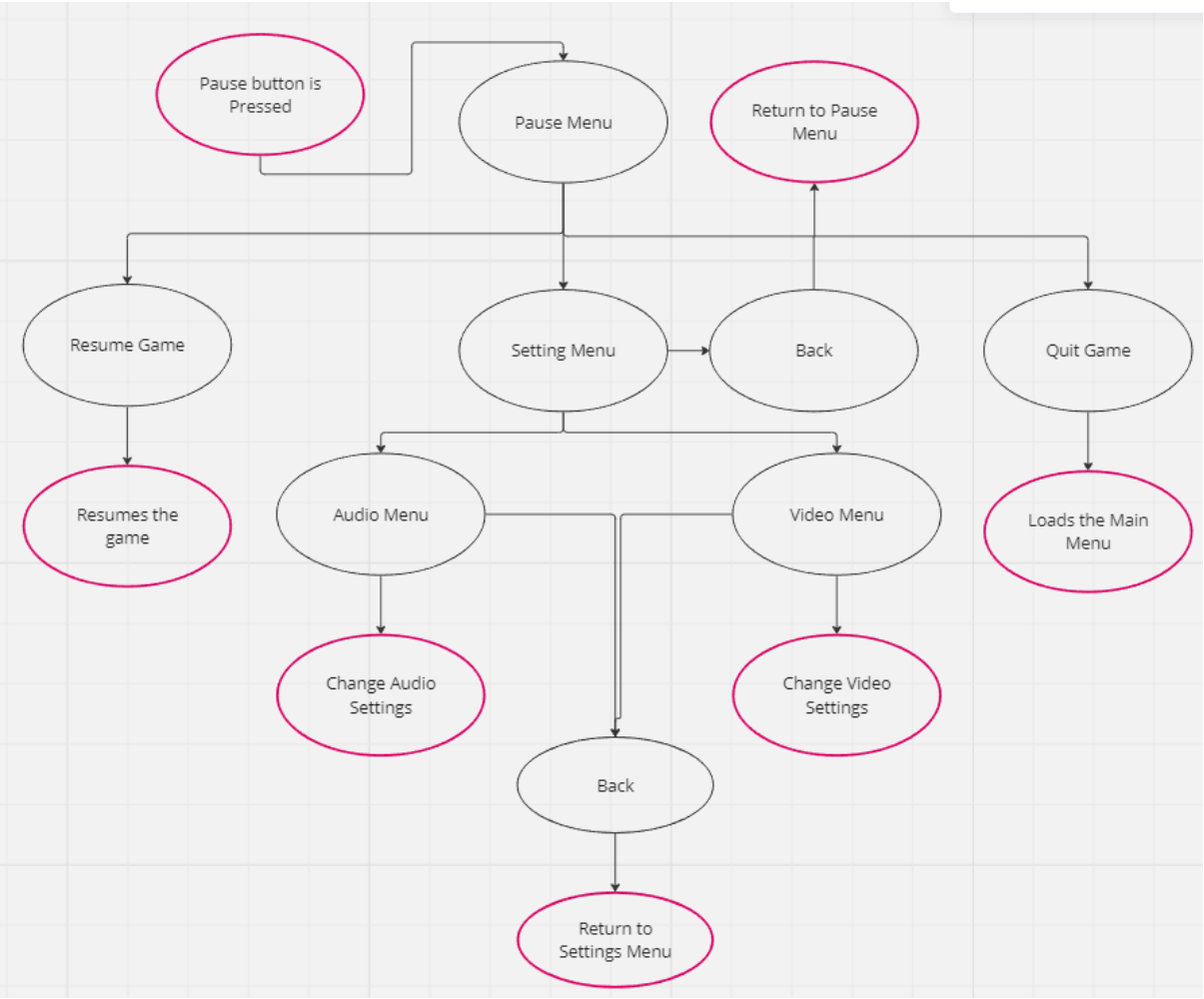


Figure 19 Pause Menu User Flow Diagram

Figure 3 shows how the user will interact with the death menu. This is shown when the player dies. The player has two options:

The “Quit” button, this returns the user to the main menu.

The “Restart” button, this will restart the current scene.

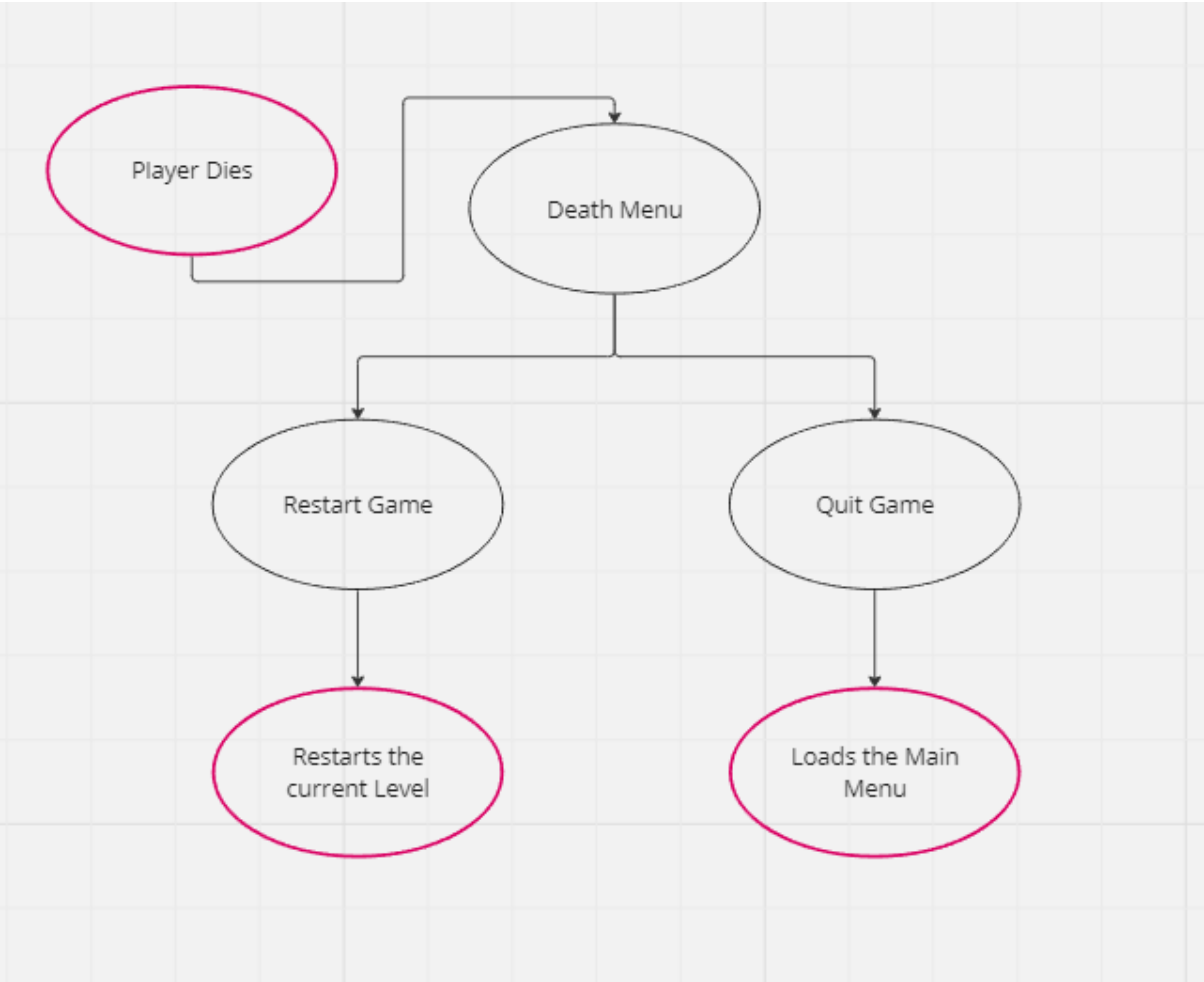


Figure 20 Death Menu User Flow Diagram

4.2 User Interface Design

Figures 4 – 9 show the author's sketches for what they want the menus to look like.

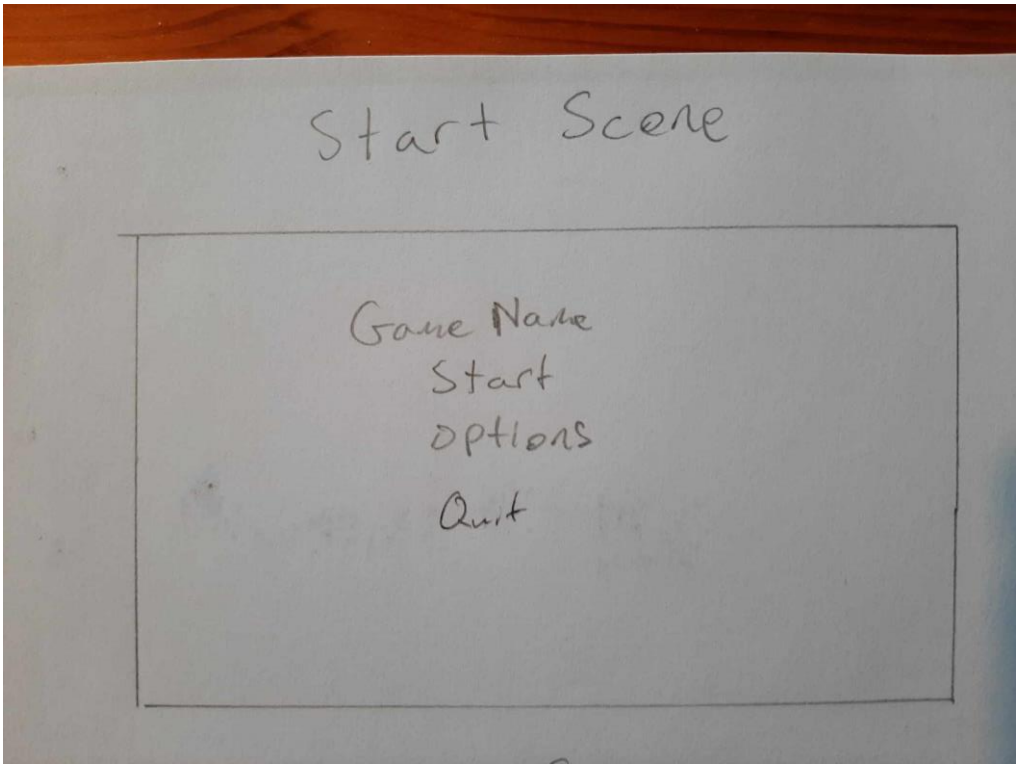


Figure 21 Start Menu sketch

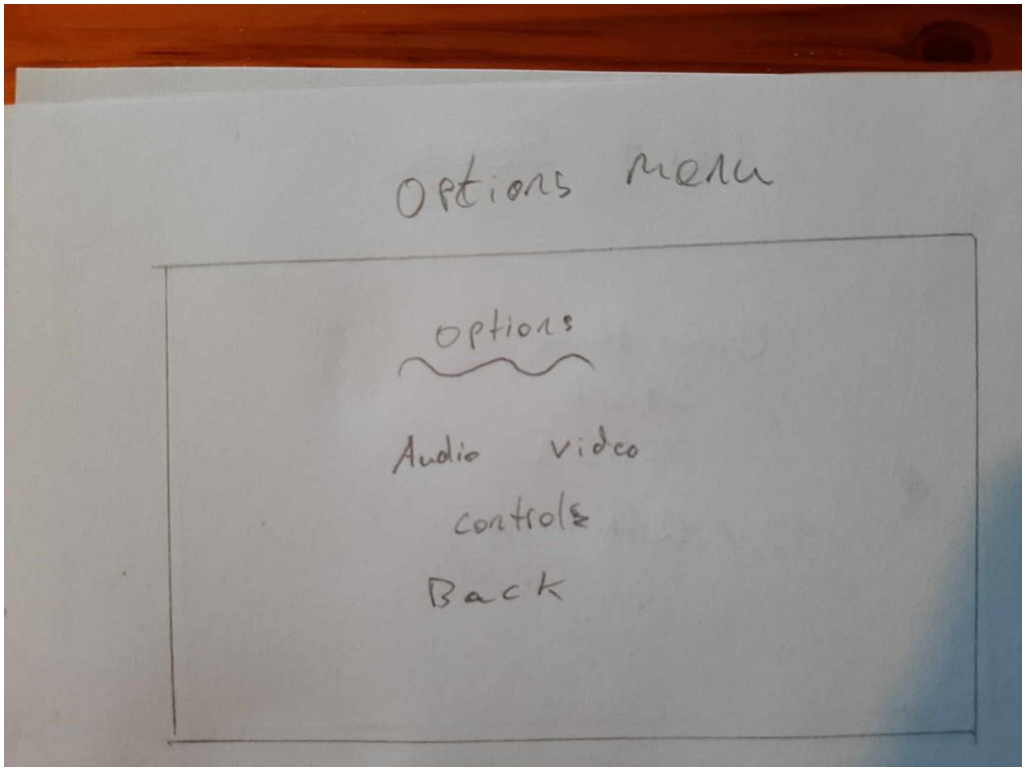


Figure 22 Options Menu sketch

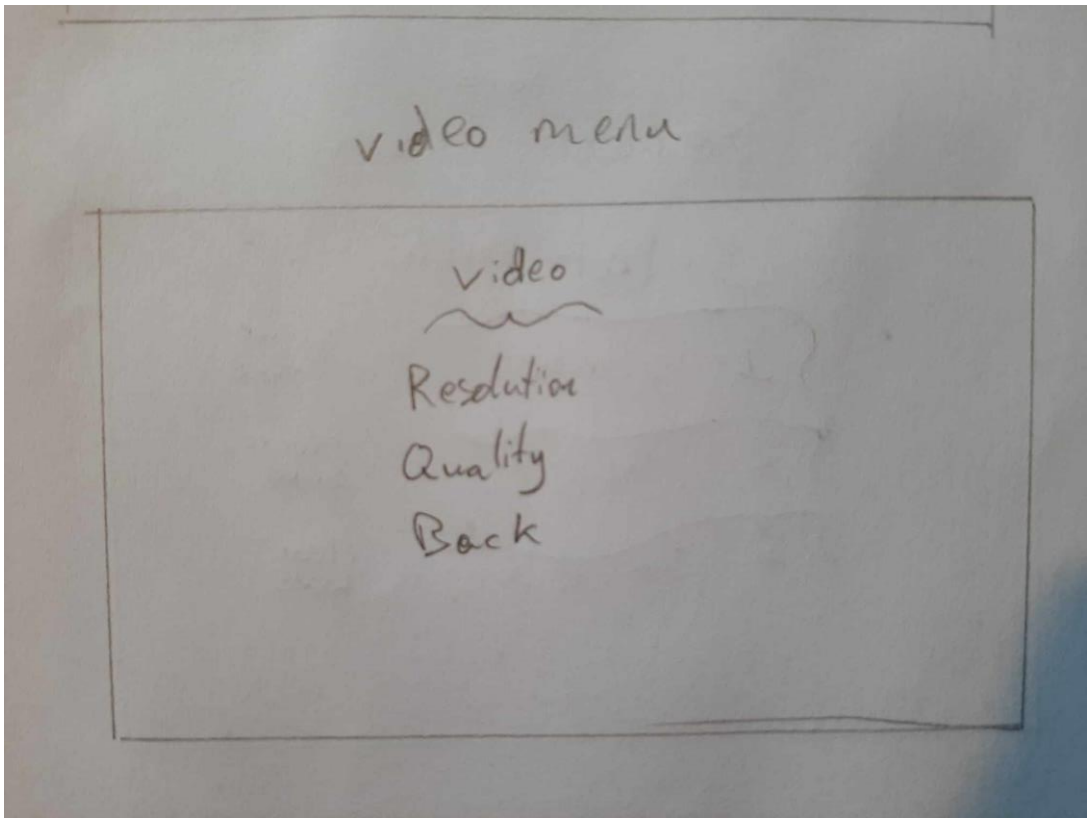


Figure 23 Video Menu sketch

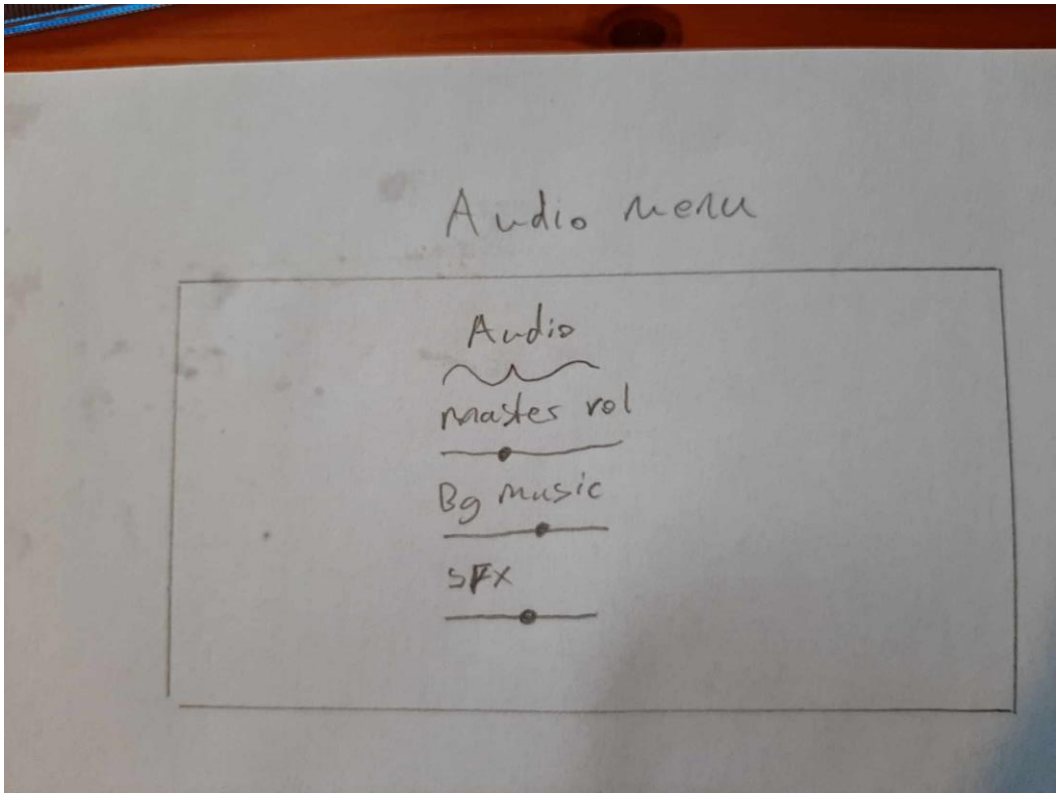


Figure 24 Audio Menu sketch

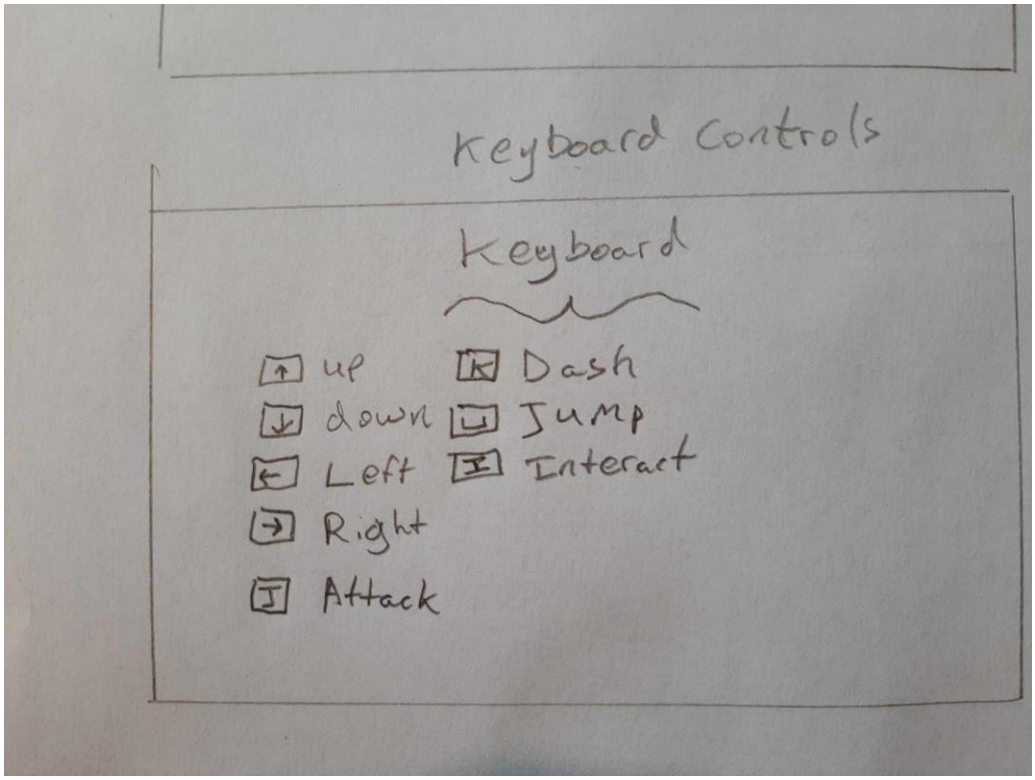


Figure 25 Keyboard Menu sketch

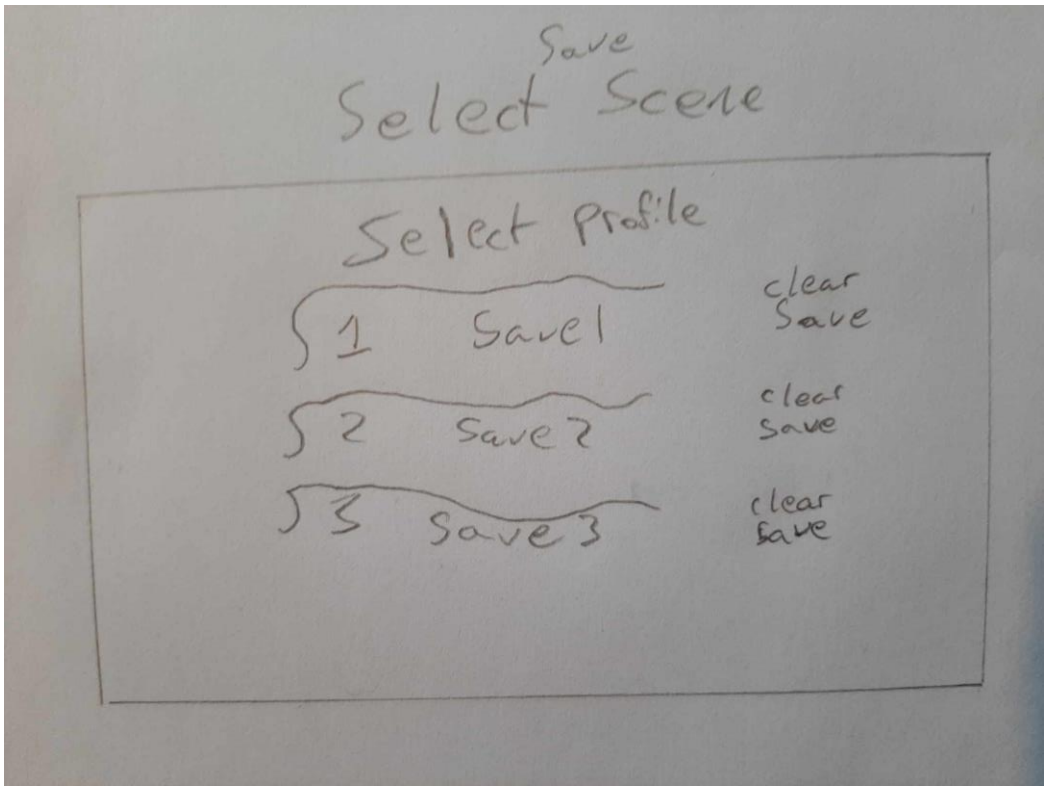


Figure 26 Select Save Menu sketch.

Figures 10 – 15 show the author's developed versions of the menus.

Figure 27 Main Menu

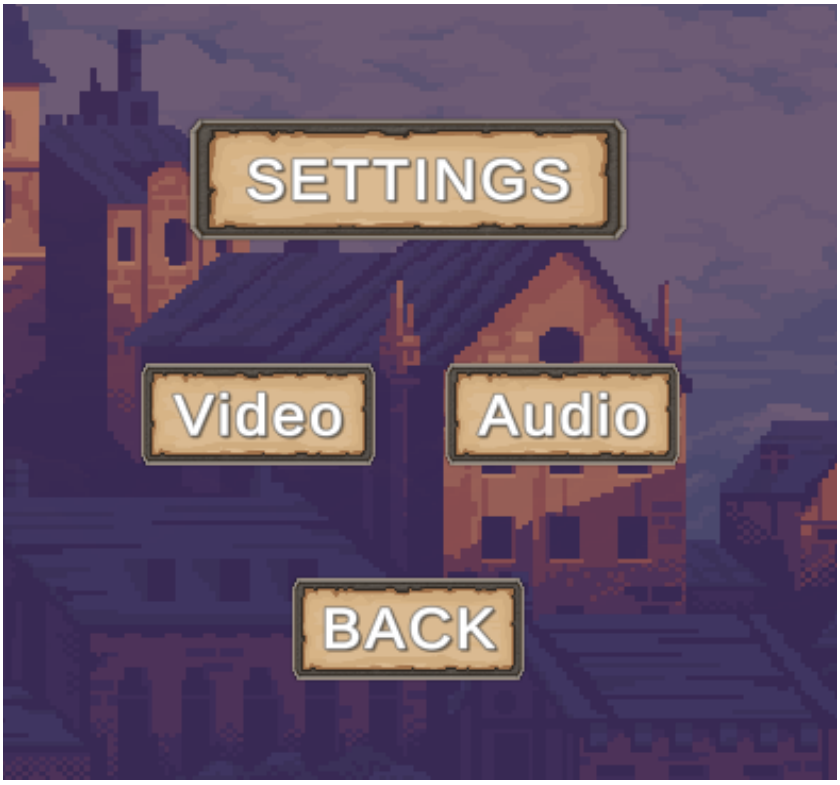


Figure 28 Settings Menu

Figure 29 Settings Menu Audio

Figure 30 Settings Menu Video

Figure 31 Pause Menu

Figure 32 Death Menu

4.2.1 Inspiration

Figure 33: Screenshot taken by the author showing the Menu UI

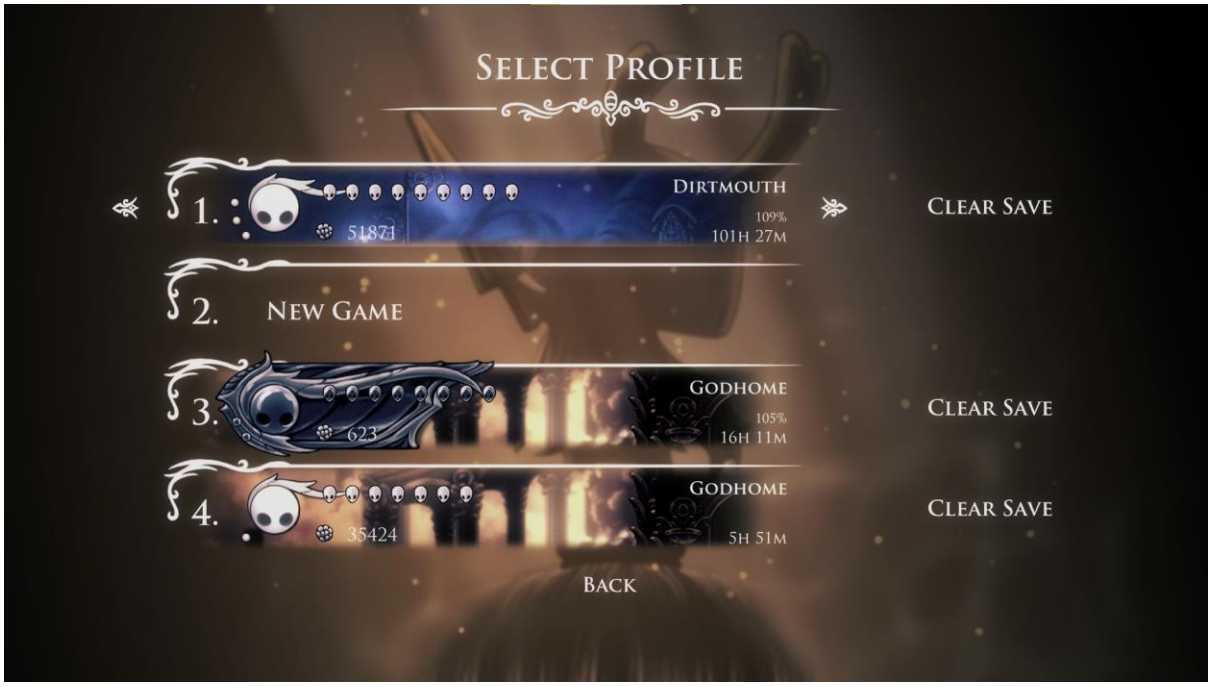


Figure 34: Screenshot taken by the author showing the save files

In **Figure 18** we can see that there are different layers to even a 2D game. This adds depth to a game where you can only move up down, left and right as seen in **Figure 18** The background labelled **A** is the furthest back and will appear behind all other objects in the game. While the background labelled **B** is on top of **A** and will appear on top of that, the area labelled **C** is the plane that the player is on and can explore. The layer Labelled **D** is in front of the player Level and the player will be hidden behind objects that are on this layer. There is also a layer in front of **D** that is used occasionally.

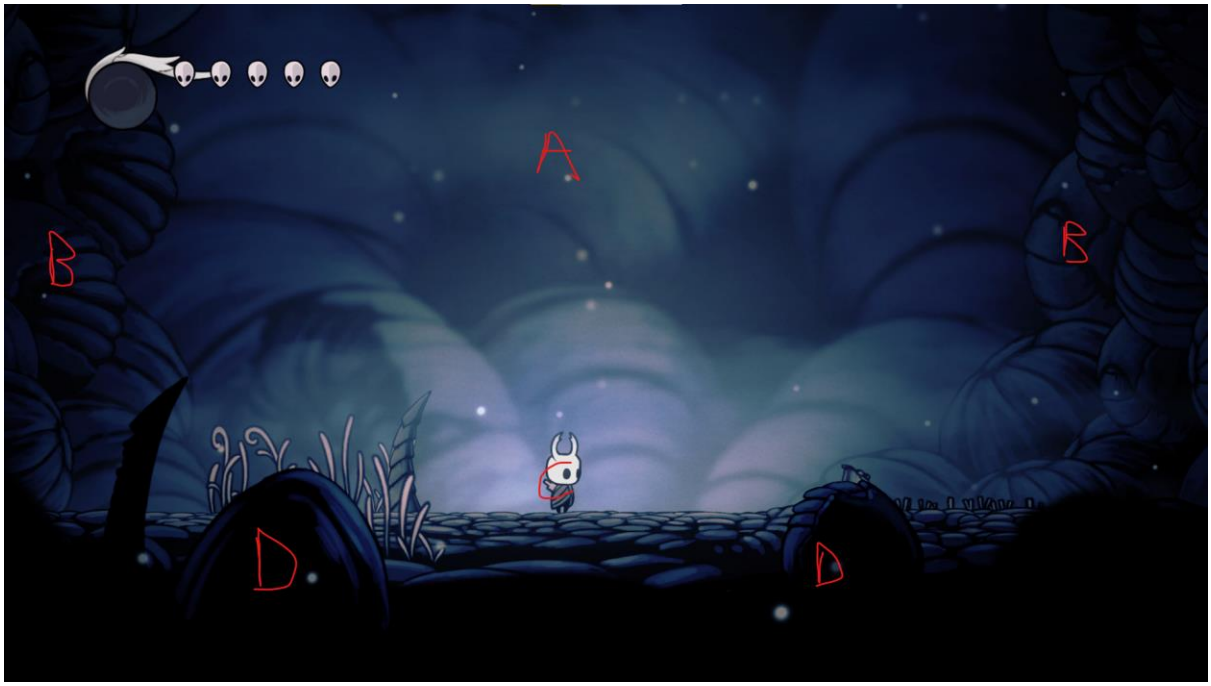


Figure 35: Screenshot taken by the author showing an example of how layers are used in Hollow Knight

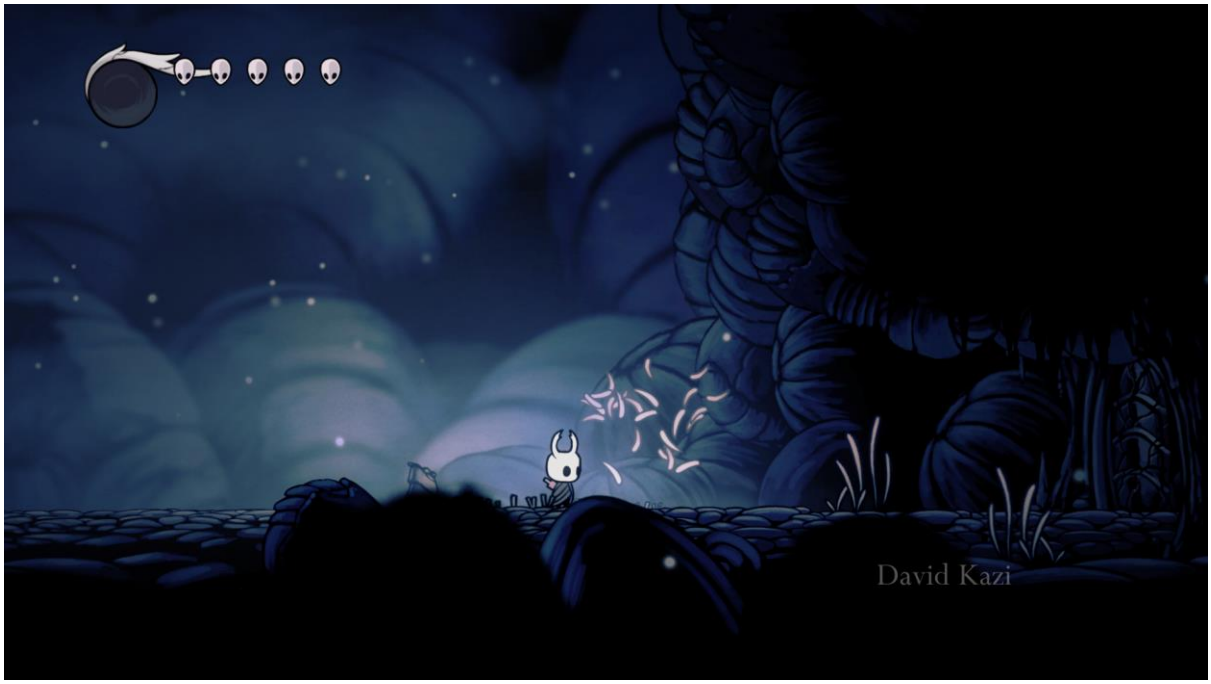


Figure 36: Showing interactable environment from Hollow Knight

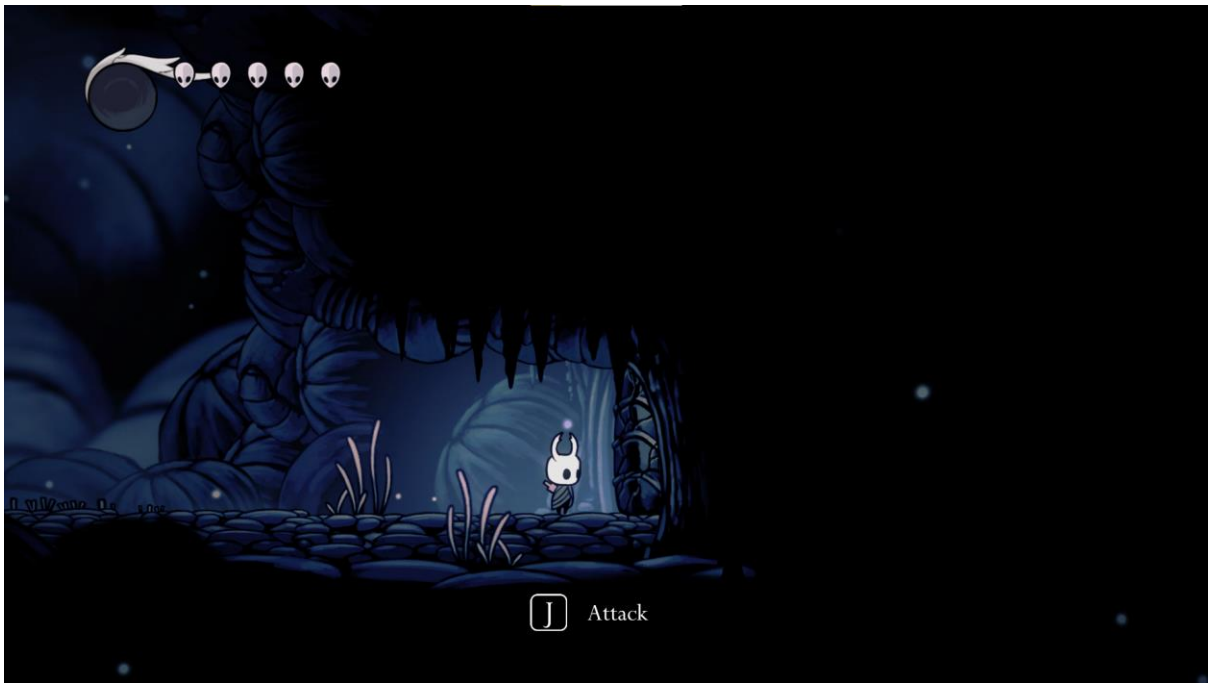


Figure 37: Showing interactable environment from Hollow Knight

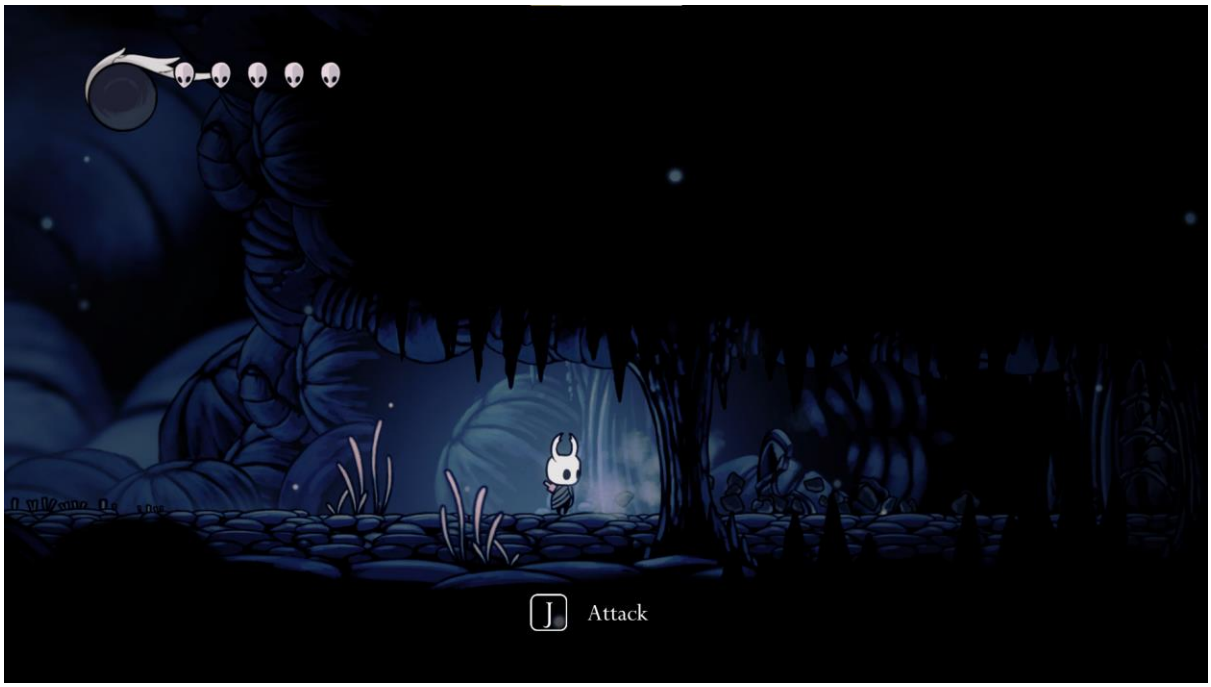


Figure 38: Showing interactable environment from Hollow Knight

Figure 39: Full map of every area in Hollow Knight

4.3 Proposed Map Layout

4.3.1 The Crystal Village 1

In **Figures 23 – 24** the developed versions of The Crystal Village 1 are shown.

Figure 40 The Crystal Village 1

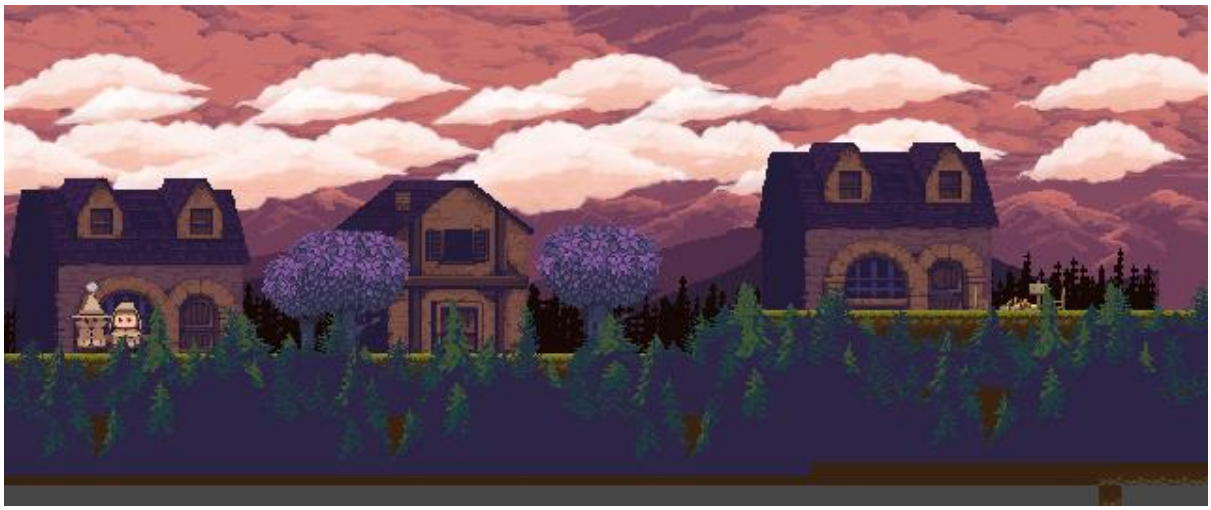


Figure 41 The Crystal Village 1

4.3.2 The Crystal Village 2

In **Figures 25 – 26** the developed versions of The Crystal Village 2 are shown.

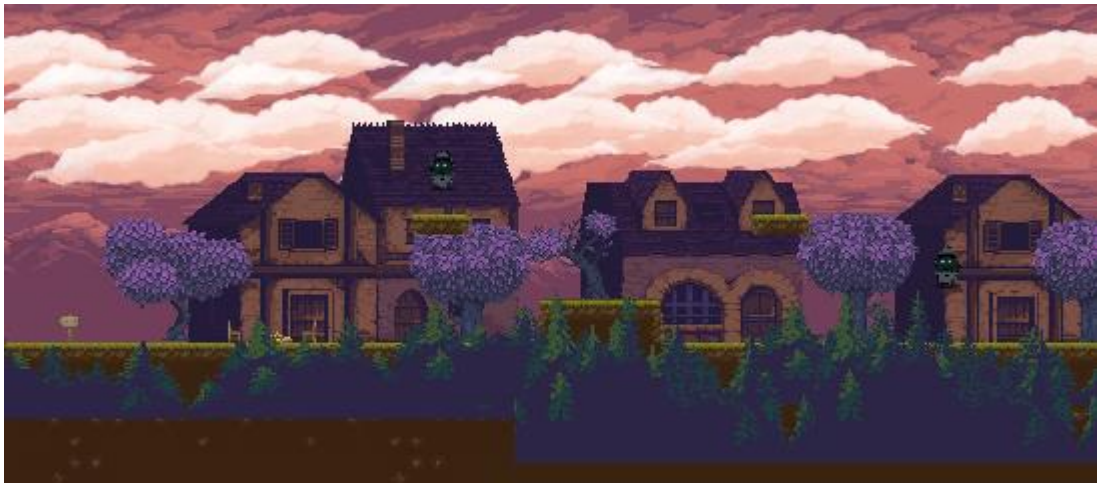


Figure 42 The Crystal Village 2

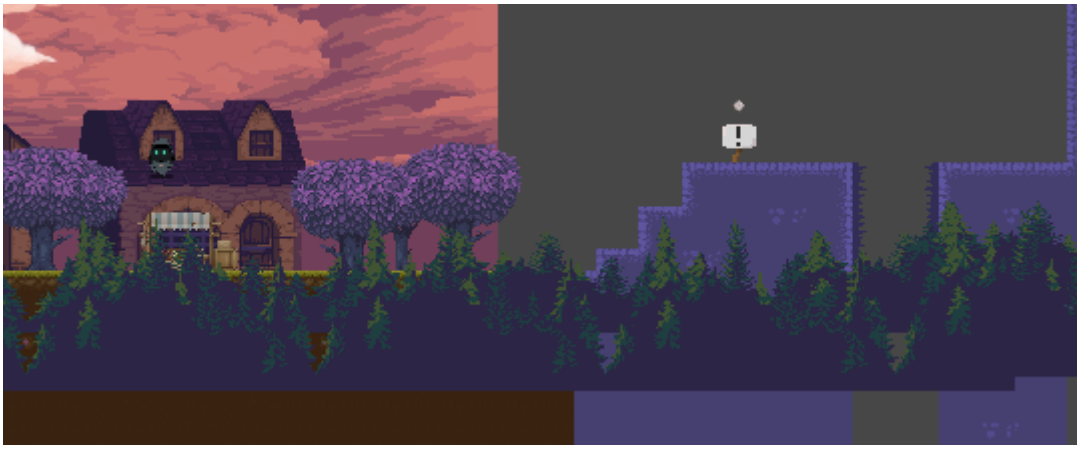


Figure 43 The Crystal Village 2

4.3.3 The Crystal Mines

The **Figures 27 – 29** show the developed versions of the map.

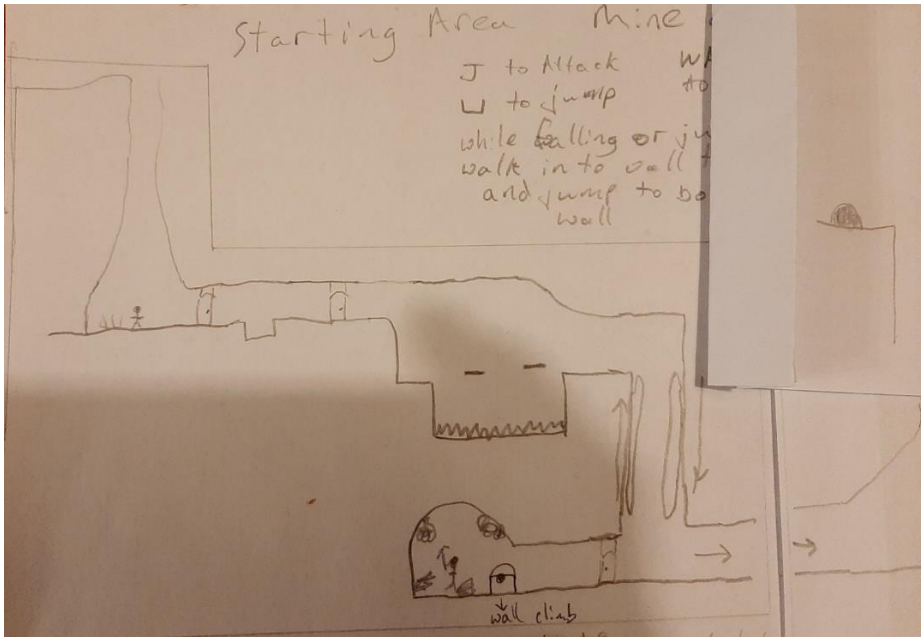


Figure 44 Section 1 of The Crystal Mines

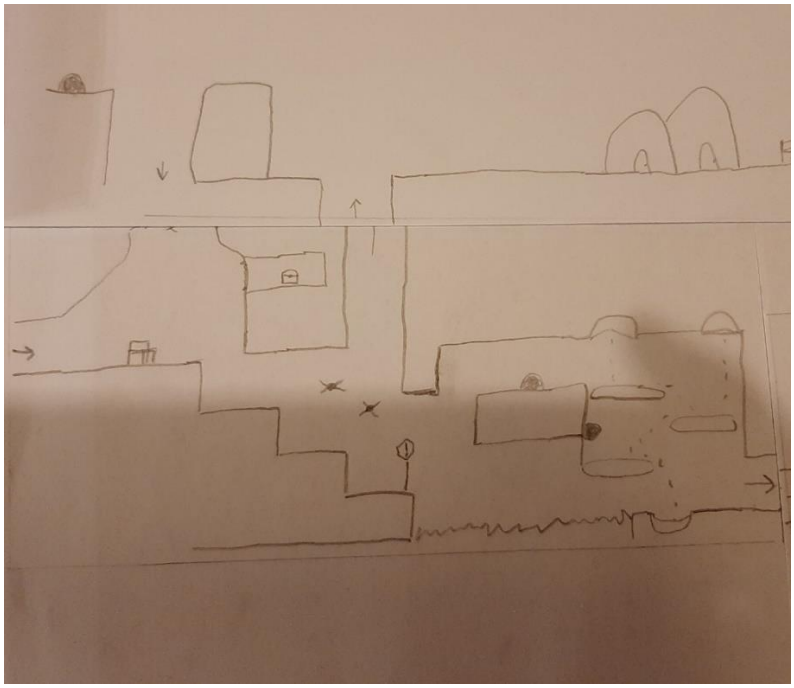


Figure 45 Section 2 of The Crystal Mines

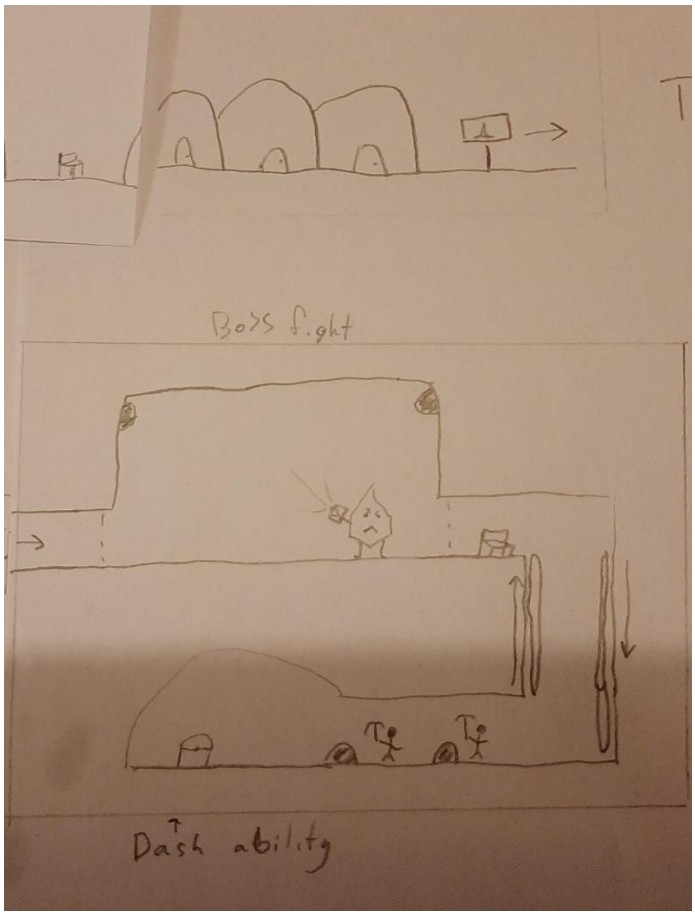


Figure 46 Section 3 of The Crystal Mines

The **Figures 30 – 33** show the developed versions of the map.

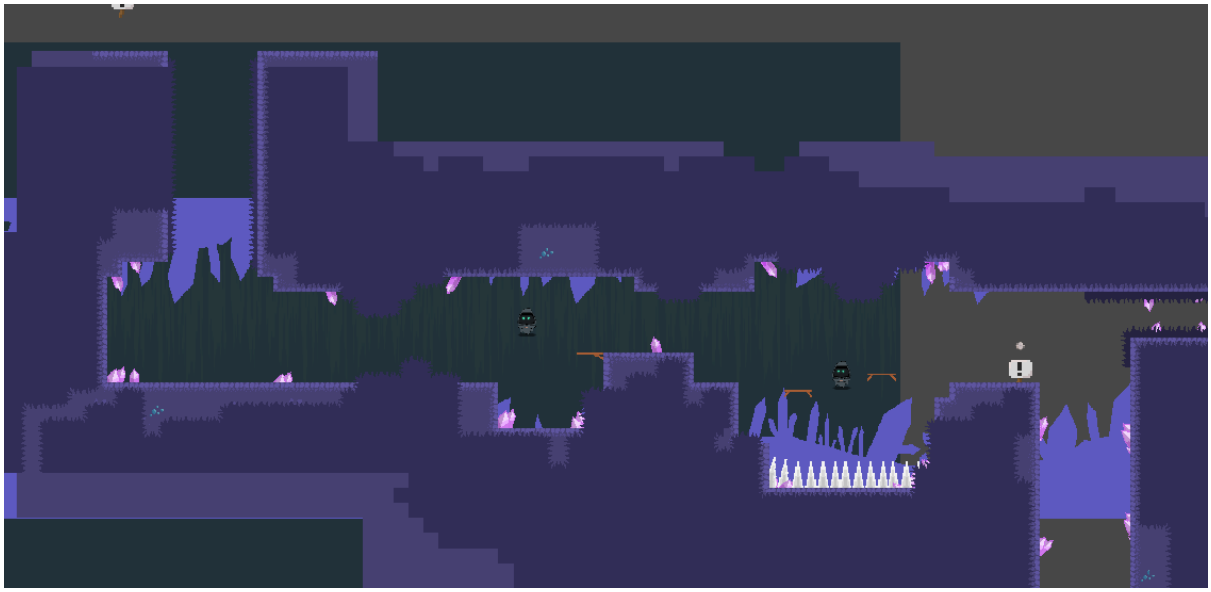


Figure 47 Crystal Mines Section 1

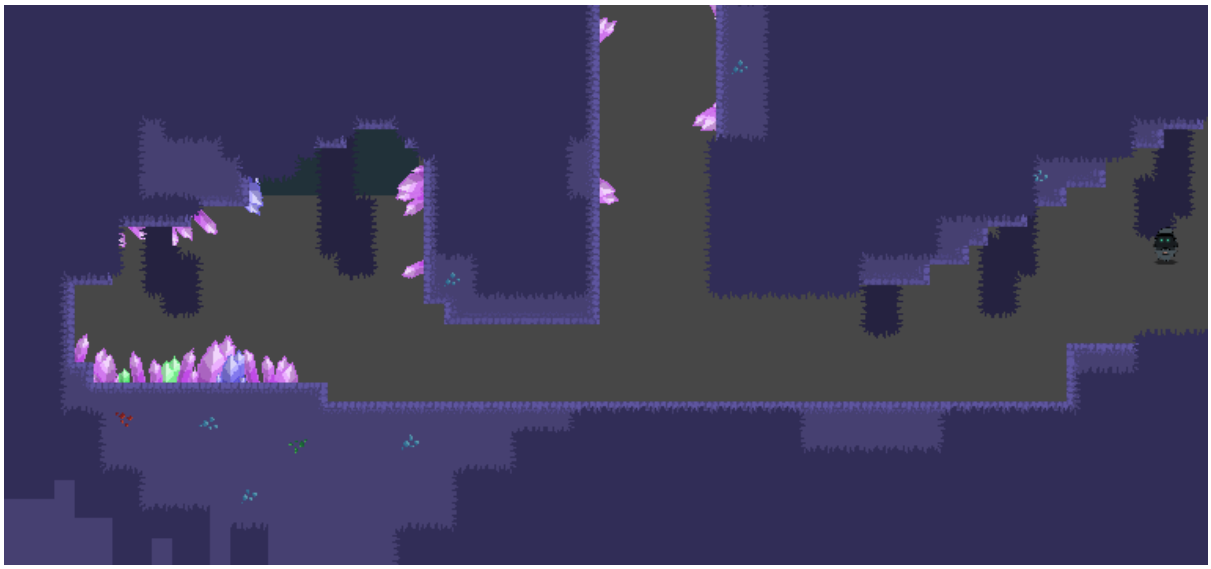


Figure 48 Crystal Mines Section 2

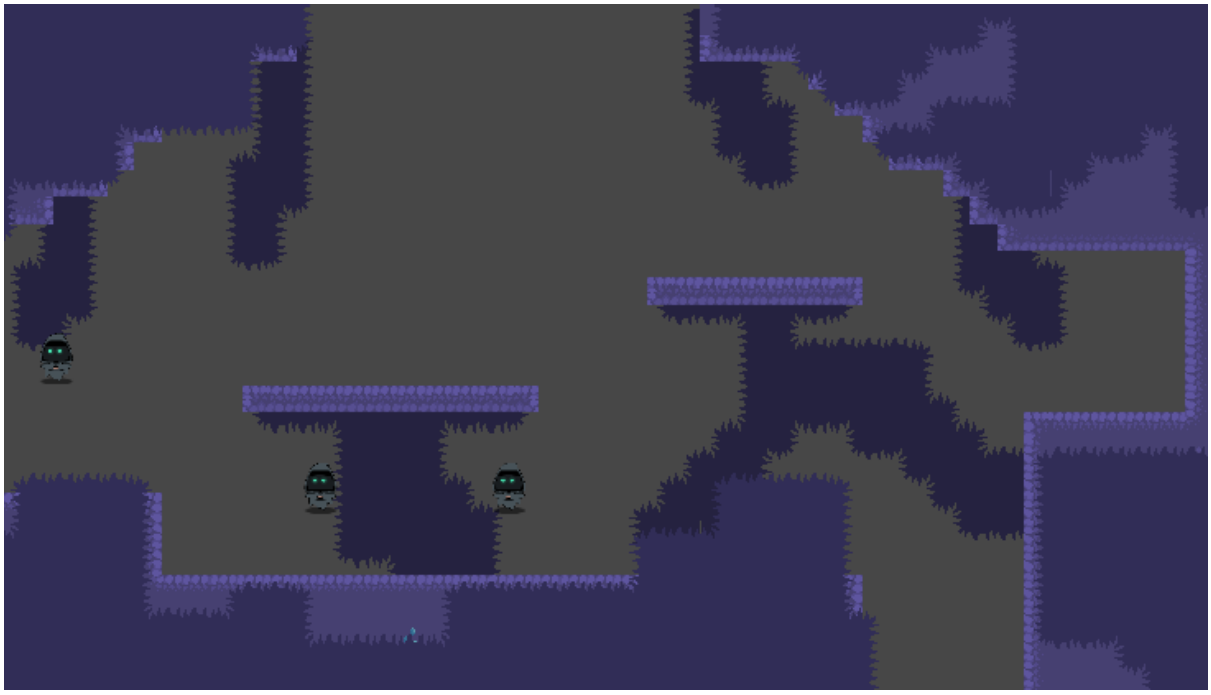


Figure 49 Crystal Mines Section 3

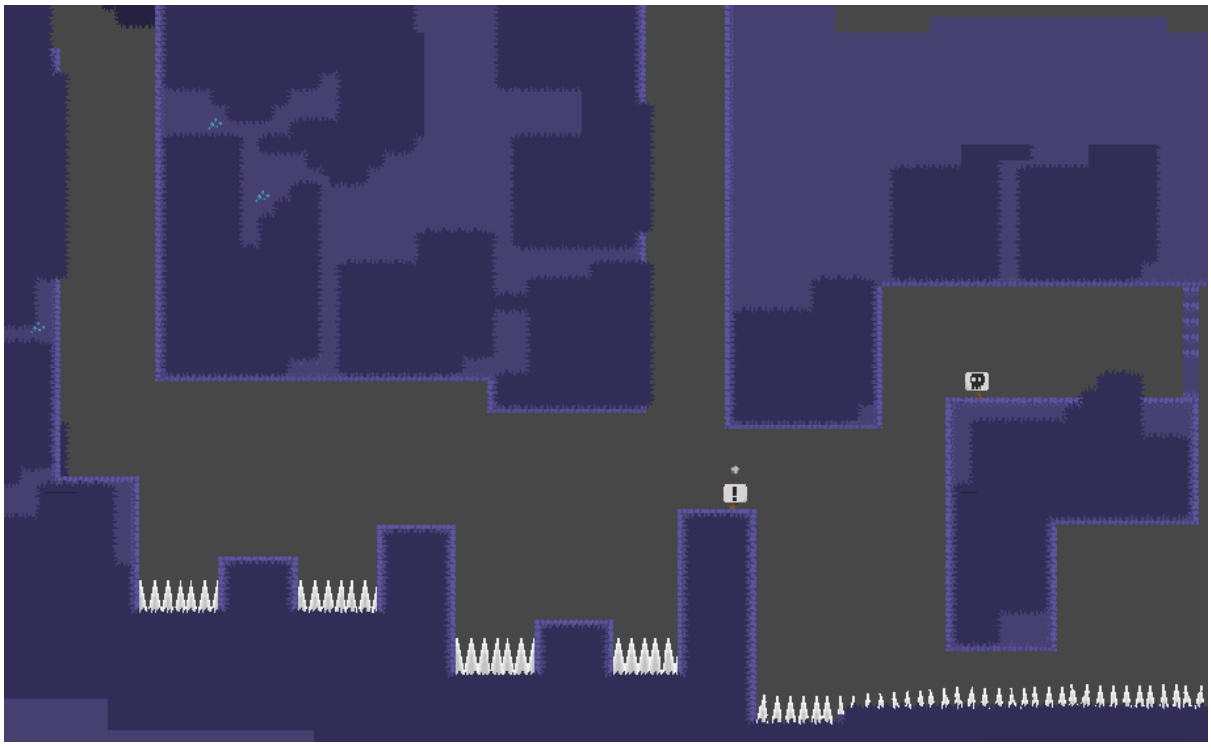


Figure 50 Crystal Mines Section 4

4.3.4 The Crystal Mines Boss

The **Figure 34** shows the developed map of the crystal mine boss room.

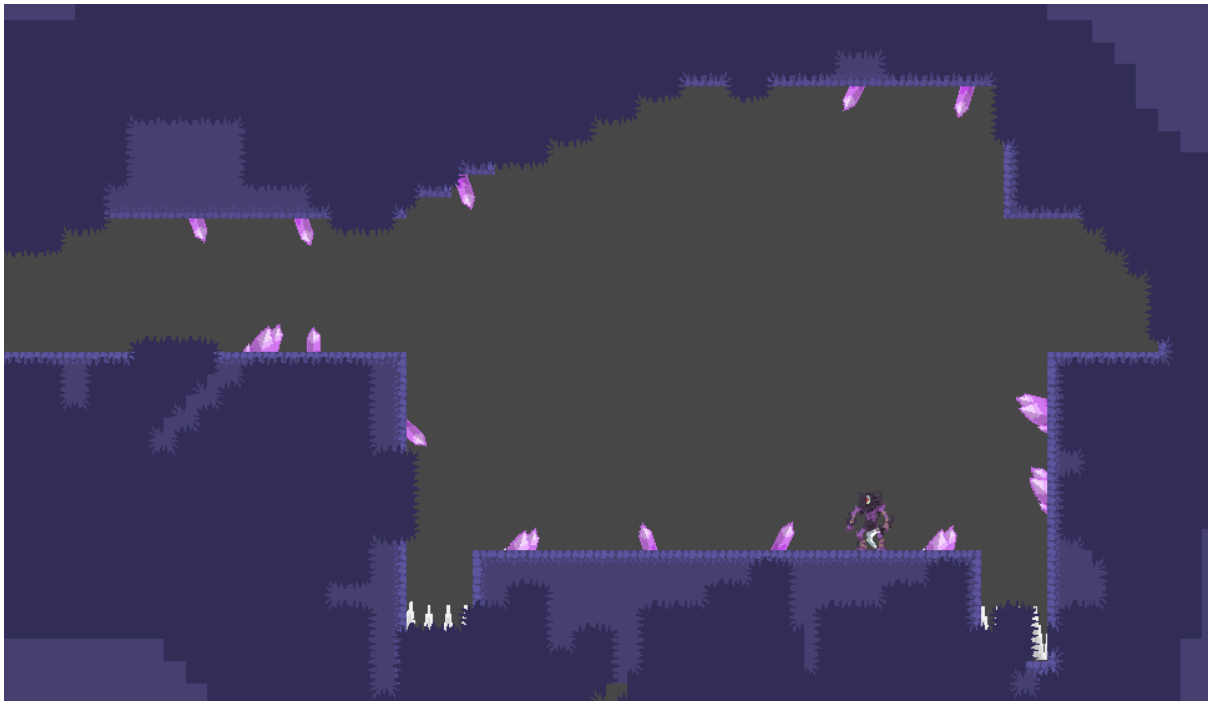


Figure 51 Crystal Mines Boss

5 Implementation Chapter

5.1 Introduction

This chapter will go into detail on all the functions created by the author and how the requirements are implemented into the game.

5.2 IsWalled() function

This function will return a Boolean variable, using:

“wallCheck” - an empty game object that is placed on the side of the player.

“wallLayer” - a layer mask of all the walls with the Layer “Wall”, and

“radOfCircle” - the radius used to check if the player and the wall are touching this circle, this variable is also used in the “IsGrounded()” function.

```
[Header("Wall Details")]
[SerializeField]private Transform wallCheck;
[SerializeField]private LayerMask wallLayer;
```

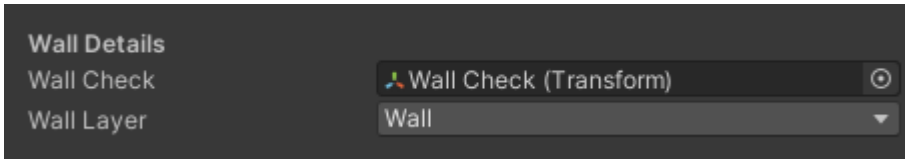
```
private bool IsWalled()
{
    return Physics2D.OverlapCircle(wallCheck.position, radOfCircle, wallLayer);
}
```

Figure 52 IsWalled() function

This is showing the “Walls” object tagged with the layer “Wall”:

The “Wall Check” object is part of the player similar to “Ground Check”.

In the inspector the Wall Layer is set on “Wall” and the Wall Check object selected.



5.3 IsGrounded() Function

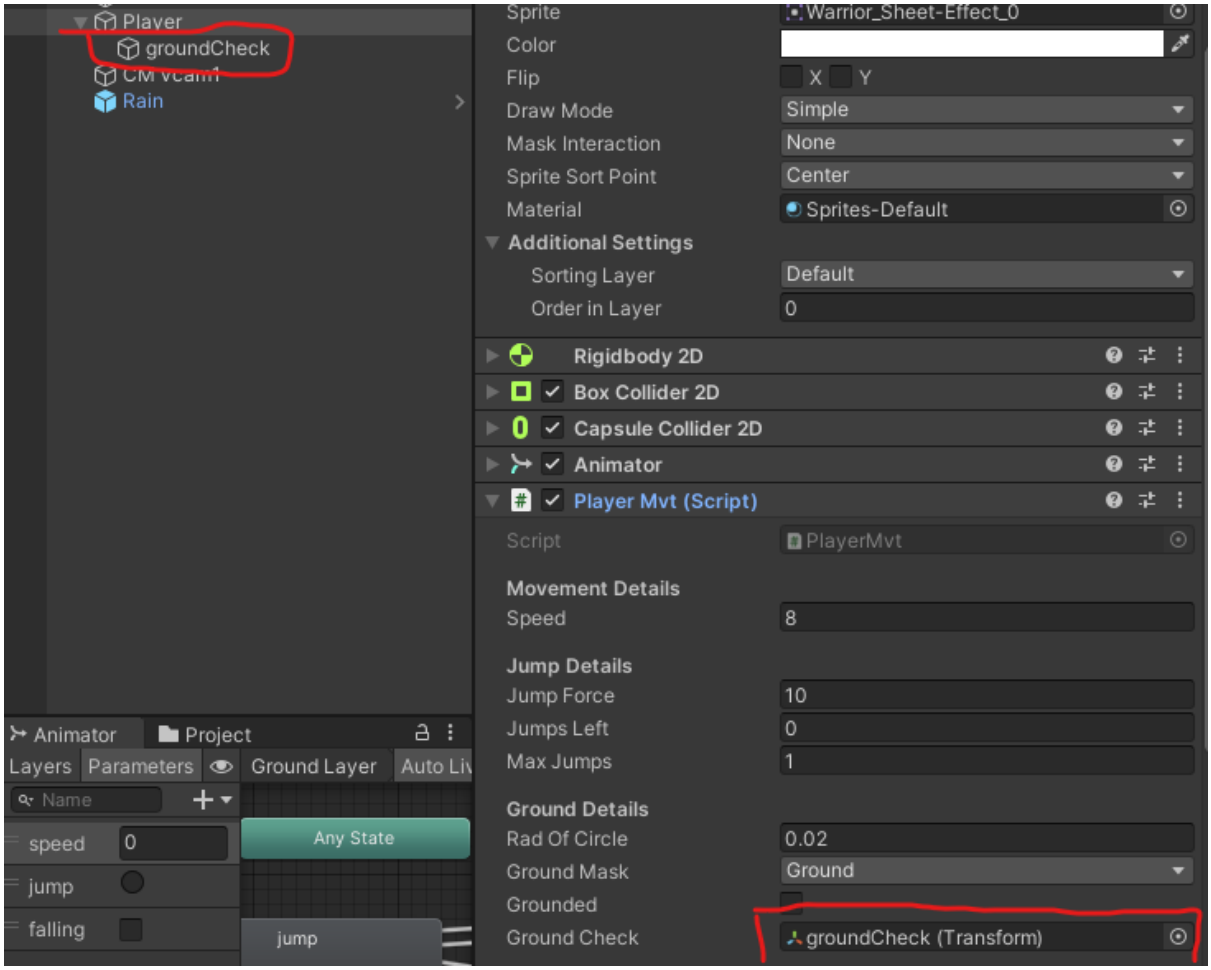
This function returns a Boolean variable, using the variables “groundCheck” (just under the player’s collider), “radOfCircle” (the radius of that circle), and the layer mask “groundMask”. The ground mask is applied to all objects that are in the layer “Ground”. The Boolean returns true if the ground mask overlaps with the circle under the player’s collider and false if it doesn’t.


```
private bool IsGrounded()
{
    //drawing a small circle under the rigid body to check if its touching the ground mask. if it is
    return true. if not return false.
    return Physics2D.OverlapCircle(groundCheck.position, radOfCircle, groundMask);


}
```

Figure 53 IsGrounded() Function

The author also draws a Gizmo which can be seen in the editor but not in the game. This makes it easier to see where the ground is intersecting with the player.

```
//used to draw a gizmo that is visible to the editor but not in game.
private void OnDrawGizmos()
{
    Gizmos.DrawSphere(groundCheck.position, radOfCircle);
}
```


Player
groundCheck
Rain
Sprite
Warrior_Sheet-Effect_0
Color
Flip
X
Y
Draw Mode
Simple
Mask Interaction
None
Sprite Sort Point
Center
Material
Sprites-Default
Additional Settings
Sorting Layer
Default
Order in Layer
0
Rigidbody 2D
Box Collider 2D
Capsule Collider 2D
Animator
Player Mvt (Script)
Script
PlayerMvt
Movement Details
Speed
8
Jump Details
Jump Force
10
Jumps Left
0
Max Jumps
1
Ground Details
Rad Of Circle
0.02
Ground Mask
Ground
Grounded
Ground Check
groundCheck (Transform)
Animator
Project
Layers
Parameters
Ground Layer
Name
speed
0
jump
falling
Any State
jump

5.4 WallSlide() Function

This function makes the player slowly slide down a wall when they move into it if they are not on the ground already. The “Y” velocity is Clamped to not exceed a certain speed. This function uses the “isWallSliding” Boolean and “wallSlidingSpeed” float which controlls the speed the player will fall at.

The player will slide down the wall as long as they are moving towards the wall.

```
private bool isWallSliding;
private float wallSlidingSpeed = 2f;
```

```
private void WallSlide()
{
    if(IsWalled() && !IsGrounded() && (direction > 0f || direction < 0f))
    {
        isWallSliding = true;
        rb.velocity = new Vector2(rb.velocity.x, Mathf.Clamp(rb.velocity.y,
        -wallSlidingSpeed, float.MaxValue));

    }
    else
    {
        isWallSliding = false;

    }
}
```

Figure 54 WallSlide() function

5.5 Flip() Function

This function will change the direction of the object it is used on. The author changes the scale of the object's X value by multiplying it by -1 which just flips the direction the object is facing.

Below shows how the flip function is used In the "Update()" function. If the object is not facing to the right, and the direction value is greater than 0, then it flips the object to the opposite direction and vice versa.

```
//method used to change the direction a rigid body is facing
private void Flip()
{
        facingRight = !facingRight;

        Vector3 theScale = transform.localScale;
        theScale.x *= -1;
        transform.localScale = theScale;
}
```

Figure 55 Flip() function Version 1

```
//making sure the player is facing the correct direction.
if (!facingRight && direction > 0f)
{
    Flip();
}
//making sure the player is facing the correct direction.
else if(facingRight && direction < 0f)
{
    Flip();
}
```

Figure 56 ChangeDirection() function

5.5.1 Flip() Function Update

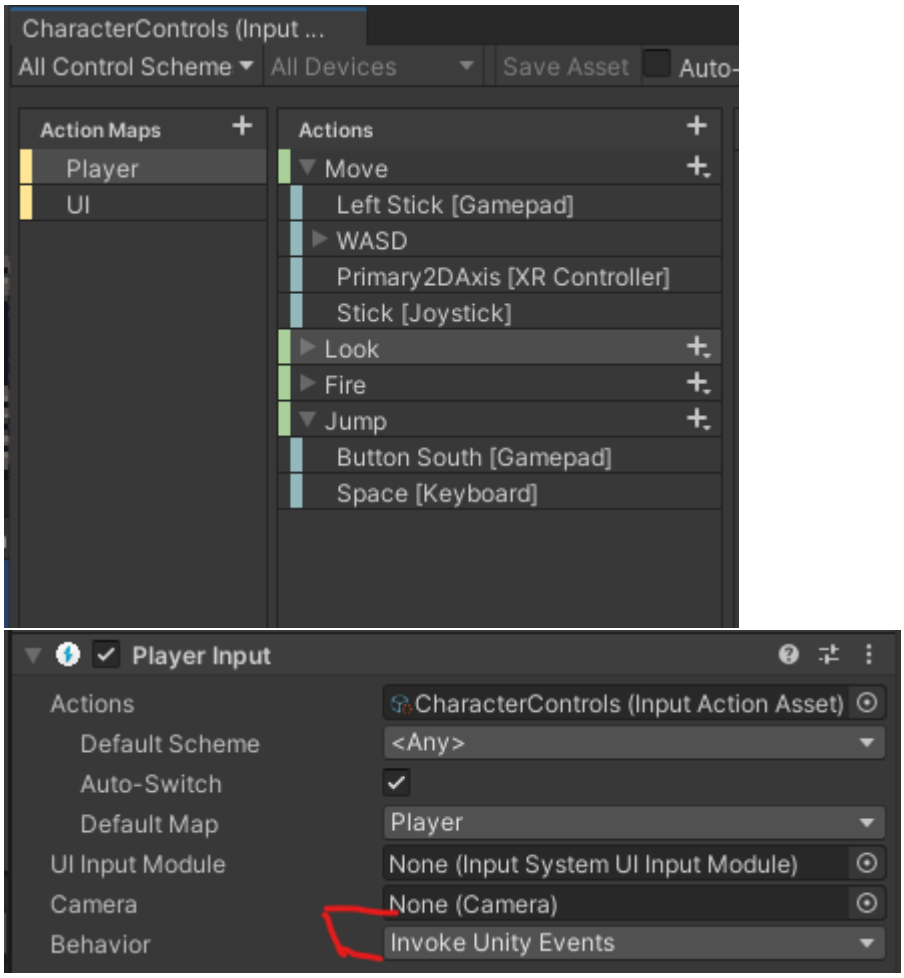
With the "Cinemachine" framing transposer the camera will lean to one side give the player a bigger view on the side they are facing. However, when flipping the character as shown above, the facing direction is not being changed. The new way of flipping the character rotates them on their Y rotation.

```
protected virtual void Flip()
{
    //OLD VERSION OF CHANGING DIRECTION
    //this made the camera not work correctly.

    // facingRight = !facingRight;

    // Vector3 theScale = transform.localScale;
    // theScale.x *= -1;
    // transform.localScale = theScale;
    // cameraFollowObject.Turn();

    if(facingRight)
    {
        Vector2 rotator = new Vector2(transform.rotation.x, 180f);
        transform.rotation = Quaternion.Euler(rotator);
        facingRight = !facingRight;
    }
    else
    {
        Vector2 rotator = new Vector2(transform.rotation.x, 0f);
        transform.rotation = Quaternion.Euler(rotator);
        facingRight = !facingRight;
    }

}
```

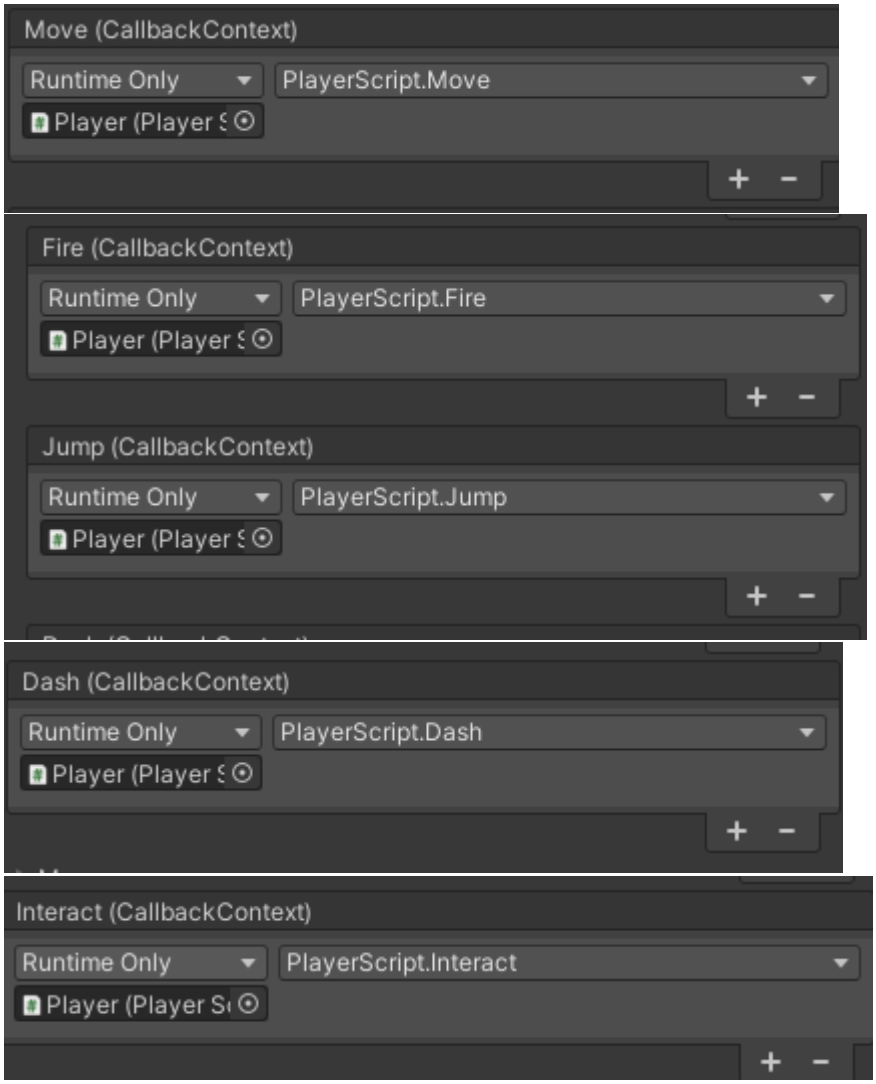
Figure 57 Flip() function Version 2

5.6 Player Movement

The author uses Unity's new Input System for keyboard and mouse, and Gamepad. On keyboard the player will use the "A" and "D" keys to move direction, and the "Space" key to Jump. On Gamepad the player will use the left joystick to move and the button south to jump.



In the Events tab the author has their "Player.cs" script which is attached to the player. They use the "Move()", "Jump()", "Dash(), "Fire()" and "Interact()" functions. For example, when the space bar is pressed the "Jump()" function checks if the action was performed and if the player has a jump available, it will make the player jump. The Author will go into more detail in the individual Functions later in this section.



The Author created the “Move()” function which reads the x direction the player is trying to move. The direction is a value of 1, -1 or 0. If the player is not moving the value is 0. If the player is moving the value is either 1 or -1 depending on the direction. This function also starts the run animation which happens when speed is not 0, however, the jump animation takes priority over the run animation. For example, if the player jumps and is moving to the side it will not play the run animation because the jump animation is being played.

5.6.1 Fire

The Author created the “Fire()” function which first checks:

1. If the player can attack and if they can, it will use the “Attack()” function which activates the “attack1” trigger. This plays the attack animation. It is called attack1 because the author wants to have two swings of the attack if the player presses attack in quick succession, this might not be implemented because of time constraints.
2. Next the “Attack()” function will put all enemies that are in the attack range when the button is pressed into an array called “hitEnemies”.
3. It will run the array through a “foreach” loop that will deal damage to all the enemies hit.

4. “Fire()” will then put the attack on a “Cooldown” where the player cannot attack again for about a second.

5.6.2 Jump

The Author created the “Jump()” function which will first check:

1. If the player can wall jump, which is something the player can only do when they are touching a wall. If true this will start a “Coroutine” called “WallBounce()” which will play the jump animation and set the player’s gravity scale to 0 so the wall jump isn’t affected by outside forces. It then pushes the player away from the wall and up into the air as if they jumped off the wall. The player’s gravity scale is then reset back to its original value.
2. If the player is not “isWallSliding” (the variable used to check if the player is sliding on the wall) and their max number of jumps is greater than the current number of jumps used, the variable “jumpForce” is applied to the player which makes the player jump. This is also where the coyote timer is used which will be discussed later under “Coyote Time Jump”.
3. “Jump()” will check if the “Jump” key was released or cancelled while the player was moving up (y velocity greater than 0). When this happens the y velocity is multiplied by 0.5 to slow the player down. This allows the player to jump at different heights depending on how long they hold the “Jump” key.

5.6.3 Dash

The Author created the “Dash()” function which first checks:

1. If the player can dash with the Boolean “canDash”. If true, the “Coroutine” “Dash()”will start.
2. This “Coroutine” sets the “dashing” animation trigger, makes the player’s gravity scale 0, adds a velocity to the player and starts emitting a trail renderer. It will wait for “dashingTime” seconds.
3. Once done waiting, it will turn off the trail renderer, reset the player’s gravity scale and reset the “dashing” animation trigger. It will then wait for “dashingCooldown” seconds before setting “canDash” back to true.

5.7 Fire() Function

The “Fire()” function is using the new input system in Unity. When the specified input is pressed (J on keyboard, Right Trigger on gamepad) the player will attack. This function needs the following variables:

“attackRange” the circumference of the circle where an enemy can be hit,

“attackPoint” this is a transform, which is attached to the player object and will move with them.

“enemyLayers” this layer mask will be used to check for any enemies that have the “Enemy” layer hit in the attack range.

“attackDamage” is how much damage the player will do to the enemy.

“attacking” is used to check if the player is attacking or not.

The author was planning on having a two-step swing where the first swing is a down-swing and the second is an upswing. If attacking back-to-back the player would do the first attack then the second. But there was not enough time to implement this.

“attackRate” is used for a cooldown on the player attack so the player cannot attack too quickly, and

the “nextAttackTime” variable is used to check if the player can do their next attack.

```
[Header("Attack Details")]
[SerializeField]private float attackRange = .8f;
[SerializeField]private Transform attackPoint;
[SerializeField]private LayerMask enemyLayers;
private float attackDamage = 40f;
private bool attacking;
//private float chainAttackTime = 0.7f;
private float attackRate = 2f;
private float nextAttackTime = 0f;
```

When the fire input action (J key on keyboard, Right trigger on gamepad) is pressed, it checks if the current time is greater than the next attack variable. This means the player cannot attack while the attack cooldown is in progress. “Fire()” then uses the “Attack()” function which plays the attack animation, detects the enemies in the attack range and then damages them.

```
public void Fire(InputAction.CallbackContext context)
{
    if(Time.time >= nextAttackTime)
    {
        Attack();
        nextAttackTime = Time.time + 1f / attackRate;
    }
}
```

Figure 58 Fire() function

5.7.1 Detecting enemies

The “Collider2D[]” array “hitEnemies” stores any enemies that are tagged with the “enemyLayers” layer mask in the “attackRange”.

5.7.2 Damaging enemies

Running a “foreach” loop iterates through each enemy in “hitEnemies” (the array that stored all enemies hit in the attack range) then the function “TakeDamage()” is used to deal the attack damage variable to the enemy.

```
private void Attack()
{
    //play attack anim
    myAnimator.SetTrigger("attack1");


    //detect enemies
    Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(attackPoint.position,
    attackRange, enemyLayers);


    //damage them
    foreach(Collider2D enemy in hitEnemies)
    {
        enemy.GetComponent<EnemyScript>().TakeDamage(attackDamage);

    }

}
```

Figure 59 Attack() function

The “TakeDamage()” function from “Character.cs” takes in a float “attackDamage”. The damage done is taken away from the “currentHealth” variable that stores the current health of the character. This has been updated so that the player and enemies now

share this function with inheritance. Once the damage is done, the animation for “Hurt” will trigger. If the health is less than 0 the character will “Die()”

```
public void TakeDamage(float damage)
{
    currentHealth -= damage;

    //play hurt anim
    myAnimator.SetTrigger("Hurt");

    if(currentHealth <= 0)
    {
        Die();
    }
}
```

Figure 60 TakeDamage() Function

The “Die()” function from “Character.cs” will play the death animation, and start the “Coroutine” “DisableOnDeath()”

```
void Die()
{
    //die animation
    myAnimator.SetBool("IsDead", true);
    //disable enemy
    StartCoroutine(DisableOnDeath());
}
```

Figure 61 Die() Function

The “DisableOnDeath()” Coroutine from “Character.cs” will wait for a duration for the death animation to finish then it will disable both the “Collider2D” and the “SpriteRenderer”. This will ensure the player cannot collide with or see the object.

```
IEnumerator DisableOnDeath()
{
    yield return new WaitForSeconds(.8f);
    GetComponent<Collider2D>().enabled = false;
    GetComponent<SpriteRenderer>().enabled = false;
    this.enabled = false;
    yield return null;
}
```

Figure 62 DisableOnDeath() Function

5.8 Knockback.cs

This script was intended for use in the “Attack()” function, However, due to time the constraints of the project, the creator was only able to implement a knockback when the enemy collides with the player.

This script uses:

2 float variables, and

1 string variable.

```
public class Knockback : MonoBehaviour
{

    [SerializeField] private float knockForce;
    [SerializeField] private float knockbackTime;
    [SerializeField] private string otherTag;
```

The “OnTriggerEnter2D()” function will check:

If the object that collided with it has the tag “otherTag” and if the collider has a trigger.

If the object that collided with this has a rigid body, a force will be applied to it. The “knockForce” variable is used here to calculate how far the object will be knocked.

If it has a tag of “Enemy” they will be knocked back, or

If it has a tag of “Player” they will be knocked back.

```
private void OnTriggerEnter2D(Collider2D other)
{
    //checking for the specified tag entering the trigger collider.
    if(other.gameObject.CompareTag(otherTag) && other.isTrigger)
    {
        Rigidbody2D hit = other.GetComponent<Rigidbody2D>();
        //if the rigid body hit has a value.
        if(hit != null)
        {
            //Getting a value for how far the hit rigid body will be knocked.
            Vector2 difference = hit.transform.position - transform.position;
            difference = difference.normalized * knockForce;
            //adding a force to the hit object
            hit.AddForce(difference, ForceMode2D.Impulse);

            //if the thing entering the trigger collider has the enemy tag.
            if(other.gameObject.CompareTag("Enemy") )
            {
                //using the knockback function.
                other.GetComponent<Enemy>().Knockback(hit, knockbackTime);
            }
            //if the thing entering the trigger collider has the player tag.
            if(other.gameObject.CompareTag("Player"))
            {
                //using the knockback function.
                other.GetComponent<Player>().Knockback(hit, knockbackTime);
            }
        }
    }
}
```

Figure 63 Knockback Funciton

5.9 Move() Function

In the Update function the rigid body (the player) is given a velocity which is a Vector 2. This value is the direction multiplied by the “speed” variable which is 8 by default. Since this is only changing which direction the player moves, the y value is not changed.

```
public void Move(InputAction.CallbackContext context)
{
    direction = context.ReadValue<Vector2>().x;
    myAnimator.SetFloat("speed", Mathf.Abs(direction));

}
```

```
private void Update()
{
    //if the player is on the ground the falling anim will be false.
    if (IsGrounded())
    {
        myAnimator.SetBool("falling", false);

        jumpsLeft = maxJumps;

    }
    //making sure the player is facing the correct direction.
    if (!facingRight && direction > 0f)
    {
        Flip();
    }
    //making sure the player is facing the correct direction.
    else if(facingRight && direction < 0f)
    {
        Flip();
    }
    //controlling the movement of the player.
    rb.velocity = new Vector2(direction * speed, rb.velocity.y);
}
```

Figure 64 Move() Function Version 1

5.9.1 Updated Move() Function

This was moved to the “FixedUpdate()” because the player was sometimes glitching through walls and objects. It is better to deal with physics in the “FixedUpdate()”.

```
private void FixedUpdate()
{
    if(PauseScript.isPaused) return;
    if(isDashing || isWallJumping) return;
    //controlling the movement of the player, changing the x velocity.
    ChangeDirection();
    rb.velocity = new Vector2(direction * speed, rb.velocity.y);

}
```

Figure 65 Move() Function Updated

The “Move()” function was updated because the player should not be able to move while wall jumping until the jump is finished. This code ensures that the player cannot move while wall jumping.

The author also changed the speed to not be an absolute value. While the absolute value made the game unique with slight physics speed-up/slow-down movement, the author preferred having snappy movement over this.

```
//finding the direction the player is trying to move
public void Move(InputAction.CallbackContext context)
{

    if(IsWalled() && !isWallJumping)
    {
        //when the player moves.
        if(context.performed)
        {
            direction = context.ReadValue<Vector2>().x;
        }
        else if(context.canceled)
        {
            direction = context.ReadValue<Vector2>().x;
        }

    }
    else if (context.performed)
    {
        direction = context.ReadValue<Vector2>().x;
    }
    else if (context.canceled)
    {
        direction = context.ReadValue<Vector2>().x;
    }

}
```

Figure 66 Move() Function Version 2

5.10 Player Jump

As stated in the player movement section, “Jump()” is using Unity’s new Input System. The “Jump()” function takes in an input from the controller or keyboard. When the jump button is pressed, and the player has more than 0 jumps left (a variable stores how many jumps the player has left) a vertical velocity (“jumpForce”) is applied and the player moves up. The “jump” animation is played here. Once the player has jumped the “jumpsLeft” variable is now -1.

If the player cancels the jump (by letting go of the “jump” button) but their vertical velocity is greater than 0 (i.e. they are still moving up), the velocity will be multiplied by .5 which lets the player choose how high they can jump. The player can jump the highest by holding the “jump” button or they can choose to jump lower by letting go early.

```
//using the new input system to control the player jump.
public void Jump(InputAction.CallbackContext context)
{
    //when jump is pressed and jumps left is more than 0.
    if (context.performed && jumpsLeft > 0)
    {
        //adds a veloctiy (jump force) to the y value of the rigid body.
        rb.velocity = new Vector2(rb.velocity.x, jumpForce);
        myAnimator.SetTrigger("jump");
        myAnimator.SetBool("falling", false);

        //the coyote timer makes it so there is some leaniency with jumping if
        //taking away 1 jump from jumps left.
        if(coyoteTimeCounter <= 0f)
        {
            jumpsLeft -= 1;
        }

    }
    //when jump is canceled.
    else if (context.canceled && rb.velocity.y > 0f)
    {
        //allowing the player to jump higher by pressing jump for longer and l
        rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y * 0.5f);
        myAnimator.SetBool("falling", true);
        myAnimator.ResetTrigger("jump");

        coyoteTimeCounter = 0f;
    }


}
```

Figure 67 Jump() Function Version 1

510.1 Jump() function Updated

The jump function is almost the same but with added functionality. It will now not let the player jump while the game is paused. It will use the “Coroutine” “WallBounce()” if the jump button is pressed while “canWallJump” is true.

```
public void Jump(InputAction.CallbackContext context)
{
    if(PauseScript.isPaused) return;
    //Jump from the wall
    if (context.performed && canWallJump)
    {

        StartCoroutine(WallBounce());
    }
```

Figure 68 Jump Function Version 2

“WallBounce()” is explained in detail in the “WallBounce()” function later in this chapter.

The “jumpsLeft” variable was changed to the more fitting name “jumps”.

When the “jump” button is pressed, “maxJumps” is greater than “jumps” and the player isn’t wall sliding, the jump force is applied to the player’s rigidbody. The “jumps” variable then increases by 1 until the player touches the ground.

```
//when jump is pressed and jumps left is less than max jumps.
//Normal jump without wall
//using a jump after the player leaves the wall
else if (context.performed && maxJumps > jumps && !isWallSliding)
{

    isJumping = true;
    //adds a velocity (jump force) to the y value of the rigid body.
    rb.velocity = new Vector2(rb.velocity.x, jumpForce);

    //the coyote timer makes it so there is some leaniency with jumping if you
    have just left the ground and the jump button is pressed the player will still
    jump.
    if(coyoteTimeCounter <= 0f)
    {
    //addingg a jump to the jumps used.
        jumps += 1f;
    }
```

Figure 69 Jump Function Version 2

When the jump is cancelled while the player is moving up (while the jump is not yet complete) it will make the player slow down its upward velocity and start to fall.

```
//when jump is canceled.
else if ((context.canceled && rb.velocity.y > 0f) && (!canWallJump))
{
    isJumping = false;
    isWallJumping = false;
    wallJumpingDirection = 0f;
    //allowing the player to jump higher by pressing jump for longer and lower by
    pressing it for a short time.
    rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y * 0.5f);
    coyoteTimeCounter = 0f;

}
```

Figure 70 Jump Function Version 2

```
else if (context.canceled)
{
    wallJumpingDirection = 0f;
    isJumping = false;
    isWallJumping = false;
    coyoteTimeCounter = 0f;

}
```

Figure 71 Jump Function Version 2

5.10.2 Coyote Time Jump

Coyote time is used to add a short window after the player leaves the ground to still be able to jump. This takes account for human error giving the player a .2 second window to jump after they leave the ground or a platform.

Two variables are needed. One to store how long the error window will be and one that can be changed.

```
private float coyoteTime = 0.2f;
private float coyoteTimeCounter;
```

In the “Update()” function when the player is on the ground the coyoteTime is assigned to the variable “coyoteTimeCounter”. Once the player leaves the ground the “coyoteTimeCounter” will count down, when the timer gets to 0 or less the player will no longer be able to use their first jump.

```
private void Update()
{
    //if the player is on the ground the falling anim will be false.
    if (IsGrounded())
    {
        myAnimator.SetBool("falling", false);

        jumpsLeft = maxJumps;

        coyoteTimeCounter = coyoteTime;

    }
    else
    {
        coyoteTimeCounter -= Time.deltaTime;
    }
```

If the coyote timer is less than or equal to 0 then it will take away 1 jump from the “jumpsLeft” variable. This makes it so the player cant get extra jumps from the coyote timer.

```
//when jump is pressed and jumps left is more than 0.
if (context.performed && jumpsLeft > 0)
{
    //adds a velocitiy (jump force) to the y value of the rigid body.
    rb.velocity = new Vector2(rb.velocity.x, jumpForce);
    myAnimator.SetTrigger("jump");
    myAnimator.SetBool("falling", false);

    //the coyote timer makes it so there is some leaniency with jump
    //taking away 1 jump from jumps left.
    if(coyoteTimeCounter <= 0f)
    {
        jumpsLeft -= 1;
    }
```

To make sure the player can't jump indefinitely by pressing the jump button fast, the timer has to be set to 0 once the jump button is released.

```
else if (context.canceled && rb.velocity.y > 0f)
{
    //allowing the player to jump higher by pressing jump for longer and lower by pressing it for a short ti
    rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y * 0.5f);
    myAnimator.SetBool("falling", true);
    myAnimator.ResetTrigger("jump");

    coyoteTimeCounter = 0f;
}
```

The "Jump()" Function was Updated but the Coyote Timer is the same.

```
//using a jump after the player leaves the wall
else if (context.performed && maxJumps > jumps && !isWallSliding)
{

    isJumping = true;
    //adds a velocity (jump force) to the y value of the rigid body.
    rb.velocity = new Vector2(rb.velocity.x, jumpForce);

    //the coyote timer makes it so there is some leaniency with jumping if you
    have just left the ground and the jump button is pressed the player will still
    jump.
    if(coyoteTimeCounter <= 0f)
    {
    //addingg a jump to the jumps used.
        jumps += 1f;
    }

}

//when jump is canceled.
else if ((context.canceled && rb.velocity.y > 0f) && (!canWallJump))
{
    isJumping = false;
    isWallJumping = false;
    wallJumpingDirection = 0f;
    //allowing the player to jump higher by pressing jump for longer and lower by
    pressing it for a short time.
    rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y * 0.5f);
    coyoteTimeCounter = 0f;

}
else if (context.canceled)
{
    wallJumpingDirection = 0f;
    isJumping = false;
    isWallJumping = false;
    coyoteTimeCounter = 0f;

}
```

Figure 72 Updated Jump() with Coyote timer shown

5.10.3 WallBounce() Coroutine function or Wall Jump

This Coroutine is performed in the “Jump()” function when the “canWallJump” variable is true. “WallBounce()” will make “canWallJump” false, “isWallJumping”, true and trigger the “jump” animation. It will set the player’s gravity scale to 0 and apply the velocity “wallJumpingPower” to the player. The direction the player is launched is opposite to the direction the player is facing. Then it will wait for “wallJumpingTime”

seconds before resetting the player's gravity scale, changing "isWallJumping" to false and resetting the "jump" animation trigger.

```
[Header("Wall Jump Details")]
[SerializeField]private float wallJumpingDirection;
[SerializeField]private Vector2 wallJumpingPower = new Vector2(9f,9f);
[SerializeField]private float wallJumpingTime;
private float maxWallJumpingTime = .1f;
[SerializeField]private bool canWallJump = true;
[SerializeField]private bool isWallJumping;
[SerializeField]private float wallJumpingCooldown = .1f;
```

```
//using the new input system to control the player jump.
public void Jump(InputAction.CallbackContext context)
{
    if(PauseScript.isPaused) return;
    //Jump from the wall
    if (context.performed && canWallJump)
    {

        StartCoroutine(WallBounce());
    }
```

```
IEnumerator WallBounce()
{

    canWallJump = false;
    isWallJumping = true;
    myAnimator.SetTrigger("jump");
    myAnimator.SetBool("walled", false);
    newGravity = rb.gravityScale;
    rb.gravityScale = 0f;
    rb.velocity = new Vector2(wallJumpingDirection * wallJumpingPower.x,
    wallJumpingPower.y);
    //tr.emitting = true;

    yield return new WaitForSeconds(wallJumpingTime);

    //tr.emitting = false;
    rb.gravityScale = originalGravity;
    isWallJumping = false;
    myAnimator.ResetTrigger("jump");


}
```

Figure 73 WallBounce() Function

In the “Update()” function the “wallJumpingDirection” is set to the opposite of the current direction, and “wallJumpingTime” is assigned while the player is touching the wall.

This has been updated to the “Player.cs” script “Update()” function.

```
if (IsWalled())
{


    wallJumpingDirection = -direction;
    wallJumpingTime = maxWallJumpingTime;

}
else
{
    wallJumpingTime -= Time.deltaTime;
}
```

Figure 74 if IsWalled() in the Update() function

5.10.4 WallJumpCooldown() Coroutine function

This function is used to make sure the first time the player jumps onto the wall and every time after, there will be a cooldown between jumps. In the “FixedUpdate()” it checks if the player “IsWalled()”. If they are, it will start the “WallJumpCooldown()” Coroutine.

```
private void FixedUpdate()
{
    if(PauseScript.isPaused) return;
    if(isDashing || isWallJumping) return;
    //controlling the movement of the player, changing the x velocity.
    if(IsWalled()) StartCoroutine(WallJumpCooldown());
    ChangeDirection();
    rb.velocity = new Vector2(direction * speed, rb.velocity.y);

}
```

This Coroutine will wait for “wallJumpingCooldown” seconds before making “canWallJump” true again, which allows the player to jump.

```
IEnumerator WallJumpCooldown()
{
    if (!canWallJump)
    {
        yield return new WaitForSeconds(wallJumpingCooldown);
        canWallJump = true;
    }

}
```

Figure 75 WallJumpCooldown() Function

5.10.5 Double Jump

In “PlayerScript.cs” the “Start()” function sets the current amount of times the player has jumped to the “jumps” variable.

```
//getting the rigid body and animator components.
private void Start()
{
    rb = GetComponent<Rigidbody2D>();
    myAnimator = GetComponent<Animator>();

    jumpsLeft = maxJumps;

}
```

In the “Update()” function every time the player is on the ground the “jumpsLeft” variable will reset to the max number of jumps.

```
private void Update()
{
    //if the player is on the ground the falling anim will be false.
    if (IsGrounded())
    {
        myAnimator.SetBool("falling", false);
        jumpsLeft = maxJumps;

        coyoteTimeCounter = coyoteTime;

    }
```

In the “Jump()” function as long as there is more than 0 “jumpsLeft” the player will be able to jump. So, if “maxJumps” is at 2 the player will be able to double-jump.

```
public void Jump(InputAction.CallbackContext context)
{
    //when jump is pressed and jumps left is more than 0.
    if (context.performed && jumpsLeft > 0f)
    {
        //adds a velocitiy (jump force) to the y value of the rigid body.
        rb.velocity = new Vector2(rb.velocity.x, jumpForce);
```

Figure 76 Jump() Function Double Jump

Once the jump is pressed the “jumpsLeft” will be updated with the new number of jumps left which is 1 less.

```
if (context.performed && jumpsLeft > 0f)
{
    //adds a velocity (jump force) to the y value of the ri
    rb.velocity = new Vector2(rb.velocity.x, jumpForce);
    myAnimator.SetTrigger("jump");
    myAnimator.SetBool("falling", false);

    //the coyote timer makes it so there is some leaniency
    ground and the jump button is pressed the player will s
    //taking away 1 jump from jumps left.
    if(coyoteTimeCounter <= 0f)
    {
        jumpsLeft -= 1;
    }
```

Figure 77 Jump() Function Double Jump

In the “Update()” function “maxJumps” is set to “jumpsLeft” when the player is on the ground. This means the player’s “jumpsLeft” will reset to whatever value of “maxJumps” when they are on the ground.

```
private void Update()
{
    //if the player is on the ground the falling anim will be false.
    if (IsGrounded())
    {
        myAnimator.SetBool("falling", false);

        jumpsLeft = maxJumps;
```

Figure 78 Jump() Function Double Jump

5.11 Dash() Function

When the dash button is pressed (K on keyboard / button east on gamepad) and the player is able to dash, the “Dash()” coroutine will run. This function uses a trail renderer, a float for the power of the dash, a float for the duration of the dash, a Boolean to check if the player can dash, a Boolean to check if the player is dashing (to inhibit movement while dashing) and a float for the dash cool down.

```
[Header("Dash Details")]
[SerializeField]private TrailRenderer tr;
[SerializeField]private float dashingPower = 40f;
[SerializeField]private float dashingTime = .1f;
private bool canDash = true;

private bool isDashing;
private float dashingCooldown = 1f;
```

5.11.1 Dash() Coroutine Function

This function sets the “canDash” variable to false to stop multiple dashes while the current dash is activated. “IsDashing” is set to true to stop the player’s other movement abilities.

```
private void Update()
{
    if(PauseScript.isPaused) return;
    if(isDashing || isWallJumping)
        return;
```

The “dashing” animation is played, the gravity scale is set to 0, the dashing power is given to the player’s “rigidbody”, and the trail renderer starts emitting. The coroutine waits for “dashingTime” seconds and turns off the trail renderer, resets the player’s gravity scale, animation and “isDashing”. It waits for the cool down to finish before setting “canDash” back to true.

```
//when dash() is run the original gravity of the player is stored, the players
gravity is 0 so they are not affected by gravity while they dash
//the players velocity is changed (dashing power), and the trail starts
emitting
canDash = false;
isDashing = true;
myAnimator.SetTrigger("dashing");
originalGravity = rb.gravityScale;
rb.gravityScale = 0f;
rb.velocity = new Vector2(dashingPower, 0f);
tr.emitting = true;

//waits until the end of the dash.
yield return new WaitForSeconds(dashingTime);

//turns off the trail, resets the gravity scale.
tr.emitting = false;
rb.gravityScale = originalGravity;
isDashing = false;
myAnimator.ResetTrigger("dashing");

//wait for the cool down before being able to dash again.
yield return new WaitForSeconds(dashingCooldown);

canDash = true;
```

Figure 79 Dash() Function

5.12 Interact() Function

The “Interact()” function can be improved still but for now it does its job. The author will explain how they will improve it at the end of this section. The button to Interact on Keyboard is “E” and on Gamepad is the north button on the right side. When performed the Function “CheckInteractions()” is used. In the “Player.cs” script, the variables that will be used are:

“interactRange” (the radius of the circle) which surrounds the NPC or interactable object, and

“NPCLayers” a layer mask used to check if the player is overlapping the “interactRange” of any NPC or interactable object.

```
[Header("Interactables")]
[SerializeField]float interactRange = 3f;
[SerializeField] private LayerMask NPCLayers;
```

```
public void Interact(InputAction.CallbackContext context)
{

    if(context.performed)
    {
        CheckInteractions();
    }


}
```

Figure 80 Interact() Function

5.12.1 CheckInteractions() Function

This function finds an array of colliders adjacent to the player (Collider2D) and stores them in a variable. It then uses a “foreach” loop to iterate through each “Collider2D” in the array and uses the Unity Function “TryGetComponent()”

The author has Created a C# script “NPCInteractable.cs” which is attached to NPCs. This script will be explained after this section.

If it finds the component, it will use the “Interact()” function from that script.

```
private void CheckInteractions()
{

    Collider2D[] colliderArray = Physics2D.OverlapCircleAll(transform.position,
    interactRange, NPCLayers);
    foreach (Collider2D collider in colliderArray)
    {
        if(collider.TryGetComponent(out NPCInteractable npcInteractable))
        {
            npcInteractable.Interact();
        }

    }
}
```

Figure 81 CheckInteractions() function

5.12.2 NPCInteractable.cs

This script was created to allow the player to interact with characters, through text, but in the future could be changed to also include object interactions such as opening door or using a lift. Unfortunately, project time constraints do not allow the author to explore this further.

The “Interact()” function in the “NPCInteractable.cs” script checks if:

The dialogue panel is visible or not, if its not visible it will make it visible and start the “Typing()” Coroutine.

The dialogue text is the same as the dialogue index, the “NextLine()” function is used to move the text to the next line, and

if neither of these are used the “ResetText()” is used which removes the text and dialogue panel.

The continue button is usable once the text is finished typing or when the player presses the interact button again.

```
0 references
public void Interact()
{
    if (!dialoguePanel.activeInHierarchy)
    {
        dialoguePanel.SetActive(true);
        StartCoroutine(Typing());
    }
    else if (dialogueText.text == dialogue[index])
    {
        NextLine();
    }
    else
    {
        resetText();
    }

    contButton.SetActive(true);
}
```

Figure 82 Interact() function in the NPCInteractable script

5.12.3 ResetText() Function

The “ResetText()” function is used to remove the current text and make the dialogue panel invisible.

```
4 references
public void resetText()
{
    dialogueText.text = "";
    index = 0;
    dialoguePanel.SetActive(false);
}
```

Figure 83 resetText() Function

5.12.4 Typing() Function

The “Typing()” Coroutine function is used to display a typing effect, it uses a foreach loop to go through each letter in the dialogue array to show each letter as if its being typed. The “wordSpeed” variable will dictate how fast this is.

```
2 references
public IEnumerator Typing()
{
    foreach(char letter in dialogue[index].ToCharArray())
    {
        dialogueText.text += letter;
        yield return new WaitForSeconds(wordSpeed);
    }
}
```

Figure 84 Typing() Function

5.12.5 ResetText() Function

The “NextLine()” function will go to the next line of the dialogue. It will remove the current dialogue text and start the “Typing()” function for the next line of text. This has a bug, if the player uses the interact button repeatedly this script will not stop the typing coroutine each time it is interacted with, thus making a jumble of words that are being typed from different parts of the script at the same time. This can be fixed by stopping the “Typing()” coroutine once the “NextLine()” function is used. However, due to the time constraints of this project this will not be fixed by the due date.

```
public void NextLine()
{
    contButton.SetActive(false);

    if(index < dialogue.Length -1)
    {
        index++;
        dialogueText.text = "";
        StartCoroutine(Typing());
    }
    else
    {
        resetText();
    }
}
```

Figure 85 NextLine() Function

5.12.6 OnTriggerEnter2D() & OnTriggerExit2D() Function

The “OnTriggerEnter2D()” and “OnTriggerExit2D()” functions will check if the player is in range to interact with the NPC, and will reset the text once they leave the radius of the trigger. This prevents the player from having the dialogue box open while not being near the NPC.

```
private void OnTriggerExit2D(Collider2D other)
{
    if(other.CompareTag("Player"))
    {
        playerIsClose = false;
        resetText();
    }
}
```

```
private void OnTriggerEnter2D(Collider2D other)
{
    if(other.CompareTag("Player"))
    {
        playerIsClose = true;
        resetText();
    }
}
```

5.12.7 Updated NPCInteractable.cs

The below image is the updated “Interact()” function. It now will go to the next dialogue index when the player interacts again.

```
public void Interact()
{
    if (!dialoguePanel.activeInHierarchy)
    {
        resetText();
        dialoguePanel.SetActive(true);
        StartCoroutine(Typing());
    }
    else if (dialoguePanel.activeInHierarchy)
    {
        NextLine();
    }
    else if (dialogueText.text == dialogue[index])
    {
        NextLine();
    }
    else
    {
        resetText();
    }
```

Figure 86 Version 2 Interact() Function in NPCInteractable

The “resetText()” function has been updated to stop the “Typing()” Coroutine.

```
5 references
public void resetText()
{
    StopCoroutine(Typing());
    dialogueText.text = "";
    index = 0;
    dialoguePanel.SetActive(false);
}
```

Figure 87 Version 2 resetText() Function in NPCInteractable

The “NextLine()” function has also been updated to stop the “Typing()” Coroutine.

```
2 references
public void NextLine()
{
    if(index < dialogue.Length -1)
    {
        StopCoroutine(Typing());
        index++;
        dialogueText.text = "";
        StartCoroutine(Typing());
    }
    else
    {
        resetText();
    }
}
```

Figure 88 Version 2 NextLine() Function in NPCInteractable

The author attempted to make the icon above all interactable objects appear while in range and disappear while out of range.

```
// void showInteractIcon()
// {
//     if(playerIsClose)
//     {
//         interact.SetActive(true);
//     }
//     else
//     {
//         interact.SetActive(false);
//     }
// }
```

5.13 One Way platforms

The one-way platforms use the variable “fallThrough”. This will be a toggleable Boolean for whether the player can fall through the object or not.

```
[Header("One Way Platform")]
public bool fallThrough;
```

This is currently only working for keyboard. When the player presses the down arrow or “S” key the player will fall through the object.

```
//dealing with platforms that you can stand on but also choose to jump down from.
//when the down arrow key is pressed the bool fall through is set to true and lets
the player fall.
if (Keyboard.current.downArrowKey.isPressed || Keyboard.current.sKey.isPressed)
{

    fallThrough = true;
}
else
{
    fallThrough = false;
}
```

Figure 89 From the Player Update() function

5.13.1 OneWayPlatform.cs script

This script will first get the “BoxCollider2D” from the object it is placed on.

```
private void Awake()
{

    box = GetComponent<BoxCollider2D>();
}
```

5.13.2 OnCollisionEnter2D() Function

When another object first collides with this object it checks for the Tag “Player “.

```
//when an object collides with the platform check if it is the player using the compar
tag function.
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.collider.CompareTag("Player"))
        playerControls = collision.gameObject.GetComponent<PlayerScript>();
}
```

Figure 90 From OneWayPlatform.cs

Updated the GetComponent to <Player>() from <PlayerScript>() This was the reason the “One Way Platforms” didn’t work when inheritance was being added. The creator did not update this until the end of the project.

```
0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.collider.CompareTag("Player"))
        playerControls = collision.gameObject.GetComponent<Player>();
}
```

Figure 91 From OneWayPlatforms.cs

5.13.3 OnCollisionStay2D() Function

When the player is still in contact with this object and the variable “fallThrough” from “Player.cs” is true, the “Coroutine” “DisableCollision()” will start.

```
private void OnCollisionStay2D(Collision2D collision)
{
    if(playerControls == null)
        return;
    if(playerControls.fallThrough)
    {
        StartCoroutine(DisableCollision());
    }
}
```

Figure 92 From OneWayPlatforms.cs

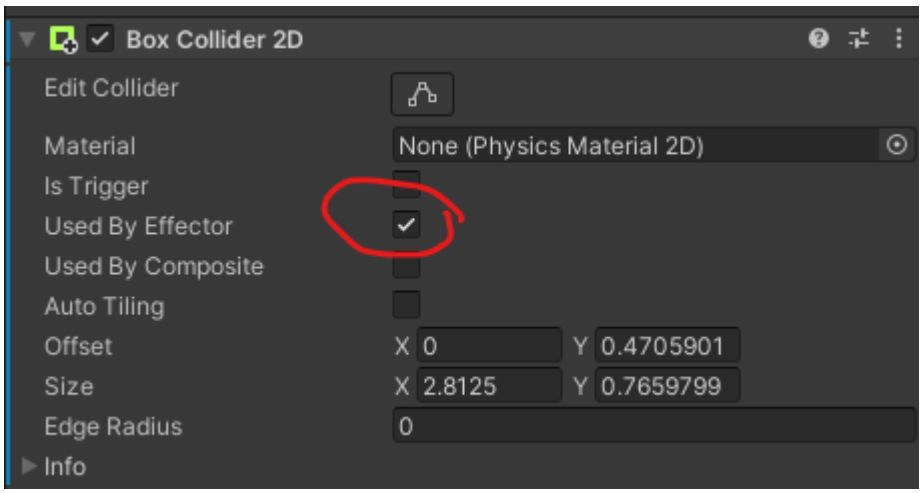
5.13.4 DisableCollision() Function

The “DisableCollision()” function disables the “box” which will turn off the box collider of the object. After half a second the object’s “box” will be enabled again, and the “playerControls” are set back to null to ensure the player does not fall through the object again.

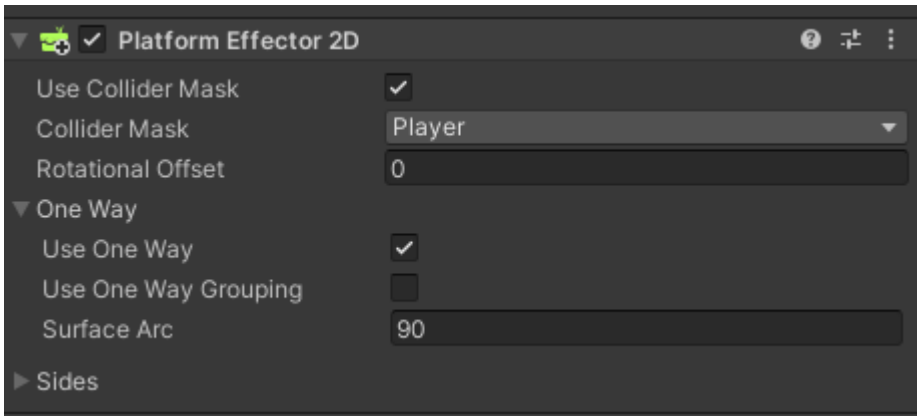
```
private IEnumerator DisableCollision()
{

    box.enabled = false;
    yield return new WaitForSeconds(0.5f);
    box.enabled = true;
    playerControls = null;

}
```

Figure 93 From OneWayPlatforms.cs

The object’s box collider must have “Used By Effector” ticked.



The Platform Effector 2D lets the player jump onto the object but only if the player is landing on the object from a specified angle e.g., below image is 90 Degrees. This allows the player the to jump up through the object but not down unless the down button is pressed while in contact with it.


Platform Effector 2D
Use Collider Mask
Collider Mask
Player
Rotational Offset
0
One Way
Use One Way
Use One Way Grouping
Surface Arc
90
Sides

5.14 RespawnScript.cs

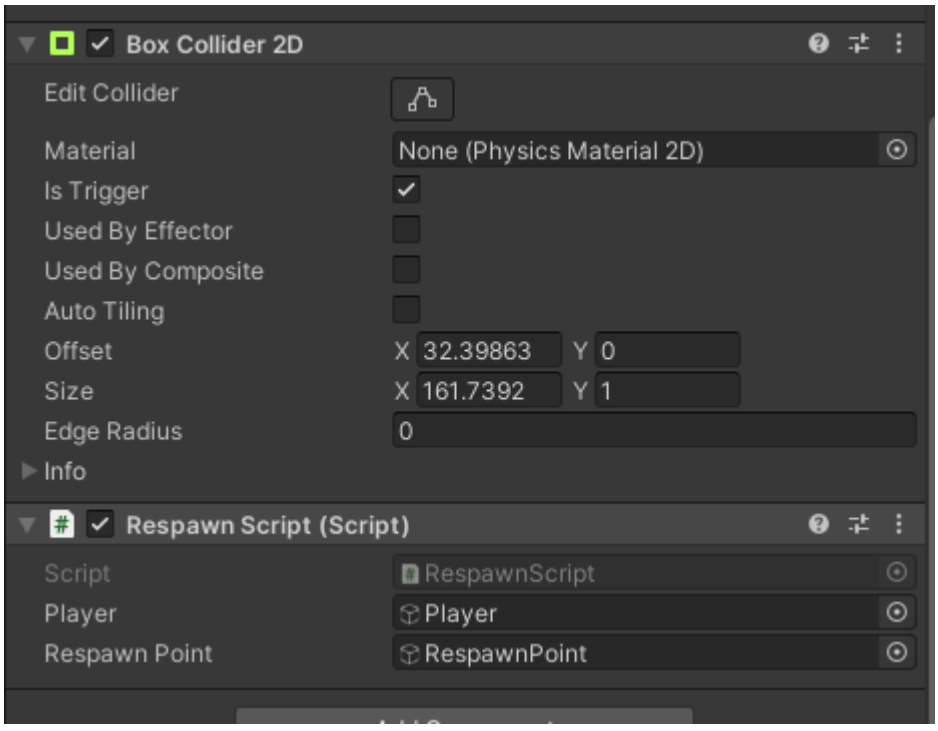
This script uses the player position and the respawn point position.

This script is attached to the “RespawnCollider” object whose box collider’s “Is Trigger” is checked.

When the player enters the box collider of “RespawnCollider” the “OnTriggerEnter2D()” function moves the player’s current position to the respawn point’s position.

```
public GameObject player;
public GameObject respawnPoint;

private void OnTriggerEnter2D(Collider2D other)
{
    player.transform.position = respawnPoint.transform.position;
}
```

Figure 94 From RespawnScript.cs



5.14.1 CheckpointScript.cs

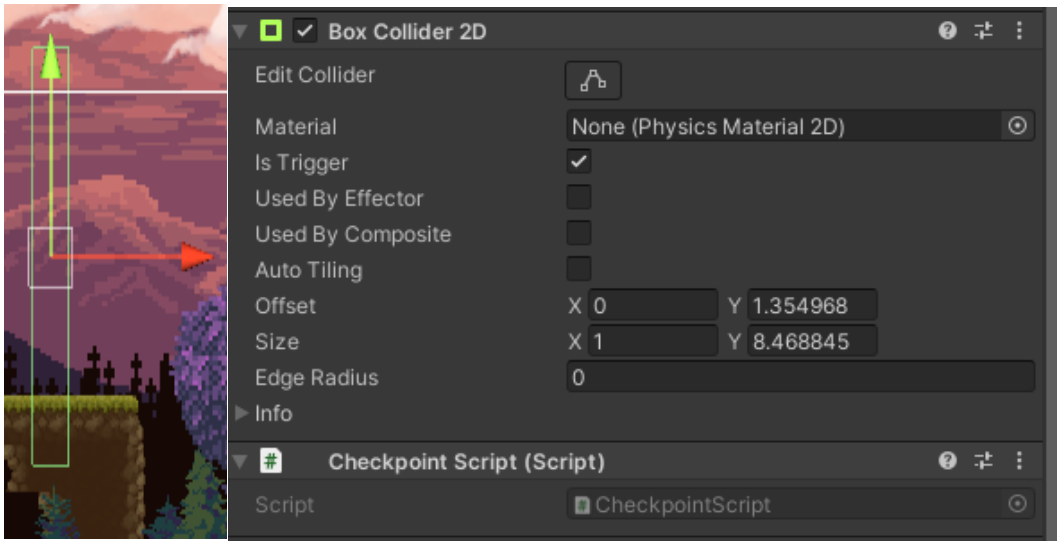
This script uses a reference to the “RespawnScript.cs”.

The “Awake()” function runs as soon as the script is started, it gets the component of the “RespawnCollider” and the respawn point can now be updated to a new point.

```
private RespawnScript respawn;

void Awake()
{
    respawn = GameObject.FindGameObjectWithTag("Respawn").
    GetComponent<RespawnScript>();
}


private void OnTriggerEnter2D(Collider2D other)
{
    if(other.gameObject.CompareTag("Player"))
    {
        respawn.respawnPoint = this.gameObject;
    }
}
```

Figure 95 From CheckpointScript.cs

These checkpoint objects are placed throughout the level and use a Box Collider 2D with the “Is Trigger” checked.



5.15 PauseScript.cs

This pause script is put on the empty menu controller object.

A reference to “playerControls” is used.

The variables used in this script are:

The GameObject “PauseMenu”, and

“isPaused” which will enable / disable actions depending on its state.

The “isPaused” variable is a “public static bool”. This is to make sure it is accessible to other classes.

```
{
    [SerializeField] private GameObject pauseUI;
    [SerializeField] public static bool isPaused = false;
    private PlayerControls playerControls;
```

```
void Awake()
{
    playerControls = new PlayerControls();
}
```

5.15.1 PauseCheck() Function

When the player presses the escape key, the “PauseCheck()” function will run.

```
private void Start()
{
    playerControls.Menu.Escape.performed += _ => PauseCheck();
}
```

“PauseCheck()” will check if the “isPaused” variable is true or false. If it is true it will run the “ResumeGame()” function, otherwise it will run the “PauseGame()” function.

```
private void PauseCheck()
{
    if(isPaused)
    {
        ResumeGame();
    }
    else
    {
        PauseGame();
    }
}
```

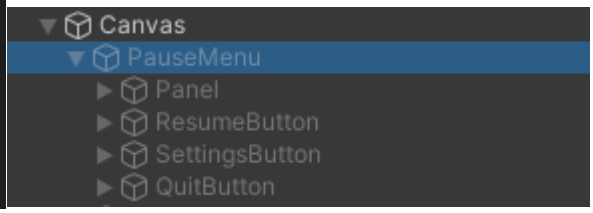
```
private void OnEnable()
{
    playerControls.Enable();
}
private void OnDisable()
{
    playerControls.Disable();
}
```

Figure 96 PauseCheck() Function

5.15.2 PauseGame() Function

"PauseGame()" sets the timescale to 0, stops the game sounds, sets "isPaused" to true, and enables the Pause Menu UI.

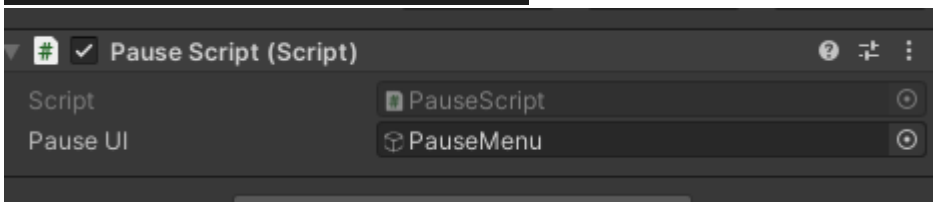
```
void PauseGame()
{
    Time.timeScale = 0;
    AudioListener.pause = true;
    isPaused = true;
    pauseUI.SetActive(true);
}
```



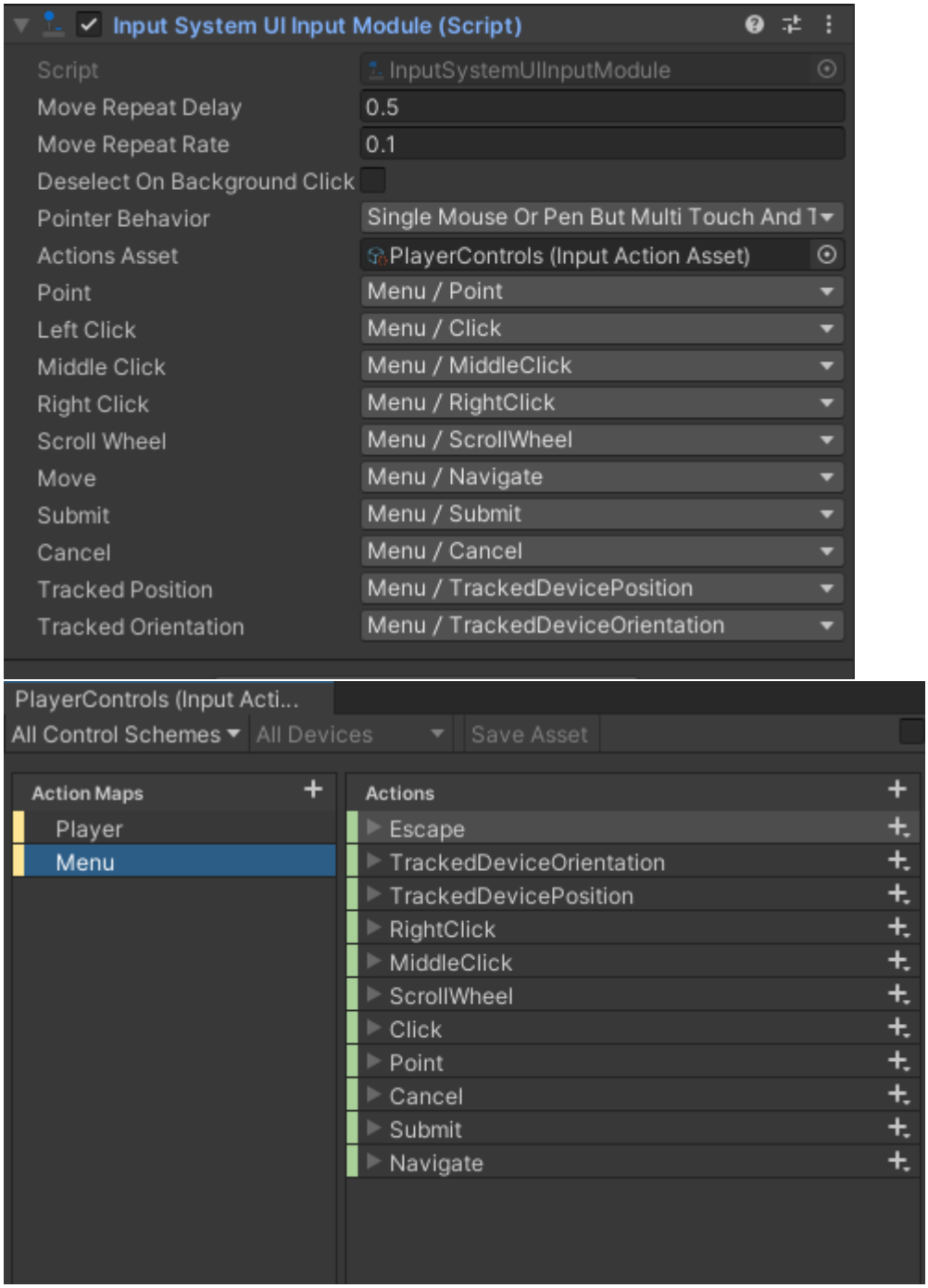
5.15.3 ResumeGame() Function

"ResumeGame()" sets the timescale back to 1, enables the game sounds, sets "isPaused" to false and disables the Pause Menu UI.

```
public void ResumeGame()
{
    Time.timeScale = 1;
    AudioListener.pause = false;
    isPaused = false;
    pauseUI.SetActive(false);
}
```



The Menu Input system using Unity's new input system.



5.15.4 Updated “PauseScript Update()” function

This now show the death UI when the player dies.

```
0 references
void Update()
{

    if(isDead)
    {
        StartCoroutine(PauseGameDeath());
    }
    else StopCoroutine(PauseGameDeath());

}
```

Figure 97 From the PauseScript.cs Update() Function

5.15.5 PauseGameDeath() Function

Once the players death animation finishes it will set isDead to true, pauses the game and shows the “Death UI Menu” menu.

```
2 references
public IEnumerator PauseGameDeath()
{
    yield return new WaitForSeconds(.8f);
    Time.timeScale = 0;
    isDead = true;
    isPaused = true;
    deathUI.SetActive(true);
    yield return null;
}
```

Figure 98 From PauseScript.cs

5.15.6 RestartGame() Function

This restarts the current scene.

```
0 references
public void RestartGame()
{

    isPaused = false;
    isDead = false;
    deathUI.SetActive(false);
    Time.timeScale = 1;
    SceneController.instance.RestartScene();

}
```

Figure 99 From PauseScript.cs

5.15.7 Updated PauseScript.cs Start() Function

Changed “Start()” to make sure the health bar isn’t showing on the Main Menu scene.

```
0 references
private void Start()
{
    if(SceneManager.GetActiveScene().buildIndex == 0)
    {
        healthBarUI.SetActive(false);
        mainMenuUI.SetActive(true);
    } else
    {

        mainMenuUI.SetActive(false);
        healthBarUI.SetActive(true);

    }


    playerControls.Menu.Escape.performed += _ => PauseCheck();
}
```

5.16 EnemyScript.cs

All of the functions in this script have been explained previously in the “Fire()”. This script is placed on enemies. It gets their rigid body and animator components, and their “maxHealth” can be changed in the inspector. This script will be combined with the “PlayerScript.cs” once Inheritance is implemented as they both should use all these functions and variables.

```
private Rigidbody2D rb;
private Animator myAnimator;

public float maxHealth = 100;
private float currentHealth;

// Start is called before the first frame update
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    myAnimator = GetComponent<Animator>();

    currentHealth = maxHealth;
}
```

5.17 MainMenu.cs

This Script has two functions: “PlayGame()”and “QuitGame()”. This script is used in the starting menu. The “PlayGame()” function uses the scene manager to load the first scene and “QuitGame()” closes the application.

```
public class MainMenu : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }

    public void QuitGame()
    {
        Debug.Log("quit");
        Application.Quit();
    }

}
```

Figure 100 MainMenu.cs

5.18 SettingsMenu.cs

In the settings menu the player can change the audio or video. They can change resolutions and graphics, or SFX volume, Background music, and Master volume.

This script uses an array of resolutions, the “Text Mesh Pro” dropdown button, and an audio mixer.

```
Resolution[] resolutions;
public TMP_Dropdown resolutionDropdown;
public AudioMixer audioMixer;
```

5.18.1 SetResolution()

This function uses an array of resolutions, and a reference to the dropdown box. It will get the resolutions that the screen can use and display them in the dropdown box.

```
//Array of resoulutions.
Resolution[] resolutions;
//Dropdown reference
public TMP_Dropdown resolutionDropdown;
```

In the “Start()” function the resolutions array variable stores the screen’s possible resolutions. The dropdown box is then cleared of the current options.

```
0 references
void Start()
{
    masterSlider.value = PlayerPrefs.GetFl
    musicSlider.value = PlayerPrefs.GetFlo
    sfxSlider.value = PlayerPrefs.GetFloat

    resolutions = Screen.resolutions;

    resolutionDropdown.ClearOptions();
    int currentResolutionIndex = 0;
```

Figure 101 From SettingsMenu.cs Start() Function

The “AddOptions()” Unity function takes in a list of strings, not an array of resolutions. The array of resolutions needs to be formatted into a list of strings.

A for loop goes through the elements in the resolutions array. The option string gets the width and height of the [i] element in the resolutions array and then it is added to the options list.

```
int currentResolutionIndex = 0;

List<string> options = new List<string>();
for (int i = 0; i < resolutions.Length; i++)
{
    string option = resolutions[i].width + " x " + resolutions[i].height;
    options.Add(option);

    if(resolutions[i].width == Screen.currentResolution.width && resolutions
    [i].height == Screen.currentResolution.height)
    {
        currentResolutionIndex = i;
    }
}
```

Figure 102 From SettingsMenu.cs Start() Function

The options list is then added to the resolution dropdown.

```
resolutionDropdown.AddOptions(options);
```

Figure 103 From SettingsMenu.cs Start() Function

To check if the resolutions match with the screen's current resolution the if statement will update the current resolution index.

```
if(resolutions[i].width == Screen.currentResolution.width && resolutions
[i].height == Screen.currentResolution.height)
{
    currentResolutionIndex = i;
}
```

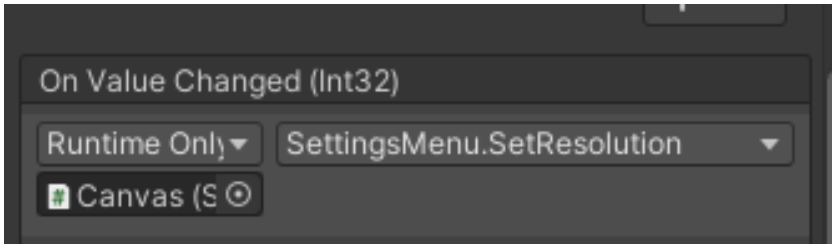
Figure 104 From SettingsMenu.cs Start() Function

The resolution value is set, and the value is refreshed so the correct value is shown.

```
resolutionDropdown.AddOptions(options);
resolutionDropdown.value = currentResolutionIndex;
resolutionDropdown.RefreshShownValue();
```

Figure 105 From SettingsMenu.cs Start() Function

When the dropdown value is updated the “SetResolution()” function is used.



It uses the “Screen.SetResolution()” function to make the screen resolution equal to what was selected.

```
public void SetResolution(int resolutionIndex)
{
    Resolution resolution = resolutions[resolutionIndex];
    Screen.SetResolution(resolution.width, resolution.height, Screen.fullScreen);
}
```

Figure 106 SetResolution() Function in SettingsMenu.cs

5.18.2 SetQuality() Function

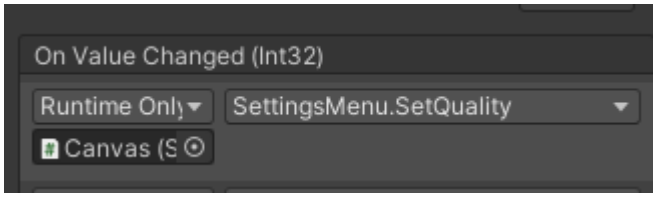
This function will change the quality of the game to fit the selected value.

The “qualityIndex” variable is 0 for Low, 1 for Medium and 2 for High. When accessing the “QualitySettings” the quality is set to the value selected.

Figure 107 Example of what the dropdown looks like

```
public void SetQuality (int qualityIndex)
{
    QualitySettings.SetQualityLevel(qualityIndex);
}
```

Figure 108 SetQuality() Function



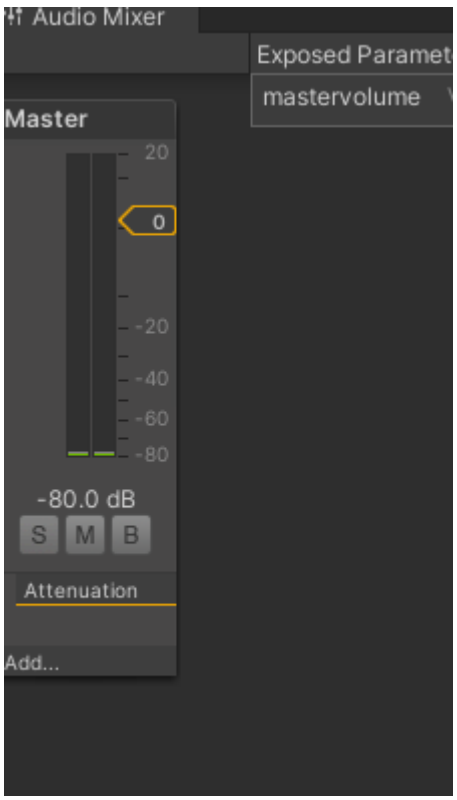
5.18.3 SetVolume()

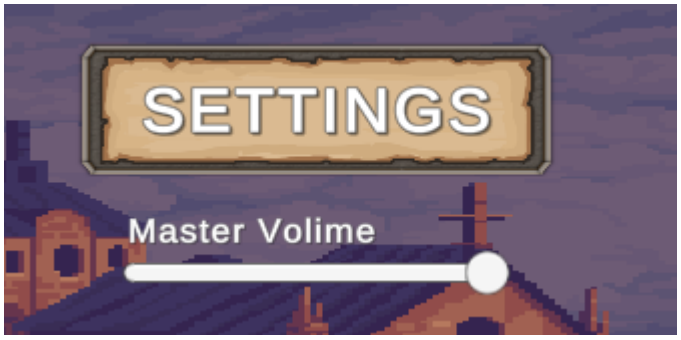
These functions take in float variables.

```
public void SetMasterVolume(float volume)
{
    audioMixer.SetFloat("mastervolume", volume);
}

public void SetMusicVolume(float volume)
{
    audioMixer.SetFloat("musicvolume", volume);
}

public void SetSFXVolume(float volume)
{
    audioMixer.SetFloat("sfxvolume", volume);
}
```

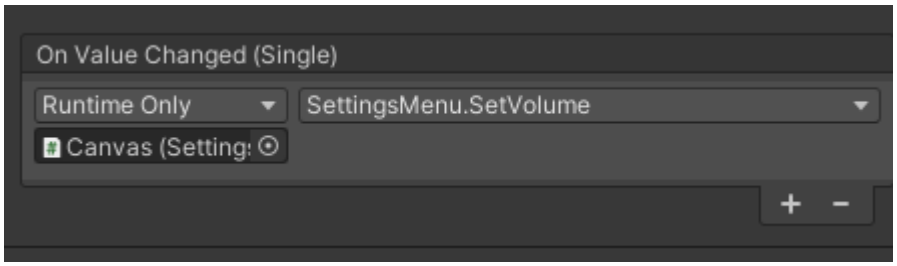
Figure 109 SetVolume() Functions in SettingsMenu.cs

In the audio mixer the value of the master volume is called “mastervolume”. Using the “SetFloat” function above, the master volume becomes the float value passed into the “SetVolume()” function.





Using the slider “On Value Changed” augment, the value on the slider is passed into the function.



The minimum value of the slider is -80 and the max is 0 matching with the Master Audio Mixer.

5.18.4 Updated SetVolume() Saves between scenes.

An “AudioManager.cs” Script was created to handle audio in the scene.

“SetVolume()” has been broken up into 3 separate functions:

“SetMasterVolume()”, “SetMusicVolume()” and “SetSFXVolume()”.

The Variables that control the mixer groups were turned into const variables, the naming convention for const variables are all capital letters. These values will not change.

```
//The name of the variables that controls the parts of the mixer
public const string MASTER_MIXER = "mastervolume";
public const string MUSIC_MIXER = "musicvolume";
public const string SFX_MIXER = "sfxvolume";
```

Figure 110 From SettingsMenu.cs

In the “Awake()” function, when a value on the slider is changed, the Set x volume functions are used for whichever slider was changed.

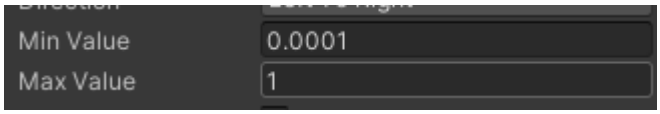
```
0 references
void Awake()
{

    masterSlider.onValueChanged.AddListener(SetMasterVolume);
    musicSlider.onValueChanged.AddListener(SetMusicVolume);
    sfxSlider.onValueChanged.AddListener(SetSFXVolume);


}
```

Figure 111 From SettnigsMenu.cs

The math equation here is used to change the volume logarithmically as the volume on the mixer changes logarithmically. The slider value was also changed to fit this version, from -80 to 0 to between 0 and 1. The unity function “SetFloat()” changes the Audio Mixer’s variable (“MASTER_MIXER”) to the value of the math equation.

```
public void SetMasterVolume(float volume)
{

    audioMixer.SetFloat(MASTER_MIXER, Mathf.Log10(volume) * 20);
}

1 reference
public void SetMusicVolume(float volume)
{
    audioMixer.SetFloat(MUSIC_MIXER, Mathf.Log10(volume) * 20);
}

1 reference
public void SetSFXVolume(float volume)
{
    audioMixer.SetFloat(SFX_MIXER, Mathf.Log10(volume) * 20);
}
```

Figure 112 From SettingsMenu.cs



In the “OnDisable()” function the “SetFloat()” function will be used to store the values to the “PlayerPrefs” which is player preferences that are stored on the computer. The “AudioManager.MASTER_KEY” is the variable name for the master volume mixer, and the slider value is “masterSlider.value”.

This is saving the sound details to the player preferences.

```
//when exiting the values for the master, music and sfx slider will be stored in
player preferences.
0 references
void OnDisable()
{
    PlayerPrefs.SetFloat(AudioManager.MASTER_KEY, masterSlider.value);
    PlayerPrefs.SetFloat(AudioManager.MUSIC_KEY, musicSlider.value);
    PlayerPrefs.SetFloat(AudioManager.SFX_KEY, sfxSlider.value);
}
```

Figure 113 From SettingsMenu.cs

5.19 AudioManager.cs.

In the “AudioManager.cs” script the settings are loaded, since an “Audio Manager” is going to be in every scene.

In the “LoadSound()” function the value of the float that’s stored in player preferences is acquired with the Unity function “GetFloat()”. If it can’t find a value for “MASTER_KEY” it will default to 1 (100% volume). The volume of each mixer is then set with “SetFloat()”, and the value is put into the logarithmic equation to get the correct value.

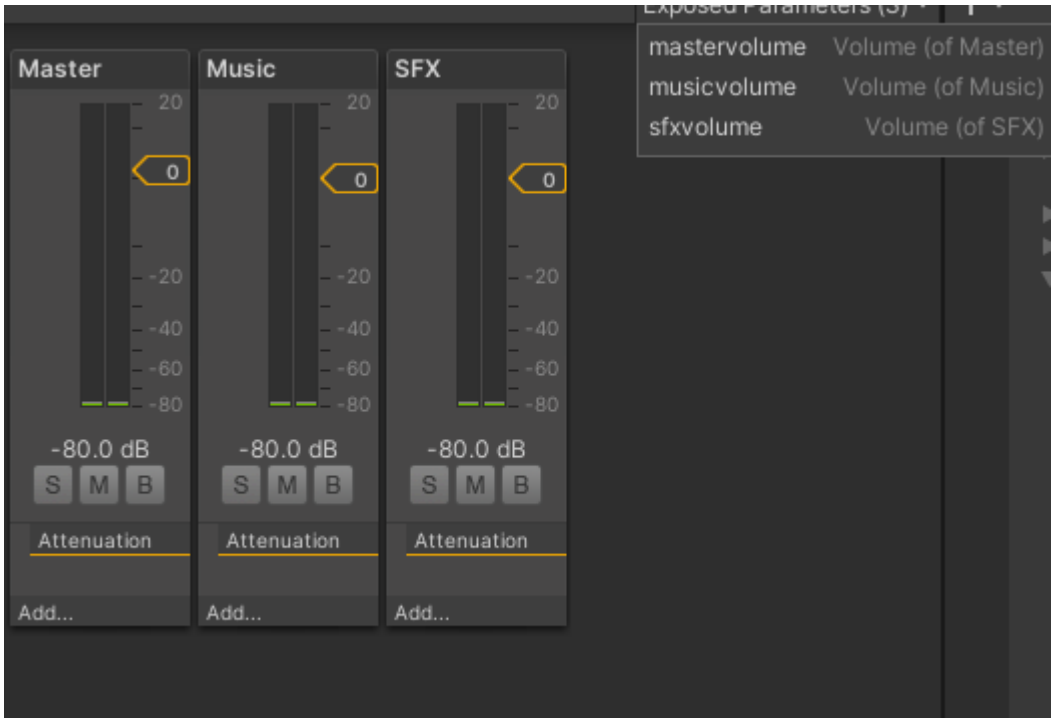
```
public const string MASTER_KEY = "masterVolume";
public const string MUSIC_KEY = "musicVolume";
public const string SFX_KEY = "sfxVolume";

1 reference
void LoadSound()
{
    float masterVol = PlayerPrefs.GetFloat(MASTER_KEY, 1f);
    float musicVol = PlayerPrefs.GetFloat(MUSIC_KEY, 1f);
    float sfxVol = PlayerPrefs.GetFloat(SFX_KEY, 1f);

    mixer.SetFloat(SettingsMenu.MASTER_MIXER, Mathf.Log10(masterVol) * 20);
    mixer.SetFloat(SettingsMenu.MUSIC_MIXER, Mathf.Log10(musicVol) * 20);
    mixer.SetFloat(SettingsMenu.SFX_MIXER, Mathf.Log10(sfxVol) * 20);
}
```

Figure 114 From AudioManager.cs

The “AudioManager” object has the “AudioManager.cs” script on it. Once the scene loads it will check if there is an instance of “AudioManager”. If there is not, it will set an instance that will not be destroyed when scene is loaded. If there is an instance, it will destroy the game object to make sure there is only 1 “AudioManager” per scene.

```
void Awake()
{
    if(instance == null)
    {
        instance = this;
        DontDestroyOnLoad(gameObject);
    }
    else
    {
        Destroy(gameObject);
    }

    LoadSound();
}
```

The groups of this mixer are set up so the Master controls both the Music and SFX sounds but they don't have any impact on each other.



5.19.1 Play sound Functions

The below functions are how the author would start and stop different sounds.

```
public void WalkSFX()
{
    AudioClip clip = clips[3];
    SFXSource.clip = clip;
    SFXSource.loop = true;
    SFXSource.Play();
}

0 references
public void StopWalkSFX()
{
    AudioClip clip = clips[3];
    SFXSource.clip = clip;
    SFXSource.Stop();
}

0 references
public void DashSFX()
{
    AudioClip clip = clips[4];
    SFXSource.PlayOneShot(clip);
}
```

```
public void AttackSFX()
{
    AudioClip clip = clips[1];
    SFXSource.PlayOneShot(clip);
}

0 references
public void JumpSFX()
{
    AudioClip clip = clips[0];
    SFXSource.PlayOneShot(clip);
}

0 references
public void LandSFX()
{
    AudioClip clip = clips[2];
    SFXSource.PlayOneShot(clip);
}
```

```
public void Lvl1BgMusic()
{
    AudioClip clip = clips[5];
    MusicSource.clip = clip;
    MusicSource.loop = true;
    MusicSource.Play();
}
0 references
public void Lvl2BgMusic()
{
    AudioClip clip = clips[6];
    MusicSource.clip = clip;
    MusicSource.loop = true;
    MusicSource.Play();
}

0 references
public void BossMusic()
{
    AudioClip clip = clips[7];
    MusicSource.clip = clip;
    MusicSource.loop = true;
    MusicSource.Play();
}
```

```
0 references
public void StopBgMusic()
{
    AudioClip clip1 = clips[5];
    AudioClip clip2 = clips[6];
    AudioClip clip3 = clips[7];
    MusicSource.clip = clip1;
    MusicSource.Stop();
    MusicSource.clip = clip2;
    MusicSource.Stop();
    MusicSource.clip = clip3;
    MusicSource.Stop();
}
```

Figure 115 From AudioManager.cs

5.20 BgMusic.cs

This script will play the correct background music when a new scene is loaded. It takes in a “SerializeField” float variable.

In the “Awake()” and “Start()” functions the function “SetBgMusic()” is started.

“SetBgMusic()” first stops any currently playing background music then it checks the value of the “SerializeField” float.

If the number is 1 it will play the village background music,

If it is 2 it will play the mines background music, and

If it is 3 it will play the boss fight music.

```
[SerializeField] float bgMusic;

0 references
void Awake()
{
    SetBgMusic();
}


0 references
void Start()
{
    SetBgMusic();
}

2 references
void SetBgMusic()
{

    AudioManager.instance.StopBgMusic();

    if (bgMusic == 1f)
    {
        AudioManager.instance.Lvl1BgMusic();
    }
    else if (bgMusic == 2f)
    {
        AudioManager.instance.Lvl2BgMusic();
    }
    else if(bgMusic == 3f)
    {
        AudioManager.instance.BossMusic();
    }



}
```

Figure 116 From BgMusic.cs

5.21 Inheritance

Inheritance is used to ensure there is no repeat of code. This can happen when two or more classes use the same or similar functions. In this situation the “Character” class is the parent class, and the “Player”, “NPC”, and “Enemy” classes are the child classes that inherit from “Character”. For example, all characters will have health, speed, attack damage, and jump force. Instead of making a Player class with these variables and then creating an NPC and Enemy class with the same variables, the “Character” class exists and can give its child the information. This works the same with functions not just variables.

5.21.1 Character.cs

This class will be used to hold shared functions and variables that its child classes use. All child objects will need to turn around so the “Flip()”, and “ChangeDirection()” functions are here.

```
protected private void ChangeDirection()
{
    if(!facingRight && direction > 0)
    {
        Flip();
    }
    else if (facingRight && direction < 0)
    {
        Flip();
    }
}
```

Figure 117 ChangeDirection() Function From Character.cs

```
//method used to change the direction a rigid body is facing
protected virtual void Flip()
{
    //OLD VERSION OF CHANGING DIRECTION
    //this made the camera not work correctly.

    // facingRight = !facingRight;

    // Vector3 theScale = transform.localScale;
    // theScale.x *= -1;
    // transform.localScale = theScale;
    // cameraFollowObject.Turn();

    if(facingRight)
    {
        Vector2 rotator = new Vector2(transform.rotation.x, 180f);
        transform.rotation = Quaternion.Euler(rotator);
        facingRight = !facingRight;
    }
    else
    {
        Vector2 rotator = new Vector2(transform.rotation.x, 0f);
        transform.rotation = Quaternion.Euler(rotator);
        facingRight = !facingRight;
    }

}
```

Figure 118 Flip() Function From Character.cs

All characters should be able to take damage and die so the “TakeDamage()”, “Die()” and “DisableOnDeath()” functions are here.

```
public virtual void TakeDamage(float damage)
{
    currentHealth -= damage;

    if(currentHealth <= 0)
    {
        Die();
    }
}

protected virtual void Die()
{

    //disable enemy
    StartCoroutine(DisableOnDeath());
}

public virtual IEnumerator DisableOnDeath()
{
    yield return new WaitForSeconds(.8f);
    GetComponent<Collider2D>().enabled = false;
    GetComponent<SpriteRenderer>().enabled = false;
    this.enabled = false;
    yield return null;
}
```

Figure 119 TakeDamage(), DisableOnDeath, Die() Functions From Character.cs

All characters should be able to attack. However, for this project the author only has time to have Enemy and Player attack. This function would be an abstract void function instead of virtual void, because each of its child classes would attack differently.

```
protected virtual void Attack()
{

}
```

Each character should not be able to move while the game is paused. The “Update()” and “FixedUpdate()” functions do this. In the future the creator wants the player to have allies to fight alongside. However, for the scope of this project it is not possible. The “Start()” function will set each of the character’s health and speed and make sure they are not attacking.

```
public virtual void Start()
{
    speed = maxSpeed;
    attacking = false;
    currentHealth = maxHealth;
}


public virtual void Update()
{
    if(PauseScript.isPaused) return;


}

public virtual void FixedUpdate()
{
    if(PauseScript.isPaused) return;

}
```

Figure 120 Start(), Update() & FixedUpdate() Functions From Character.cs

The “OnDrawGizmos()” function draws wireframes that are visible in the editor and not in the game. This is helpful to see where the attack points / range etc of characters are.

```
//used to draw a gizmo that is visible to the editor but not in game.
protected virtual void OnDrawGizmos()
{

    Gizmos.DrawSphere(groundCheck.position, radOfCircle);
    Gizmos.DrawSphere(wallCheck.position, radOfCircle);
    if(attackPoint == null)
        return;
    Gizmos.DrawWireSphere(attackPoint.position, attackRange);
}
```

Figure 121 OnDrawGizmos() Function from Character.cs

```
protected virtual void OnDrawGizmos()
{
    if(attackPoint == null)
        return;
    Gizmos.DrawWireSphere(attackPoint.position, attackRange);
}
```

Figure 122 Updated OnDrawGizmos() Function

All characters should be able to be knocked back. This “Knockback()” function takes in a rigid body and a float. It starts a Coroutine called “KnockbackCoroutine()” that takes a rigid body and float.

```
0 references
public virtual void Knockback(Rigidbody2D myRigidbody, float knockbackTime)
{

    StartCoroutine(KnockbackCoroutine(myRigidbody, knockbackTime));

}
```

Figure 123 Knockback() Function From Character.cs

This Coroutine will check if it has a rigid body, it will start the Coroutine “FlashCoroutine()” wait for the duration of the “knockbackTime” variable and then reset the rigid body’s velocity.

```
protected virtual IEnumerator KnockbackCoroutine(Rigidbody2D myRigidbody, float knockbackTime)
{
    if(myRigidbody != null)
    {
        StartCoroutine(FlashCoroutine());
        yield return new WaitForSeconds(knockbackTime);
        myRigidbody.velocity = Vector2.zero;

        //Vector2.zero is short hand for writing Vector2(0,0);
    }
}
```

Figure 124 KnockbackCoroutine() Function From Character.cs

The “FlashCoroutine()” function will:

Turn off the 2D trigger collider.

Flash for the “numberOfFlashes” variable number of times.

Turn on the 2D trigger collider.

```
protected private IEnumerator FlashCoroutine()
{
    int flashes = 0;
    triggerCollider2D.enabled = false;
    while(flashes < numberOfFlashes)
    {
        sprite.color = flashColour;
        yield return new WaitForSeconds(flashDuration);
        sprite.color = normalColour;
        yield return new WaitForSeconds(flashDuration);
        flashes++;
    }
    triggerCollider2D.enabled = true;
}
```

5.21.2 Player.cs

Player Variables

Below are the variables the player uses under the headings.

```
[Header("Rigidbody, Animator")]
private Rigidbody2D rb;
private Animator myAnimator;

[Header("Dash Details")]
[SerializeField]private TrailRenderer tr;
[SerializeField]private float dashingPower = 40f;
[SerializeField]private float dashingTime = .1f;
private bool canDash = true;
private bool isDashing;
private float dashingCooldown = 1f;

[Header("Extra Jump Details")]
private float coyoteTime = 0.2f;
private float coyoteTimeCounter;
private bool isJumping;
private bool isFalling;

[Header("Extra Movement Details")]
private bool isWallSliding;
private float wallSlidingSpeed = 2f;
private float originalGravity;
private bool isWalking;
```

```
[Header("Wall Jump Details")]
[SerializeField]private float wallJumpingDirection;
[SerializeField]private Vector2 wallJumpingPower = new Vector2(9f,9f);
[SerializeField]private float wallJumpingTime;
private float maxWallJumpingTime = .1f;
[SerializeField]private bool canWallJump = true;
[SerializeField]private bool isWallJumping;
[SerializeField]private float wallJumpingCooldown = .1f;

[Header("Interactables")]
[SerializeField] public float interactRange = 3f;
[SerializeField] private LayerMask NPCLayers;

[Header("One Way Platform")]
public bool fallThrough;

[Header("Camera")]
[SerializeField] private GameObject cameraFollow;
private CameraFollowObject cameraFollowObject;
```

The player will need to check if they are on the ground or against a wall so the "IsGrounded()" and "IsWalled()" functions are here.

```
//bool to check if the player is on the ground. returns true or false.
protected private bool IsGrounded()
{
    //drawing a small circle under the rigid body to check if its touching the grou
    return Physics2D.OverlapCircle(groundCheck.position, radOfCircle, groundMask);
}

//bool. checking for player / wall overlap. returns true or false.
protected private bool IsWalled()
{
    return Physics2D.OverlapCircle(wallCheck.position, radOfCircle, wallLayer);
}
```

Figure 125 IsGrounded() & IsWalled() Functions From Player.cs

Function overrides

The functions that use the override word will be able to use the "Character" classes version of the function and add onto it or have a new function with the same name.

```
#region Overrides
public override void Start()
{
```

The functions that are overridden in "Player.cs" are:

"Start()",

```
public override void Start()
{
    rb = GetComponent<Rigidbody2D>();
    myAnimator = GetComponent<Animator>();
    base.Start();
    maxJumps = 2f;
    originalGravity = rb.gravityScale;
    cameraFollowObject = cameraFollow.GetComponent<CameraFollowObject>();
}
```

Figure 126 Start() From Player.cs

"Update()"

Update is shown and explained previously in the "Player.cs Update()" or "PlayerScript.cs Update()" function.

“FixedUpdate()”

Fixed update is explained previously in the “Player.cs FixedUpdate()” or “PlayerScript.cs Update()” function.

```
public override void FixedUpdate()
{
    base.FixedUpdate();
    if(isDashing || isWallJumping) return;
    //controlling the movement of the player, changing the x velocity.
    if(IsWalled()) StartCoroutine(WallJumpCooldown());
    if(direction > 0 || direction < 0) ChangeDirection();
    rb.velocity = new Vector2(direction * speed, rb.velocity.y);

}
```

Figure 127 FixedUpdate() From Player.cs

“Attack()”

Enemies and the player will have different attacks. Below is the player’s attack. This is also explained previously in the “Player.cs Attack()” function.

```
protected override void Attack()
{
    //detect enemies
    Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(attackPoint.position, attackRange, enemyLayers);
    //damage them
    foreach(Collider2D enemy in hitEnemies)
    {
        enemy.GetComponent<Enemy>().TakeDamage(attackDamage);

    }
}
```

Figure 128Attack() Function From Player.cs

“TakeDamage()”

Each character has a different animation. This function adds the animation for the character to the “TakeDamage” function.

```
public override void TakeDamage(float damage)
{
    base.TakeDamage(damage);
    //play hurt anim
    myAnimator.SetTrigger("Hurt");
}
```

Figure 129 TakeDamage() Function From Player.cs

"DisableOnDeath()" adds the player death animation.

```
public override IEnumerator DisableOnDeath()
{
    myAnimator.SetBool("IsDead", true);
    yield return new WaitForSeconds(.8f);
    GetComponent<Collider2D>().enabled = false;
    GetComponent<SpriteRenderer>().enabled = false;
    this.enabled = false;
    yield return null;

}
```

Figure 130 DisableOnDeath() Function From Player.cs

"Flip()" makes the dash go in the correct direction, since the new way of flipping the player doesn't use the player's X scale anymore.

```
protected override void Flip()
{
    base.Flip();
    dashingPower *= -1f;
}
```

Figure 131 Flip( )Function From Player.cs

"OnDrawGizmos()" is used to show the interact range of NPCs.

```
    protected override void OnDrawGizmos()
{
    base.OnDrawGizmos();
    Gizmos.DrawWireSphere(transform.position, interactRange);
}
```

Figure 132 OnDrawGizmos() Function From Player.cs

Animation Control

The “PlayerAnimations()” function could be improved by adding Character States instead of functions to check if the correction animations is played. However, the time constraints of this project did not allow a rehash of what was already done with this section. It is used to make sure the player is using the correct animation.

```
private void PlayerAnimations()
{
    //making sure the player is using the correct animation.
    if(!isWallJumping)
    {
        IsMoving();
        WallSlide();
        IsFalling();
        IsWalling();
        IsJumping();


    }
    else if (isWallJumping)
    {
        IsJumping();
        IsMoving();
        IsFalling();

    }
}
```

Figure 133 PlayerAnimations() Function From Player.cs

Player Input

The player input classes explained earlier in the chapter are placed in the “Player.cs” class:

“Move()”, “Fire()”, “Jump()”, “Interact()”, “Dash()”, and “CheckInteractions()”.

Coroutines

The coroutines have been explained previously. The “WallJumpCooldown()”, “WallBounce()”, and “Dash()” coroutine functions are placed in this class.

Extra functions

The “IsGrounded()”, “IsWalled()” and “WallSlide()” functions are placed in this class.

5.21.3 NPC.cs

The “NPC.cs” is currently empty, However, in the future this can be added to.

5.21.4 Enemy.cs

The variables used in the “Enemy.cs” are shown in the below image.

```
[SerializeField] private string enemyName;

private Transform target;
[SerializeField] private float chaseRange;
[SerializeField] private Transform chase;

private bool alive;

[Header("Rigidbody, Animator")]
private Rigidbody2D rb;
private Animator myAnimator;
Collider2D otherTag;
```

Figure 134 From Enemy.cs

Function overrides

The functions that use the override word will be able to use the “Character” classes version of the function and add onto it or have a new function with the same name.

The functions that have been overridden in “Enemy.cs” are:

“Start()” adds to the “Character.cs Start()” function it gets the rigid body and animator components, sets the “alive” variable to true, and assigns the target for the enemy to chase.

```
0 references
public override void Start()
{
    rb = GetComponent<Rigidbody2D>();
    myAnimator = GetComponent<Animator>();
    base.Start();
    alive = true;
    target = GameObject.FindGameObjectWithTag("Player").GetComponent<Transform>();
    Collider2D otherTag = GameObject.FindGameObjectWithTag("Player").GetComponent<Collider2D>();
}
```

Figure 135 Start() Function From Enemy.cs

"Update()" plays the enemy "Idle" animation.

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

    if(currentHealth >= 0f)
    {
        myAnimator.SetBool("Idle", true);
    }


}
```

Figure 136 Update() Function From Enemy.cs

"FixedUpdate()" uses the "Move()" function that is discussed later.

```
public override void FixedUpdate()
{
    base.Update();
    Move();
}
```

Figure 137 FixedUpdate() Function From Enemy.cs

"TakeDamage()" adds the "FlashCorouitine()".

```
0 references
public override void TakeDamage(float damage)
{
    base.TakeDamage(damage);

    StartCoroutine(FlashCoroutine());

}
```

Figure 138 TakeDamage() Function From Enemy.cs

"DisableOnDeath()" sets alive to false and uses the animation.

```
public override IEnumerator DisableOnDeath()
{
    //die animation
    alive = false;
    myAnimator.SetBool("IsDead", true);
    yield return new WaitForSeconds(.8f);
    GetComponent<Collider2D>().enabled = false;
    GetComponent<SpriteRenderer>().enabled = false;
    this.enabled = false;
    yield return null;
}
```

Figure 139 DisableOnDeath() Function From Eenemy.cs

"OnDrawGizmos()" adds a circle in the Unity inspector that shows the chase range of an enemy.

```
0 references
protected override void OnDrawGizmos()
{
    base.OnDrawGizmos();
    Gizmos.DrawWireSphere(chase.position, chaseRange);
}
```

Figure 140 OnDrawGizmos() Function From Enemy.cs

Extra functions

The "Attack()" function takes in a 2D collider.

If the attack "Cooldown" is complete it checks if the collided object has the tag "Player".

Then does damage to the player and starts the attack "Cooldown".

```
protected void Attack(Collider2D other)
{
    if(Time.time >= nextAttackTime)
    {
        if(other.gameObject.CompareTag("Player"));
        {

            other.GetComponent<Player>().TakeDamage(attackDamage);
            nextAttackTime = Time.time + 1f / attackRate;
        }
    }
}
```

Figure 141 Attack() Function From Enemy.cs

The Move() function will check:

- If The player's position is inside the "chaseRange" variable.
- Then it will move towards the target (the player).

```
private void Move()
{
    if(Vector2.Distance(rb.position, target.position) < chaseRange)
    {
        transform.position = Vector2.MoveTowards(transform.position, target.
        position, speed * Time.deltaTime);
    }

}
```

Figure 142 Move() Function From Enemy.cs

The "OnTriggerEnter2D()" function checks:

- If the trigger has been collided with, and
- If the enemy is alive, attack.

```
0 references
private void OnTriggerEnter2D(Collider2D other)
{
    if(alive)
    {
        Attack(other);

    }
}
```

Figure 143 OnTriggerEnter2D() Function From Enemy.cs

5.22 SceneController.cs

This script is attached to the “GameManger” object. This object holds the “AudioManager”, the “Canvas”, the “MenuController”, and the “EventSystem”. These are necessary in each scene.

```
public static SceneController instance;

0 references
private void Awake()
{
    if(instance == null)
    {
        instance = this;
        DontDestroyOnLoad(gameObject);
    }
    else
    {
        Destroy(gameObject);
    }

}
```

Figure 144 SceneController.cs Awake() Function

When a scene is loaded, the “DontDestroyOnLoad()” function keeps the object from the previous scene. If there is another object with this script attached, the object will be destroyed.

The “SceneController.cs” script also contains the functions:

“NextLevel()” which loads the next scene.

“LoadScene()” which loads a scene by name, and

“MainMenu()” which sends the player to the main menu.

```
0 references
public void NextLevel()
{
    SceneManager.LoadSceneAsync(SceneManager.GetActiveScene().buildIndex + 1);
}

0 references
public void LoadScene(string sceneName)
{
    SceneManager.LoadSceneAsync(sceneName);
}


0 references
public void MainMenu()
{
    SceneManager.LoadScene(0);
}
```

Figure 145 SceneController.cs NextLevel(), LoadScene() & MainMenu() Functions

5.23 LoadNextArea.cs

This script is used in tandem with the “SceneManager.cs” script. However, it also toggles on and off any objects that should not be in the current scene.

This script uses:

2 private “GameObject” variables one for the player and one for the starting position.

A “SerializeField” Boolean and “SerializeField” string variable.

4 “SerializeField” “GameObject” variables.

```
private GameObject player;
private GameObject startPos;

[SerializeField] bool goNextLevel;
[SerializeField] string levelName;

[SerializeField]private GameObject scene1;
[SerializeField]private GameObject scene2;
[SerializeField]private GameObject scene3;
[SerializeField]private GameObject scene4;
```

The “Start()” function finds 6 different variables with the specified tags. It then sets the players position to the position of the “startPos” object.

```
void Start()
{
    player = GameObject.FindGameObjectWithTag("Player");
    startPos = GameObject.FindGameObjectWithTag("StartPos");

    player.transform.position = startPos.transform.position;

    scene1 = GameObject.FindGameObjectWithTag("Scene1");
    scene2 = GameObject.FindGameObjectWithTag("Scene2");
    scene3 = GameObject.FindGameObjectWithTag("Scene3");
    scene4 = GameObject.FindGameObjectWithTag("Scene4");
}
```

Figure 146 LoadNextArea.cs Start() Function

The “Update()” function will make the correct scene display to the player.

```
void Update()
{
    if(SceneManager.GetActiveScene().buildIndex == 1)
    {
        UseCam1();
    }

    if(SceneManager.GetActiveScene().buildIndex == 2)
    {
        UseCam2();
    }
    if(SceneManager.GetActiveScene().buildIndex == 3)
    {
        UseCam3();

    }
    if(SceneManager.GetActiveScene().buildIndex == 4)
    {
        UseCam4();
    }
}
```

Figure 147 LoadNextArea.cs Update() Function

"UseCam1()" makes the scene 1 enabled and all other scenes disabled.

```
1 reference
void UseCam1()
{
    scene1.SetActive(true);
    scene2.SetActive(false);
    scene3.SetActive(false);
    scene4.SetActive(false);

}
```

Figure 148 LoadNextArea.cs UseCam1 Function

"UseCam2()" makes the scene 2 enabled and all other scenes disabled.

```
void UseCam2()
{
    scene1.SetActive(false);
    scene2.SetActive(true);
    scene3.SetActive(false);
    scene4.SetActive(false);
}
```

Figure 149 LoadNextArea.cs UseCam2 Function

"UseCam3()" makes the scene 3 enabled and all other scenes disabled.

```
1 reference
void UseCam3()
{
    scene1.SetActive(false);
    scene2.SetActive(false);
    scene3.SetActive(true);
    scene4.SetActive(false);
}
```

Figure 150 LoadNextArea.cs UseCam3 Function

"UseCam4()" makes the scene 4 enabled and all other scenes disabled.

```
1 reference
void UseCam4()
{
    scene1.SetActive(false);
    scene2.SetActive(false);
    scene3.SetActive(false);
    scene4.SetActive(true);
}
```

Figure 151 LoadNextArea.cs UseCam4 Function

The author attempted to switch the priority of the current camera to change between cameras. However, the time constraints of this project did not allow for them to complete this.

```
// void SwitchPriority(CinemachineVirtualCamera camera)
// {
//     camera.Priority = 11;

// }

// void SwitchPriorityNormal(CinemachineVirtualCamera camera)
// {
//     camera.Priority = 10;
// }
```

"LoadNextArea.cs" is placed on a 2D trigger collider. The "OnTriggerEnter2D()" function:

Takes in a Collider2D collision.

Checks if the collision is done by the object tagged "Player".

If the goNextLevel variable is true, the next level will be loaded using the SceneController.

Otherwise, it will load the "levelName" variable scene.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if(collision.CompareTag("Player"))
    {
        if(goNextLevel)
        {
            SceneController.instance.NextLevel();
        }
        else
        {
            SceneController.instance.LoadScene(levelName);
        }
    }
}
```

Figure 152LoadNextArea.cs OnTriggerEnter2D() Function

5.24 HealthBar.cs

This function uses a reference to a slider.

"HealthBar.cs" changes the value on the health bar slider depending on the players "currentHealth" variable.

```
using UnityEngine.UI;

0 references
public class HealthBar : MonoBehaviour
{

    public Slider slider;

    0 references
    public void SetMaxHealth(float health)
    {
        slider.maxValue = health;
        slider.value = health;
    }
    0 references
    public void SetHealth(float health)
    {
        slider.value = health;
    }

}
```

The "SetMaxHealth()" function takes in a float variable. it will set the value of the slider to its max value. It is used in the "Player.cs Start()" to set the max health to the "maxHealth" variable.

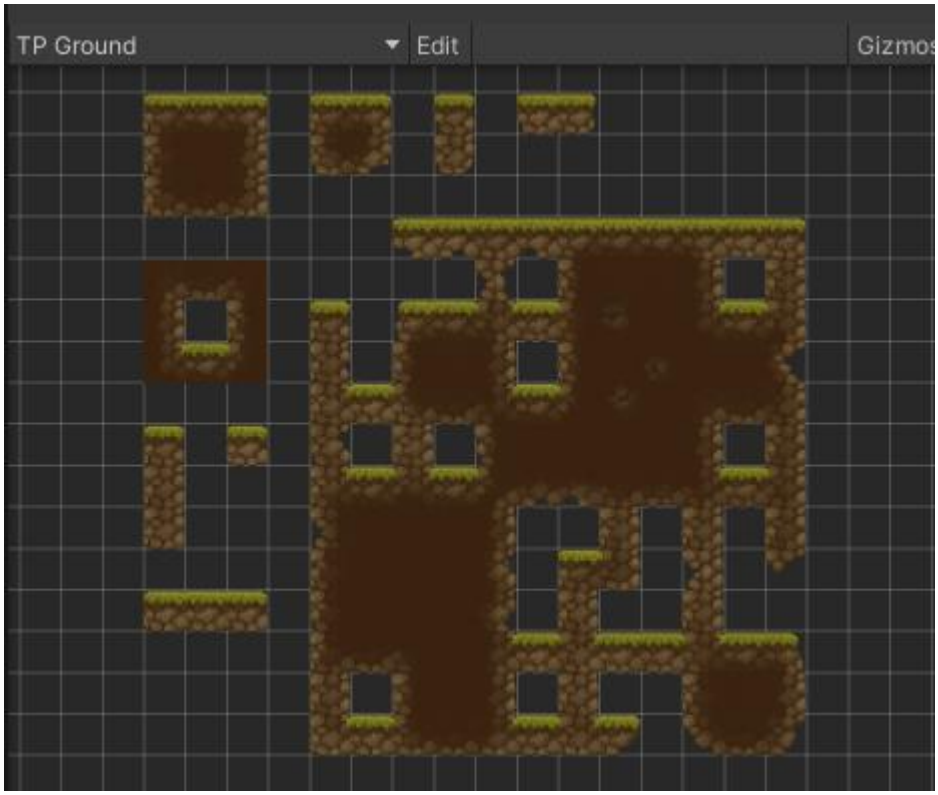
```
public override void Start()
{

    rb = GetComponent<Rigidbody2D>();
    myAnimator = GetComponent<Animator>();
    base.Start();
    healthBar.SetMaxHealth(maxHealth);
    maxJumps = 2f;
    originalGravity = rb.gravityScale;
    cameraFollowObject = cameraFollow.GetComp
}
```

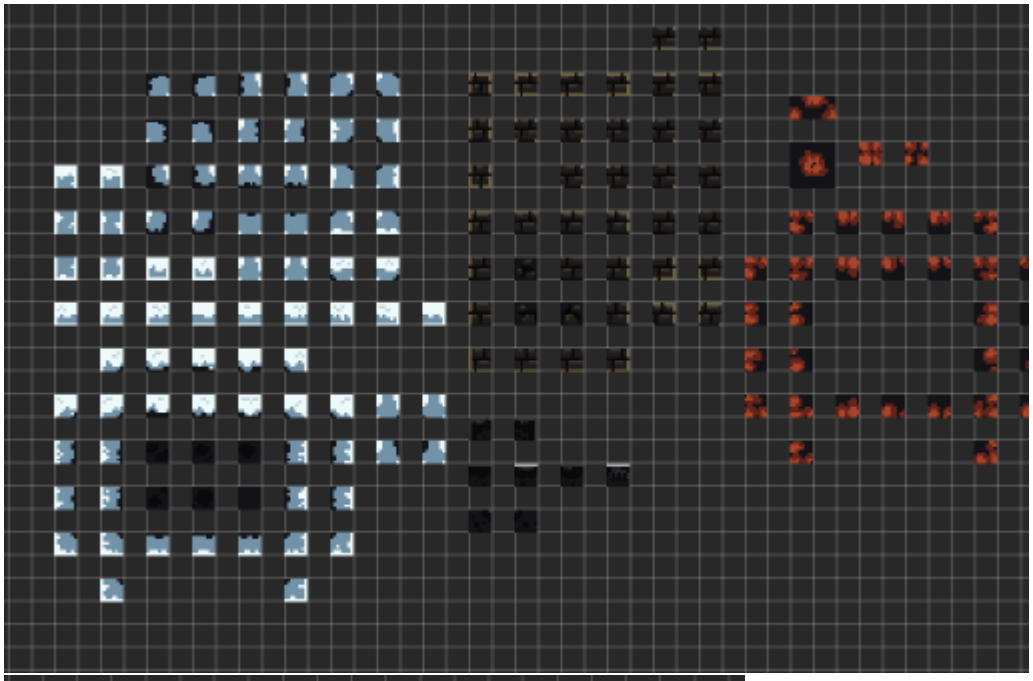
The “SetHealth()” function takes in a float variable. It sets the value of the slider to that of the “currentHealth” variable.

```
0 references
public override void TakeDamage(float damage)
{
    base.TakeDamage(damage);
    healthBar.SetHealth(currentHealth);
}
```

5.25 Tile map

Tile maps are used to create 2D worlds. They allow the creator to select the tile they would like to place and draw the tiles into the scene.





5.26 Level Creation

5.26.1 The Crystal Village 1

Figure 153 The Crystal Village 1

5.26.2 The Crystal Village 2

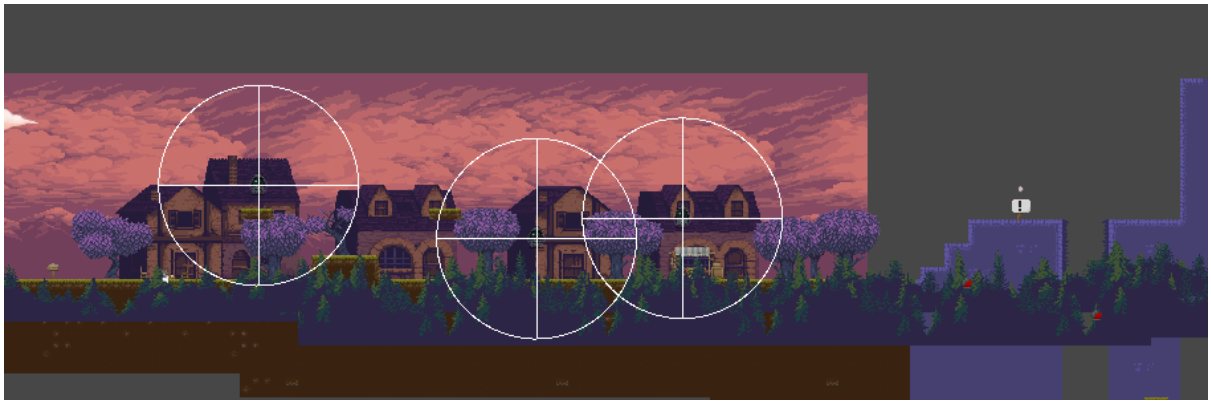
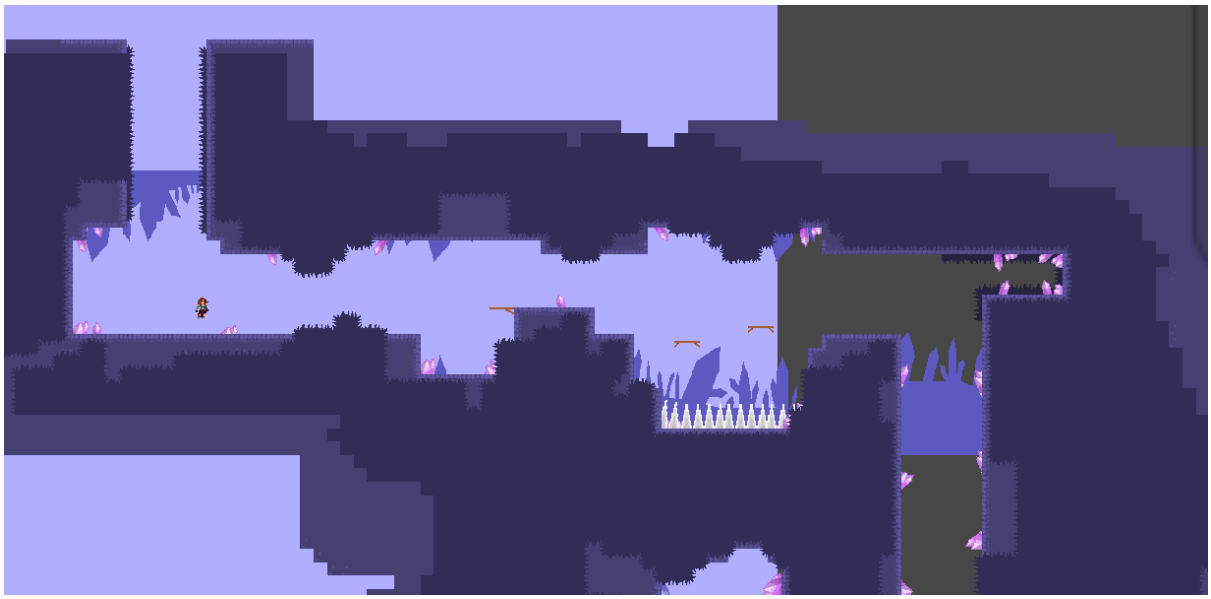
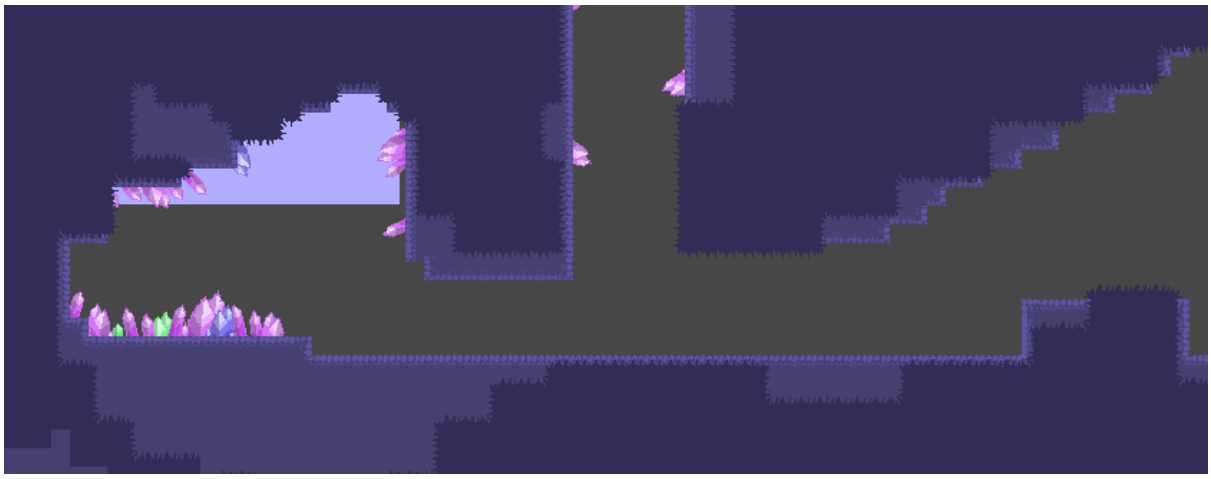
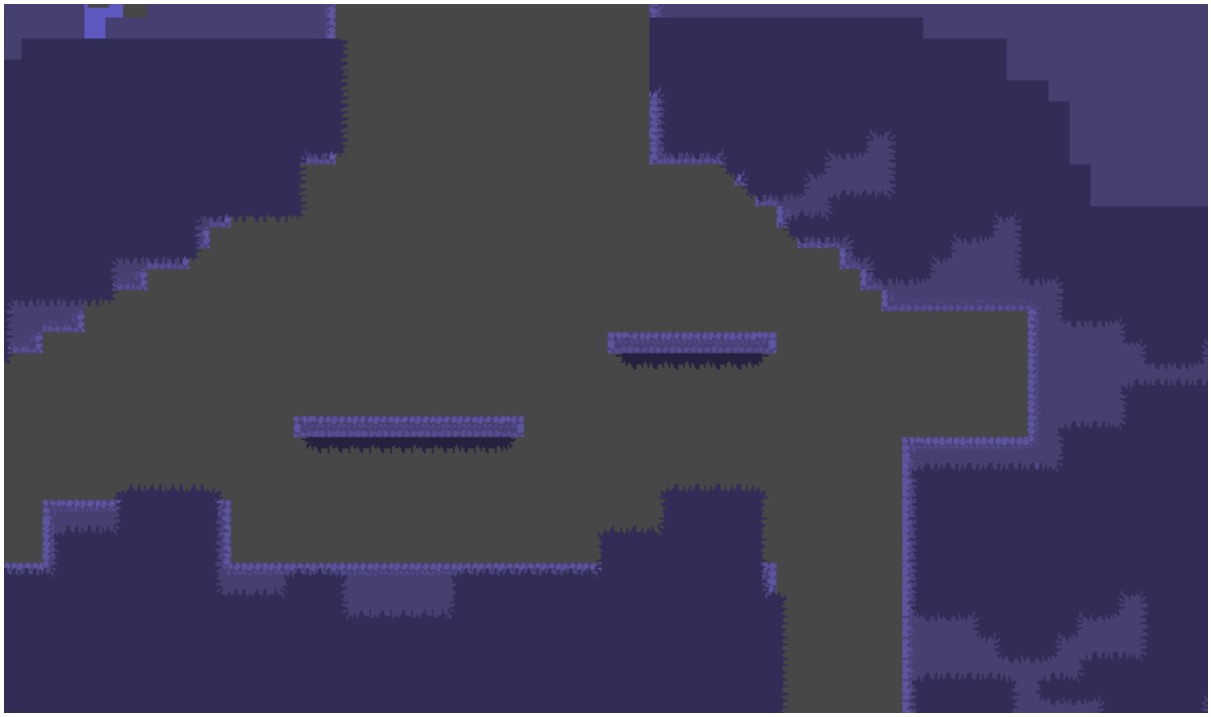


Figure 154 The Crystal Village 2

5.26.3 The Crystal Mines 1







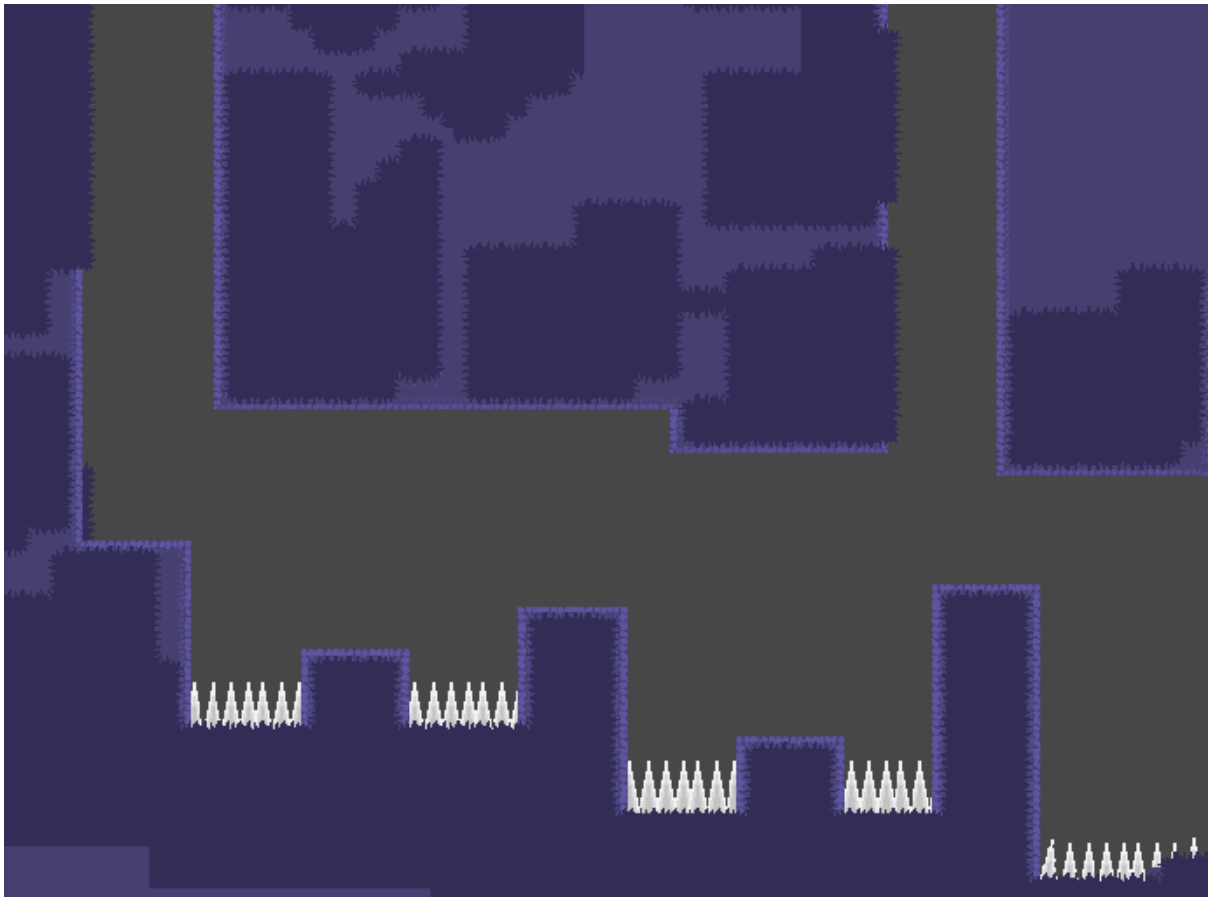


Figure 155 The Crystal Mines

5.26.4 The Crystal Mines 2 Boss room

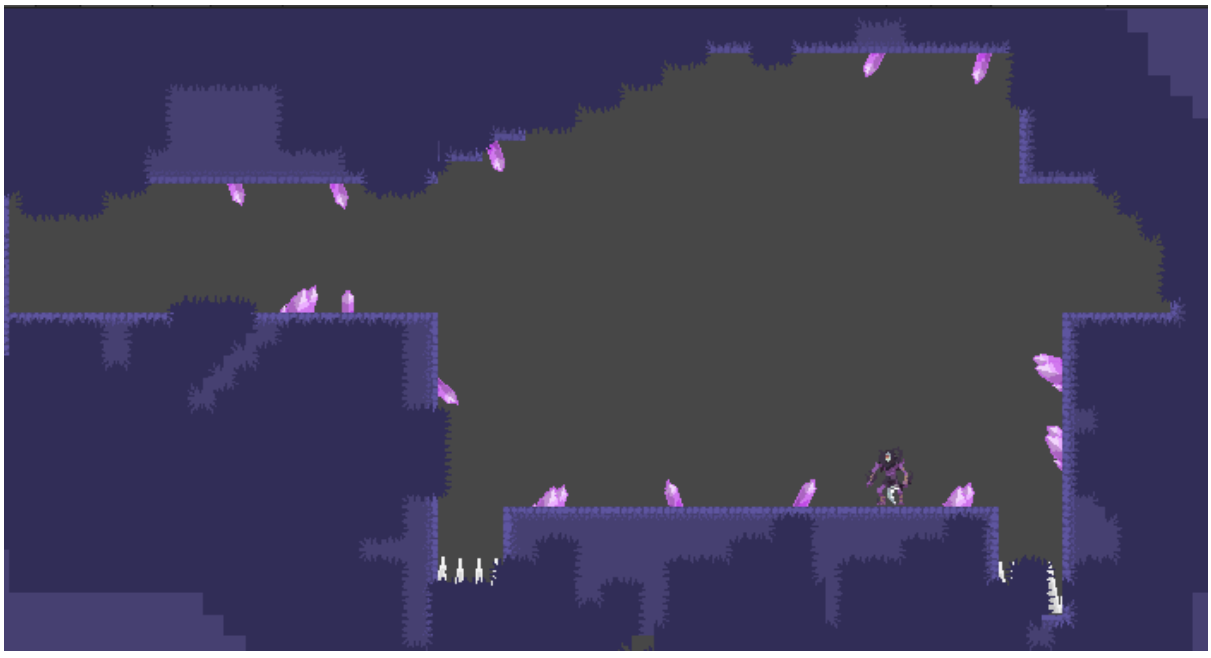


Figure 156 The Crystal Mines Boss

5.27 Sprite sheet

The player's sprite sheet has been downloaded from the Unity assets store. This came with different animations that weren't set up properly. This has been fixed by re-selecting the animation frames that combine.

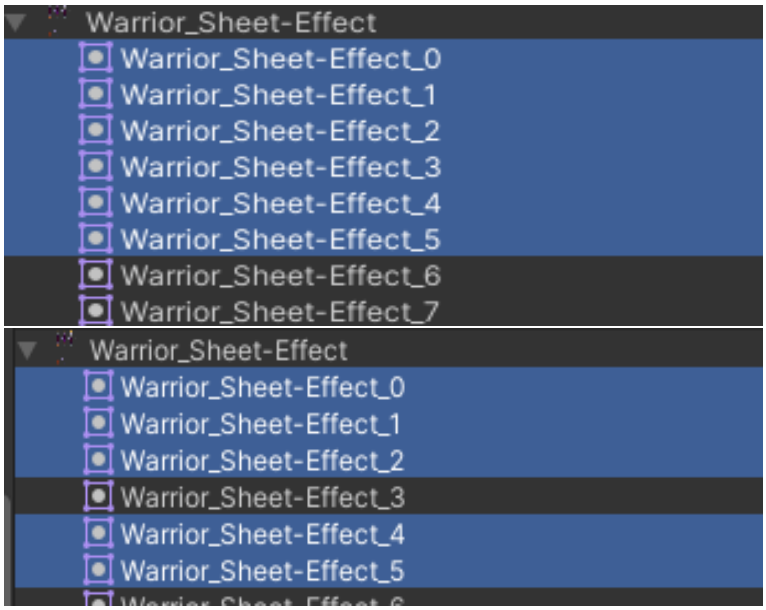


Figure 157 Player Sprite Sheet

The Sprit sheet for the Boss enemy is below. This enemy did not get implemented due to the time constraints of the project. However, he can be added in the future.

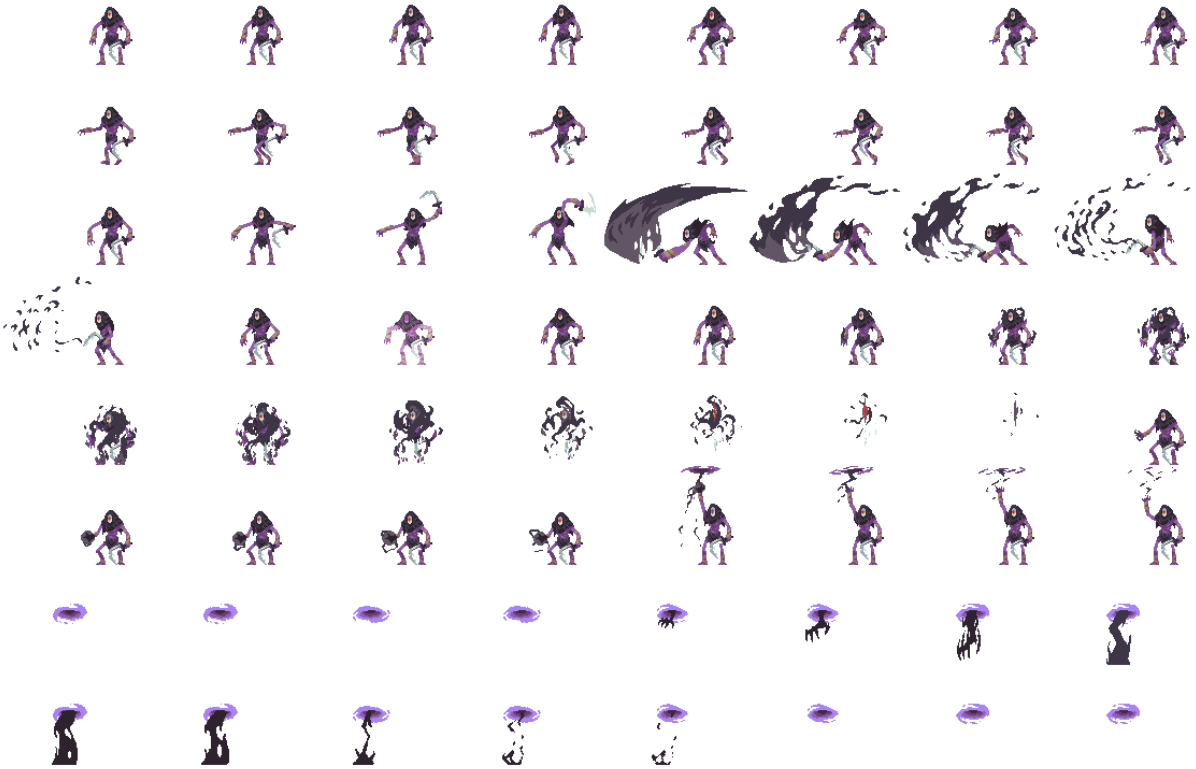


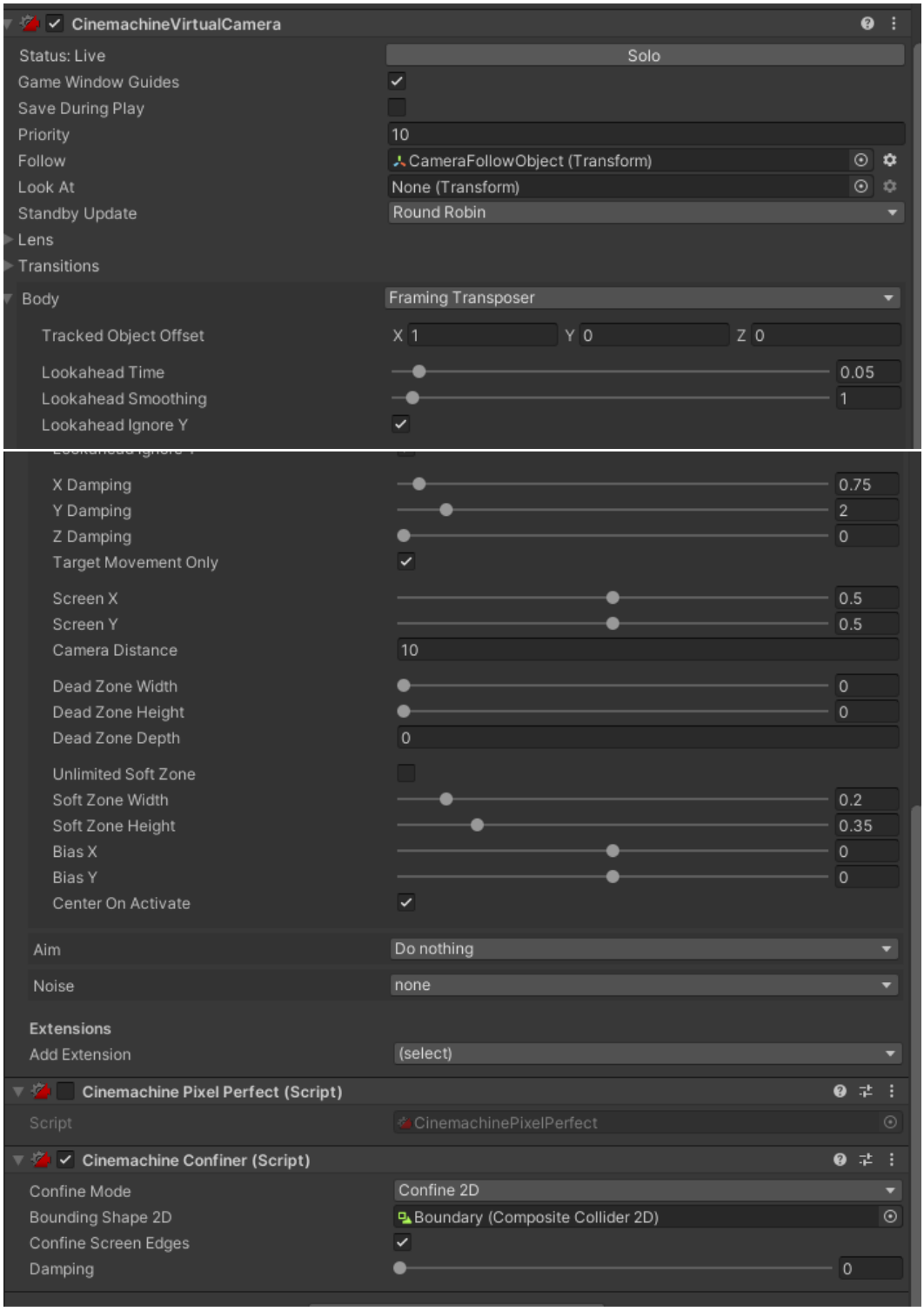
Figure 158 Sprite Sheet for the Crystal Mines Boss

5.28 Camera

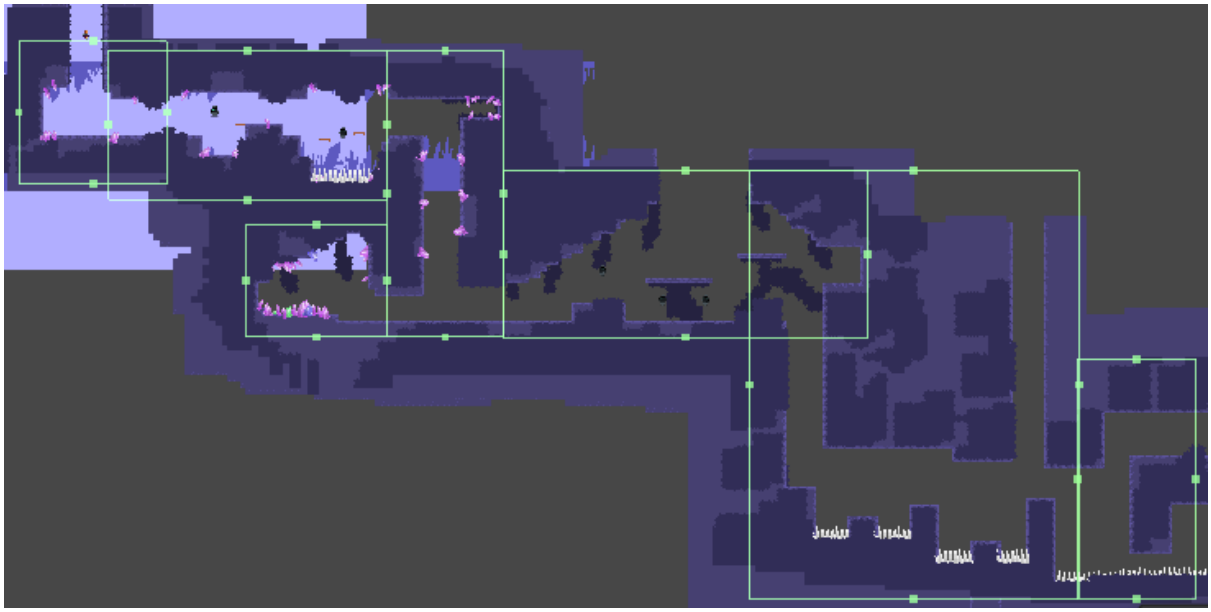
The camera plays a huge part in Metroidvania games. The package the author uses to manoeuvre the camera is “Cinemachine”.

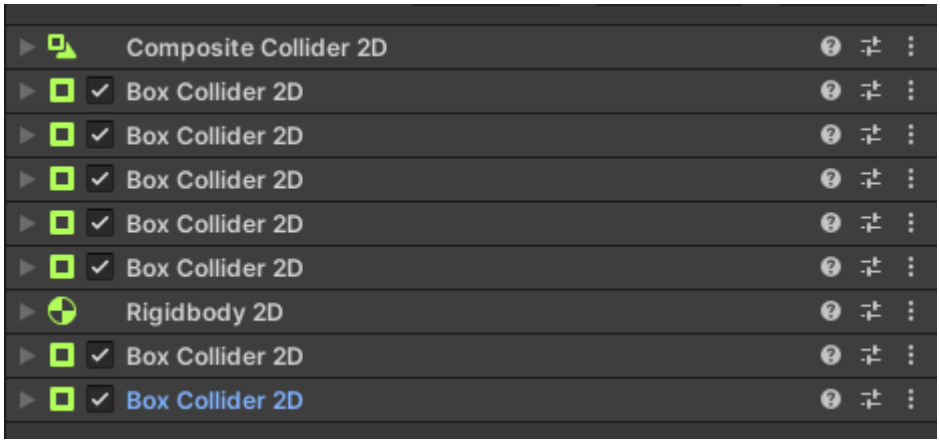
5.28.1 Cinemachine settings

This camera is used in open rooms. The author had planned to have multiple cameras with different settings, such as remaining static, showing only a specific area or not moving on a certain axis. However, the time constraints of this project did not allow for this.



The camera boundary is comprised of multiple 2D box colliders and a single 2D composite collider. The composite collider has its “IsTrigger” box ticked. A composite collider is used to combine every collider on the game object. The 2D box colliders must have the “Used By Composite” box ticked.





5.28.2 CameraFollowObject.cs

This script is attached to a game object. It uses:

A “SerializeField” transform, this object is assigned in the Unity inspector.

A reference to “Player” and a Boolean variable called “facingRight”.

In the “Awake()” function the player’s position is assigned to the player variable.

In the “Update()” function the position of the object set in the Unity inspector is moved to the position of the player.

```
public class CameraFollowObject : MonoBehaviour
{

    [SerializeField] private Transform playerTransform;


    private Player player;

    private bool facingRight;

    0 references
    private void Awake()
    {
        player = playerTransform.gameObject.GetComponent<Player>();

        facingRight = player.facingRight;
    }

    // Start is called before the first frame update
    0 references
    void Start()
    {

    }

    // Update is called once per frame
    0 references
    void Update()
    {
        transform.position = playerTransform.position;
    }
```

Figure 159 CameraFollowObject.cs

5.29 Parallax

The author uses Parallax to give depth to the 2D game they are creating. **Figure 109** below is an example of how to give the illusion of depth to the game. In the diagram below, the layer labled “A” is the player’s layer. In the Parallax.cs script the background behind the player layer will move as the player moves depending on how far away from the player layer they are.

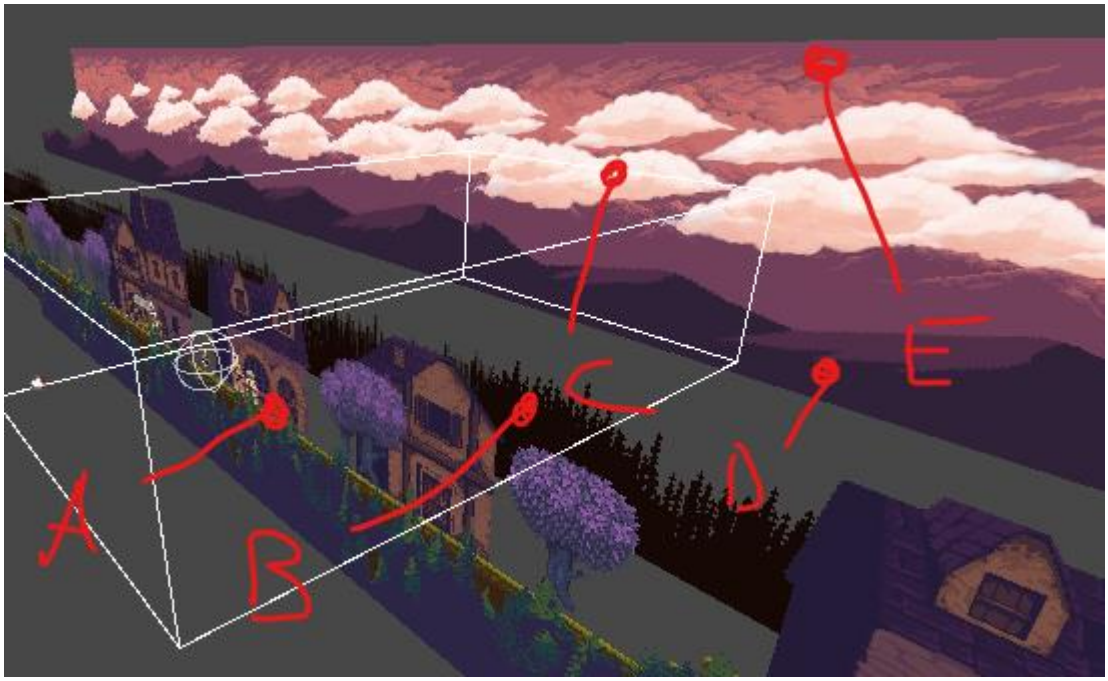
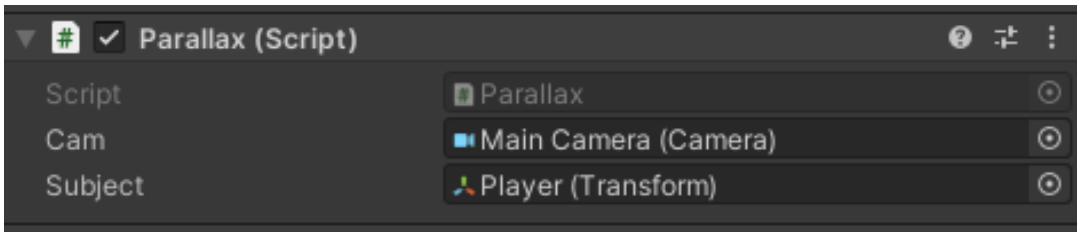


Figure 160 Parallax Example

5.29.1 Parallax.cs

This script uses the “Main Camera” object, and the player’s transform to determine how the background scene will move.



The “Start()” function assigns the startPosition and startZ variables.

```
public void Start()
{
    startPosition = transform.position;
    startZ = transform.position.z;

}
```

The “Vector2” variable “travel” is calculated each time the variable needs to be used. It holds the distance the camera has travelled from the “startPosition”.

The float “distanceFromSubject” is calculated each time it is used. This distance is along the Z axis because the further behind the “subject” the background is, then the less it should move.

The float “clippingPlane” is calculated each time it is used. If the “distanceFromSubject” variable is greater than 0 (it is behind the subject) add the z position of the camera to the “cam.farClipPlane”. Otherwise add the z position of the camera to “cam.nearClipPlane”

To get the parallax factor, the float “parallaxFactor” is used and calculates what it is by dividing the “distanceFromSubject” by the “clippingPlane”.

```
public Camera cam;
public Transform subject; // subject is the player in this situation


Vector2 startPosition;
float startZ;

//property
//distance camera has moved from original position.
Vector2 travel => (Vector2)cam.transform.position - startPosition;

//the distance that the background or forground is from the player.
float distanceFromSubject => transform.position.z - subject.position.z;

// the cam position + either the far or near clipping plane. since objects that are on
the near clipping plane will move the oposite direction.
// if distance from subject is greater than 0 then use cam.farclippingplane else use
nearclippingplane
float clippingPlane => (cam.transform.position.z + (distanceFromSubject > 0? cam.
farClipPlane : cam.nearClipPlane));

// the paralax factor is the distance from the player divided by the clipping plane.
float parallaxFactor => Mathf.Abs(distanceFromSubject) / clippingPlane;
```

Figure 161 Parallax.cs

In the “FixedUpdate()” function, a new Vector 2 called “newPos” is used to calculate where the new position should be. Then it is moved using a “transform.position”.

```
// needs to be fixed update to not have the background stutter.
public void FixedUpdate()
{
    Vector2 newPos = startPosition + travel * parallaxFactor;
    transform.position = new Vector3(newPos.x, newPos.y, startZ);
}
```

Figure 162 Parallax.cs FixedUpdate() Function

5.30 Animation

In the “PlayerMvt.cs” script “Update()” function, - Updated to “Player.cs Update()” - when the player is on the ground the falling Boolean is set to false. This will stop the fall animation from continually playing after the player falls.

```
private void Update()
{
    //if the player is on the ground the falling anim will be false.
    if (IsGrounded())
    {
        myAnimator.SetBool("falling", false);
        jumpsLeft = maxJumps;

        coyoteTimeCounter = coyoteTime;

    }
    else
    {
        coyoteTimeCounter -= Time.deltaTime;
    }

    if(rb.velocity.y < 0f)
    {
        myAnimator.SetBool("falling", true);
        myAnimator.ResetTrigger("jump");
        //this is added to make sure the falling animation isnt stuck if the player holds the button while moving.
        if(IsGrounded())
        {
            myAnimator.SetBool("falling", false);
        }
    }
```

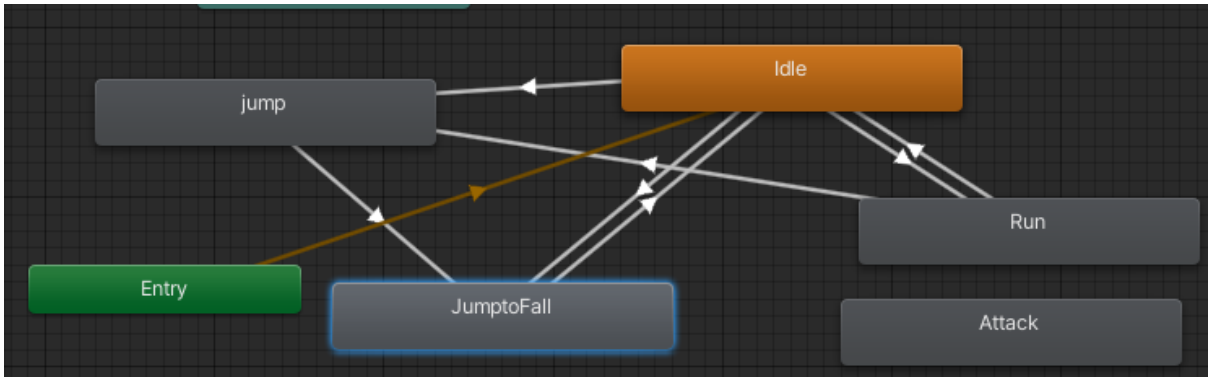
Figure 163 Old Script, does not exist anymore.

Once the player’s velocity is less than 0, the fall animation is played, and the jump trigger is reset so the player can jump again. The nested if statement checking if “IsGrounded()” is true to stop the falling animation if the player is still holding one of the movement buttons.

In “PlayerMvt.cs” in the “Jump()” function - Updated to “Player.cs Jump()” - when the player presses the “Space” button, the trigger variable jump is toggled and the Boolean variable “falling” is set to false. This will play the “Jump” animation. When the “Space” key is released the “Jump” trigger is reset and the “falling” variable is set to true.

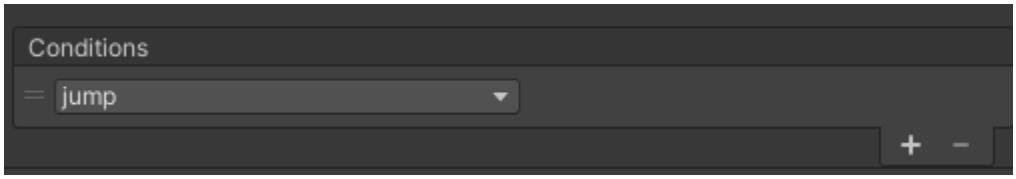
```
public void Jump(InputAction.CallbackContext context)
{
    //when jump is pressed and jumps left is more than 0.
    if (context.performed && jumpsLeft > 0f)
    {
        //adds a velocitiy (jump force) to the y value of the rigid body.
        rb.velocity = new Vector2(rb.velocity.x, jumpForce);
        myAnimator.SetTrigger("jump");
        myAnimator.SetBool("falling", false);

        //the coyote timer makes it so there is some leaniency with jumping
        //taking away 1 jump from jumps left.
        if(coyoteTimeCounter <= 0f)
        {
            jumpsLeft -= 1;
        }

    }
    //when jump is canceled.
    else if (context.canceled && rb.velocity.y > 0f)
    {
        //allowing the player to jump higher by pressing jump for longer and
        rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y * 0.5f);
        myAnimator.ResetTrigger("jump");
        myAnimator.SetBool("falling", true);

        coyoteTimeCounter = 0f;
    }


}
```

The “Idle”, “Jump”, “JumptoFall” and “Run” animations all work.

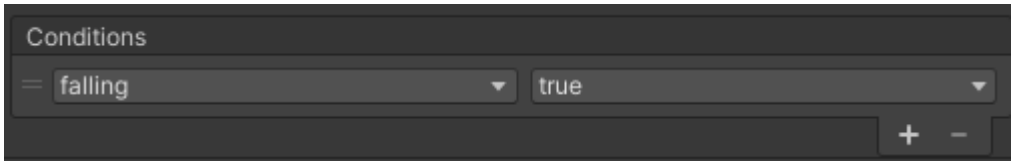


When the user doesn't press any buttons and isn't falling, the idle animation will play. If the user presses the "A" or "D" key, the run animation will play. If the user presses the "jump" key the jump animation will play.

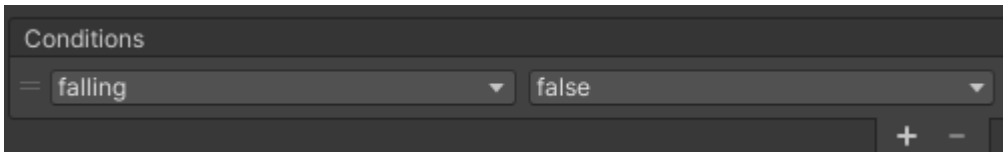
From "Idle" to "Jump", a trigger condition is needed. This checks when the parameter "jump" is toggled.



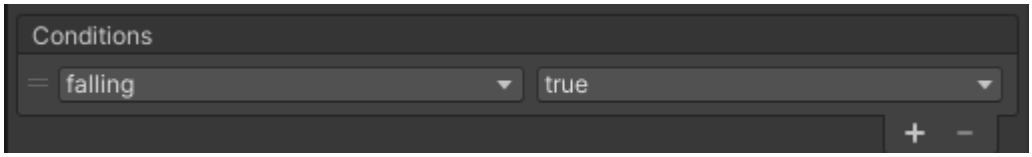
Once the user reaches the peak of their jump, stops, or has a negative y velocity, the "JumpToFall" animation will play, and the variable "falling" is set to true.



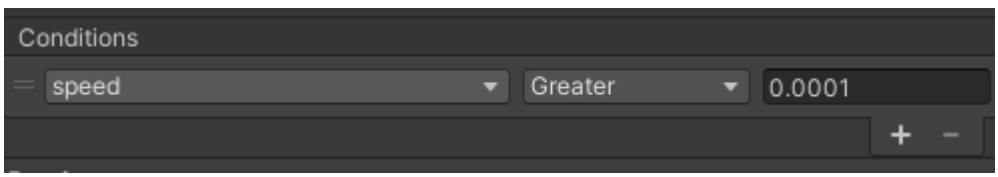
Once the Character lands on the ground "falling" is set to false and will no longer play the "JumptoFall" animation.



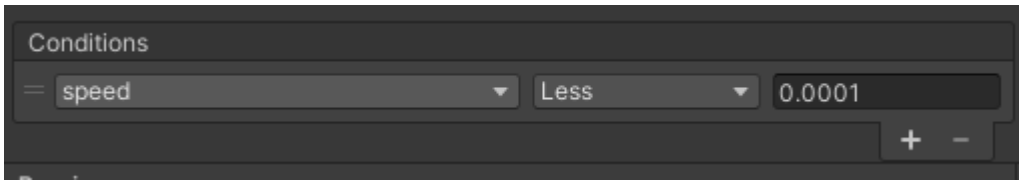
If the user falls off a platform their y velocity is negative the "JumptoFall" animation will play.



If the user presses the "A" or "D" key to move, the "Run" animation will play. When checking if speed is greater than 0.0001 is making the response quicker so the character will react faster to the input of the user.



If the user stops pressing the “A” or “D” keys the “Run” animation will stop playing. When checking if speed is less than 0.0001 the player will stop faster.



The graph below has been updated so that the “JumpToFall” will go back to “Idle” at the correct time. The player can now go from “Run” to “JumpToFall”. This means the player isn’t running in mid-air or falling while moving on the ground.

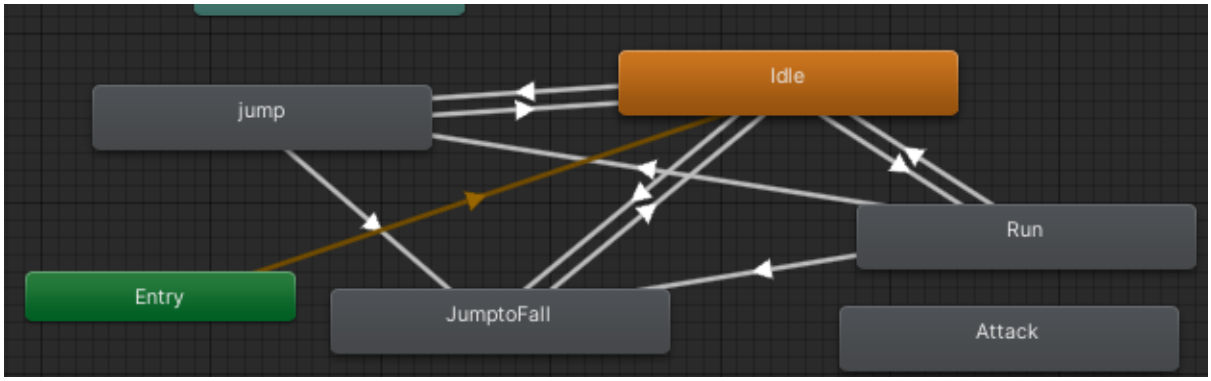


Figure 164 Unity Animator

Updated Player animator below.

The “PlayerAnimations()” Function shows how this works.

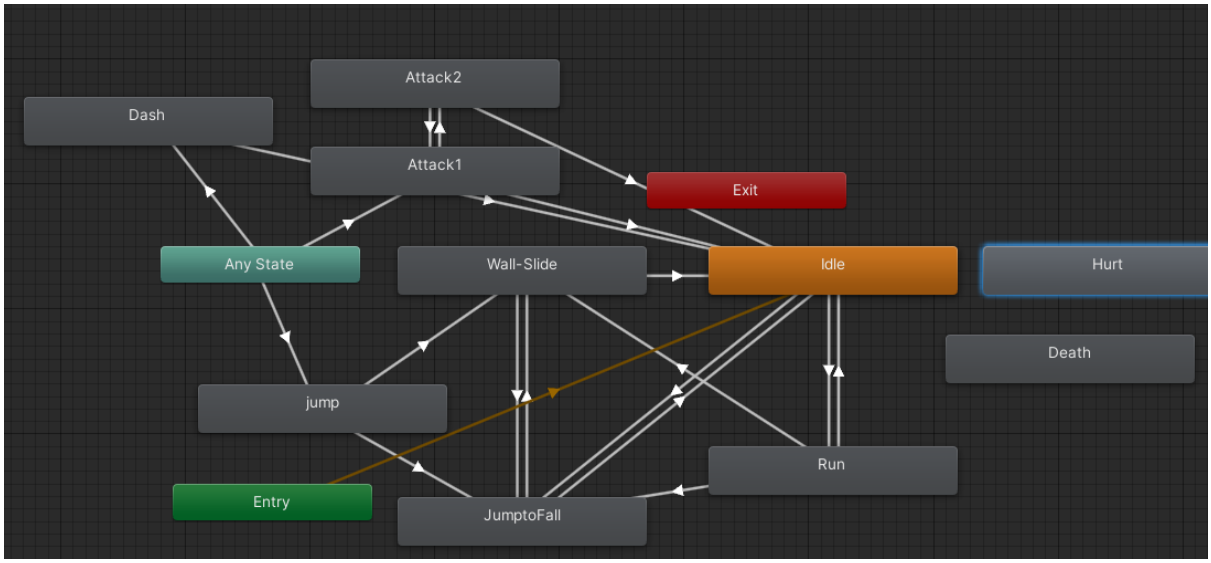


Figure 165 Unity Animator

6 Testing Chapter

This Chapter will discuss why it is necessary to test, the tests that were carried out, and the feedback from them.

6.1 Usability Testing

The usability testing in this project was done on 4 different people with varying degrees of experience with gaming.

6.2 The Objective of Testing

Testing helps the creator of the game discover what they had not thought to include. There are too many things to keep in mind when creating a game and there will always be human error. The reason testing is done is to make sure the majority of bugs are figured out before release. The creator of the game can easily tweak some values depending on how the feedback goes. For example, if an enemy is too hard to kill, the health points of the enemy can be reduced.

6.3 The Tasks

The participants were given two tasks to complete while they achieved the main objective of the levels:

- Explore the world.
- Try to get out of bounds.

6.4 The Results of Testing

The First Test.

This participant has a lot of experience with gaming. They suggested that the attack range be increased as it was hard to attack the enemies without being in the enemy attack range. This was fixed immediately. They also suggested that there was a knockback on the player attack so that it would make it easier to fight them. There was not enough time to implement this. There is a knock back when the player or enemy collide. They liked the movement of the character. They did not know there was a dash ability. This can be added as an NPC that explains how to use the dash ability. However, there was not enough time to add this. This participant managed to get out of bounds. This was fixed by making a kill box under the areas that the player can fall from.

The Second Test

This participant had some gaming experience. They did not know how to attack or dash. This could be explained by one of the NPCs in the starting area. However, there was no time to implement this. They struggled with fighting the enemies but only died twice. They did not manage to get out of bounds.

The Third Test

This participant had some gaming experience. They figured out which button was the attack button by pressing most of the buttons on the keyboard and in the process figured out they can dash as well. This participant realised you can sometimes dash through walls if you are at a specific distance. This bug was known and can not be fixed within the time constraints of the project.

The Fourth Test

This participant had little to no experience gaming. They first used the arrow keys (Up, Down, Left, Right) to move around which was different to all the other participants who used the WASD keys. They didn't know how to interact with the NPCs and needed instruction. They didn't know how to attack either and were given some assistance.

6.6 Feature Testing

This section covers the tests that were performed on different features to ensure they work correctly.

Main Menu testing.

Test	Expected Output	Actual Output	Status
Main Menu – Play Game button	Starts level 1 Click Sound is played.	Starts level 1. Click sounds is played. No background sound	Working Fixed
Main Menu – Settings Button	Show Settings Menu Click Sound is played	Show Settings Menu Click Sound is played	Working
Main menu – Settings Menu - Video Button	Show Video Settings Menu	Show Video Settings Menu	Working
Main Menu – Video Menu - Resolution Dropdown	Shows resolutions in dropdown. Click Sound is played when selecting option	Shows some resolution values dropdown. when selecting option	Not Working Correctly. Working in Game Pause Menu though
Main Menu – Video Menu - Quality Dropdown	Shows different Qualities in dropdown. Change the quality. Click Sound when selected.	Shows different Qualities in dropdown. Change the quality. Click Sound when selected.	Working
Main Menu – Settings menu - Audio Button	Shows Audio Menu Click Sound	Shows Audio Menu Click sound.	Working
Main Menu – Audio Menu – Master Volume Slider	Moving slider changes the volume of all sounds.	Moving slider changes the volume of all sounds.	Working
Main Menu – Audio Menu – SFX Slider	Moving slider changes the volume of SFX sounds.	Moving slider changes the volume of all sounds.	Working
Main Menu – Audio Menu – Background Music Slider	Moving slider changes the volume of Background Music.	Moving slider changes the volume of Background Music.	Working
Audio Menu – Back Button	Shows Settings Menu Click Sound	Shows Settings Menu Click Sound	Working
Video Menu – Back Button	Shows Settings Menu Click Sound	Shows Settings Menu Click Sound	Working
Settings Menu – Back Button	Shows Main Menu Click Sound	Shows Main Menu Click Sound	Working
Main Menu – Quit Button	Quits Application	Quits Application	Working

Player Input keyboard testing.

Test	Expected Output	Actual Output	Status
Player Jump	The Player jumps, Sound Effect is played. The Animation is played	The Player jumps, Sound Effect is played. The Animation is played	Working
Player WASD Or Arrow Keys Movement	A and D move the player Left and Right. The Run Animation is played. The Run Sound is played	A and D move the player Left and Right. The Run Animation is played. The Run Sound is played. Holding W or S will play the run sound. Also holding W or S while using A or D will half your normal running speed.	Working
Player Interact E	If the player can interact. The dialogue will show.	If the player can interact. The dialogue will show.	Working If the player presses interact too fast, it can make the typing effect type over itself.
Player Attack J	The player will hit enemies in the attack range. The attack animation is played.	The player will hit enemies in the attack range. The attack animation is played.	Working
Player Dash K	The player will dash. The dash animation will play.	The player will dash. The dash animation will play.	Working Can go through walls if the player is standing a certain distance away.
Pause Button ESC	The game will be paused	The game will be paused	Working

In Game Pause Menu Testing

Test	Expected Output	Actual Output	Status
Pause Button ESC	Resumes the game	Resumes the game	Working Does not reset the Pause menu. Eg if the settings menu is open, and the ESC button is pressed it will open the Settings menu when the ESC button is pressed again.
Resume Game Button	Resumes the game	Resumes the game	Working
Settings Button	Shows Settings Menu	Shows Settings Menu	Working
Settings Menu – Video Button	Shows Video Menu	Shows Video Menu	Working
Settings Menu – Audio Button	Shows Audio Menu	Shows Audio Menu	Working
Settings Menu – Back Button	Shows Pause Menu	Shows Pause Menu	Working
Audio Menu – Master Volume Slider	Changes the master volume	Changes the master volume	Working
Audio Menu – Background Music Slider	Changes the Background music volume	Changes the Background music volume	Working
Audio Menu – SFX Slider	Changes the SFX volume	Changes the SFX volume	Working
Audio Menu – Back Button	Shows Settings Menu	Shows Settings Menu	Working
Video Menu – Back Button	Shows Settings Menu	Shows Settings Menu	Working
Video Menu – Quality Dropdown	Changes the quality	Changes the quality	Working
Video Menu – Resolution Dropdown	Shows all available resolutions. Changes the resolution	Shows all available resolutions. Changes the resolution	Working
Quit Button	Goes back to the Main Menu	Goes back to the Main Menu	Working When pressing play the first scene will be paused. To fix it just pause and un-pause.

Player Death Menu

Test	Expected Output	Actual Output	Status
The player dies	Show the Death Menu	Show the Death Menu	Working
Death Menu – Restart Button	Restarts scene	Restarts scene	Working Scene 2 (the crystal Village 2) Will restart without background music.
Death Menu – Menu Button	Go to Main Menu	Go to Main Menu	Working

7 Project Management

7.1 Introduction

This chapter describes how the project was managed. It will discuss the separate phases of the project, the challenges encountered, and the solutions found.

7.2 Phases

7.2.1 Proposal

The author decided to create a 2D Metroidvania game with basic movement controls, health bars, sword attacks, story progression, item pickups, enemies, 3 different areas to explore (possibly 4), a boss fight for each area (this might be too much for the time frame), autosave features and save files.

7.2.2 Research

It was necessary to learn about Inheritance this was done by extensive research using Youtube videos and academic articles.

It was also necessary to learn about the design process when creating a game. This was mainly done by reading academic articles. This took longer than expected but it helped the author's understanding of the challenge that had been undertaken.

7.2.3 Requirements

Looking at Ori and the Will of the Wisps and Hollow Knight the author extracted the elements that they favoured most and decided to emulate them. Similarly, the aspects of those games they disliked most were excluded.

7.2.4 Design

It was necessary to use wireframes to create a basic understanding of how the game would look. Ori and the Will of the Wisps and Hollow Knight gave plenty of inspiration for how this game was designed.

7.2.5 Implementation

Coding and artwork took by far the longest amount of time spent creating the project. When issues arose that the author would spend hours trying to resolve, they found the best solution was to sleep on the problem and the answer became very clear in the following days. At one point the author was trying to fix one issue and was doing research on it but the answer to an unrelated problem became clear. Learning to be open to this possibility was a good learning experience. The author found being responsible for hitting specific goals throughout the project was helpful.

7.3 SCRUM Methodology

The SCRUM Methodology created more panic than assistance to the author's way of working. This took some time to figure out.

7.4 Project Management Tools

7.4.1 Trello

The author used Trello at the beginning of the project and found that it was helpful to keep track of their goal initially.

7.4.2 Journal/Notes

The author found making to do lists the most effective way to keep track of their goals.

7.4.3 GitHub

GitHub was used to store an easily accessible copy of the project online. This could have been used to retrieve a previous version of the project if an irreversible change was made. This would also allow other people to collaborate if this was a group project.

7.5 Reflection

The author was diagnosed with ADHD early in the year and it therefore made complete sense why the SCRUM Methodology was creating such panic. The author had their own version of sprints, however, but was unsure of when they would happen. The author was medicated with Ritalin, and this made a huge difference with focus. The understanding and assistance of their project supervisor was greatly appreciated and helped diminish the stress.

7.5.1 Deferral

The decision to defer in June gave the author time to make the best use of their ideas. It did mean that the tutors were not present for the duration of the implementation chapter. However, the author works well on their own. This will have to be taken into account when going into full time employment. ADHD may be seen as a negative but working in an area the author loves can produce extraordinary results.

7.5.2 Your views on the project

The author has created something that has the potential to be great. The author is much more confident about working in the industry with this year's experience and can see how much more could have been added to the game.

7.5.3 Working with a supervisor

The supervisor was very helpful when the author hit patches of demotivation. She was calm and reassuring and kept lines of communication open. The author is grateful for this.

7.5.4 Conclusion

A 6-month project creating a game from scratch is a huge amount of work, and motivational issues were to be expected. The author is very pleased to have overcome them and reached the finish line.

8 Business Opportunities

This game still needs many updates to be added to before it can properly be monetized but is in a very good position to achieve this.

8.1 Paid Download

To monetize the application the author can upload the game to Steam or a similar website that sells game downloads.

Itch.io is a website for indie game developers to upload their creations. These can be free or paid downloads.

9 Conclusion

The aim of this game was to create a 2D immersive environment while to also have complex code and interesting storyline.

Overall, this was achieved even though some of the content had to be cut from the finished product because of time constraints. The author found the problem-solving aspect of the project stimulating, and the coding interesting. In the future the author hopes to continue adding to this game. For example, the author would add a save system to save progress throughout the game, multiple boss fights, more areas to explore, and a knockback effect on the player's attack.

The author learned many skills including improving C# skills, wireframing, project management, time management, setting achievable goals, sleeping on a problem and taking rest days.

10 References

10.1 Research

Baumeister, R., Bratslavsky, E., Finkenauer, C., & Vohs, K. (2001). *Bad is Stronger than Good - Roy F. Baumeister, Ellen Bratslavsky, Catrin Finkenauer, Kathleen D. Vohs, 2001*. SAGE Journals. Retrieved 21 December 2021, from https://journals.sagepub.com/doi/abs/10.1037/1089-2680.5.4.323.

Canziba, E. (2018). *Hands-On UX Design for Developers*. Ebookcentral.proquest.com. Retrieved 21 December 2021, from https://ebookcentral.proquest.com/lib/iadt-ebooks/reader.action?docID=5485019&query=.

Isbister, K. (2006). *Better Game Characters by Design A Psychological Approach*. Gamifique.files.wordpress.com. Retrieved 23 December 2021, from https://gamifique.files.wordpress.com/2011/11/9-better-game-characters-by-design-a-psychilogical-approach.pdf.

Nakamura, J., & Csikszentmihalyi, M. (2009). *Oxford Handbook of Positive Psychology*. Google Books. Retrieved 22 December 2021, from https://books.google.ie/books?hl=en&lr=&id=6IyqCNBD6oIC&oi=fnd&pg=PA195&dq=flow+theory+csikszentmihalyi&ots=INFcVKY8pz&sig=1S-Gkx8oYfsF3VSVGvcHBlMdvFw&redir_esc=y#v=onepage&q=flow%20theory%20csikszentmihalyi&f=false.

Nichols, K., & Chesnut, D. (2014). *UX for Dummies*. Ebookcentral.proquest.com. Retrieved 21 December 2021, from https://ebookcentral.proquest.com/lib/iadt-ebooks/reader.action?docID=1674223&query=.

Rosenzweig, E. (2015). Successful user experience : Strategies and roadmaps. Elsevier Science & Technology. Retrieved 21 December 2021, from https://ebookcentral.proquest.com/lib/iadt-ebooks/reader.action?docID=2122438.

Sparks, E., & Baumeister, R. (2008). *Positive Psychology: Exploring the Best in People*. Images-insite.sgp1.digitaloceanspaces.com. Retrieved 21 December 2021, from https://images-insite.sgp1.digitaloceanspaces.com/dunia_buku/koleksi-buku-neuro-science/positive-psychology-exploring-the-best-in-people-4-volumes-set-pdfdrivecom-66441577416922.pdf#page=68.

Hejlsberg, A., Golde, P., Wiltamuth, S., & Torgersen, M. (n.d.). The C# programming language. https://books.google.ie/books?hl=en&lr=&id=ICe7ea4RscUC&oi=fnd&pg=PT14&dq=C%23%2Binheritance&ots=YcRLDZKem0&sig=LIQx-VfNXZFmZ5Odb0rb7A_XXko&redir_esc=y#v=onepage&q=C%23%20inheritance&f=false

10.2 Requirements

CHAN, C. (2022). Get lost in the best Metroidvania games for the Switch. Retrieved 11 June 2022, from https://www.imore.com/best-metroidvania-games-nintendo-switch

Crego, V. (2020). What abilities and skills to get first | Ori and the Will of the Wisps Guide. Retrieved 10 June 2022, from https://squadstate.com/guide/what-abilities-and-skills-to-get-first-ori-and-the-will-of-the-wisps-guide

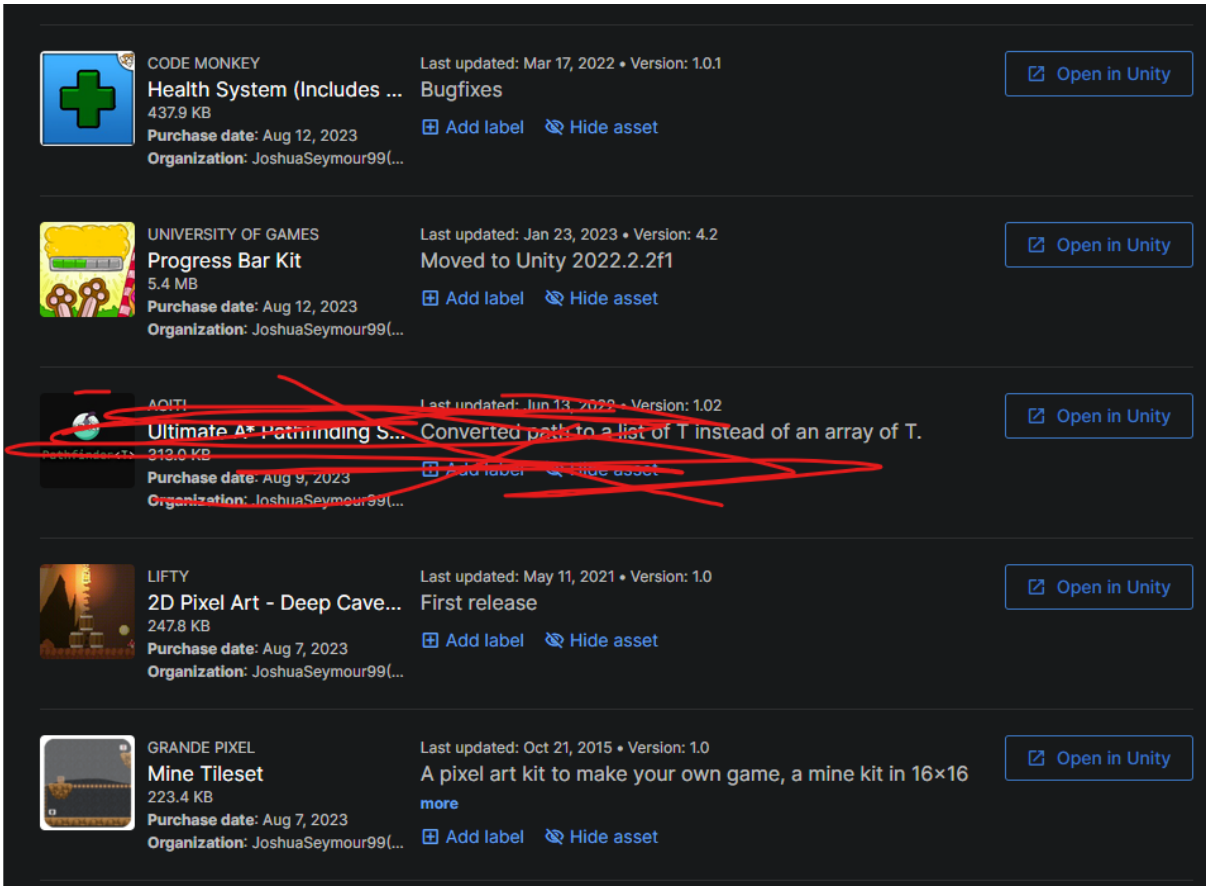
Ori and the Will of the Wisps PC & Console 2020. (2022). Retrieved 10 June 2022, from https://www.gameuidatabase.com/uploads/Ori-and-the-Will-of-the-Wisps06302020-082722-71864.jpg

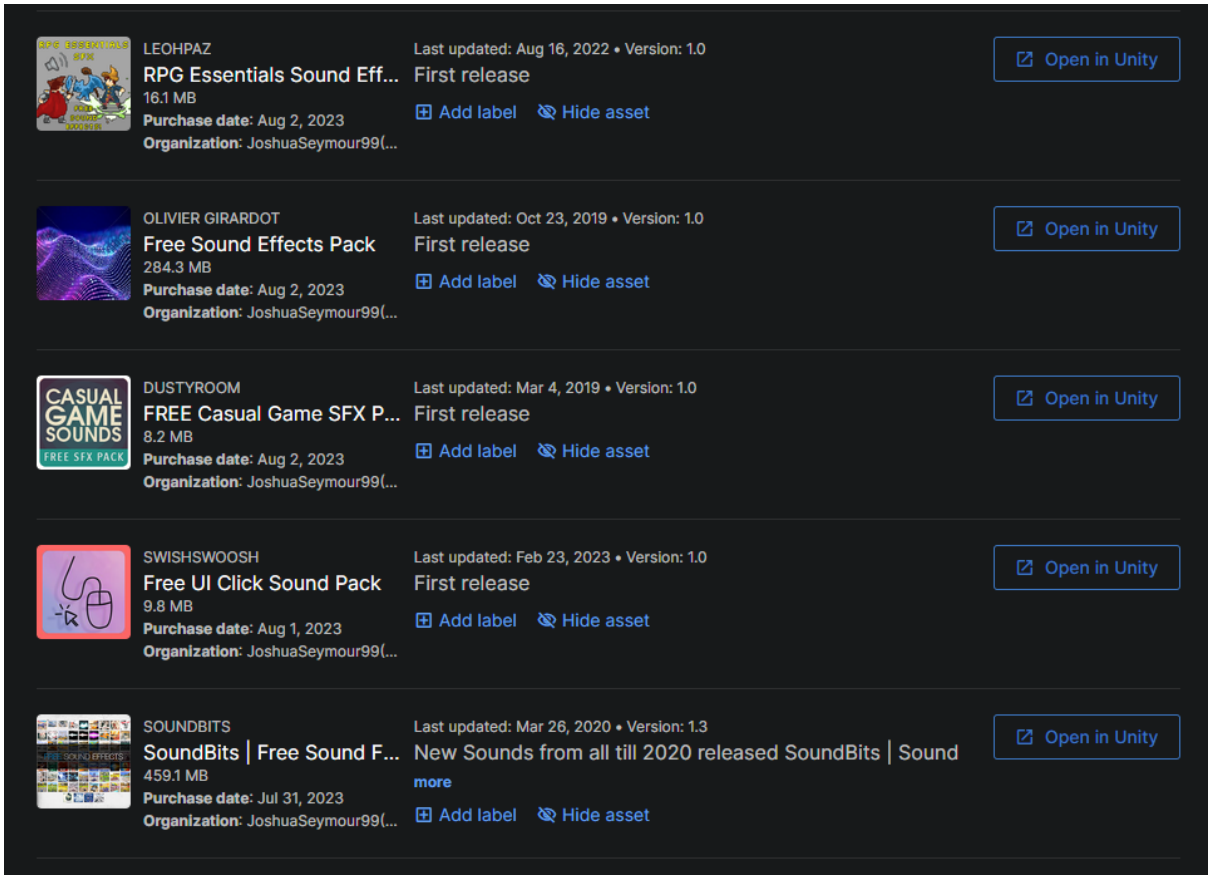
Irwin, D. (2022). Ori And The Will Of The Wisps guide: 20 tips for beginners. Retrieved 11 June 2022, from https://www.rockpapershotgun.com/ori-and-the-will-of-the-wisps-guide-20-tips-for-beginners

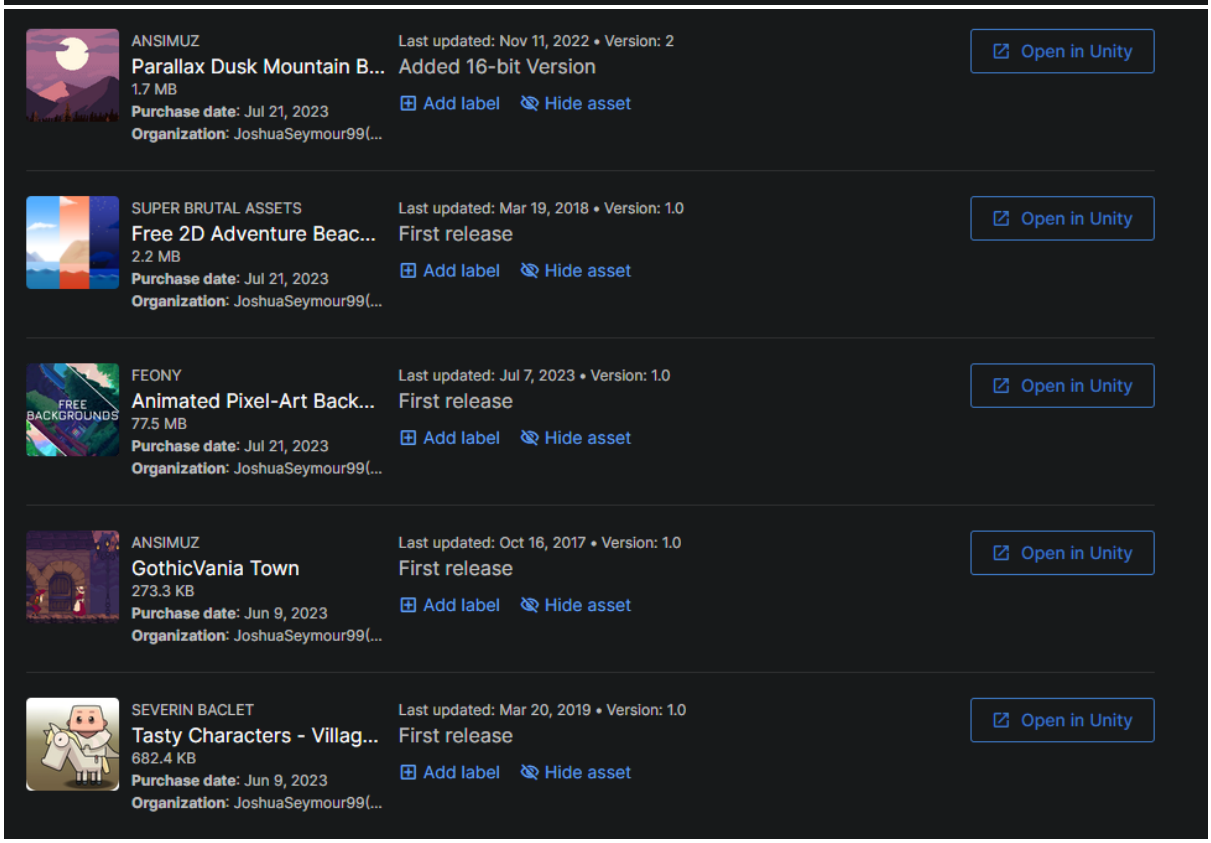
Locke, J. (2022). Retrieved 10 June 2022, from https://mspoweruser.com/hollow-knight-is-coming-to-xbox-one-with-a-physical-edition-next-year/

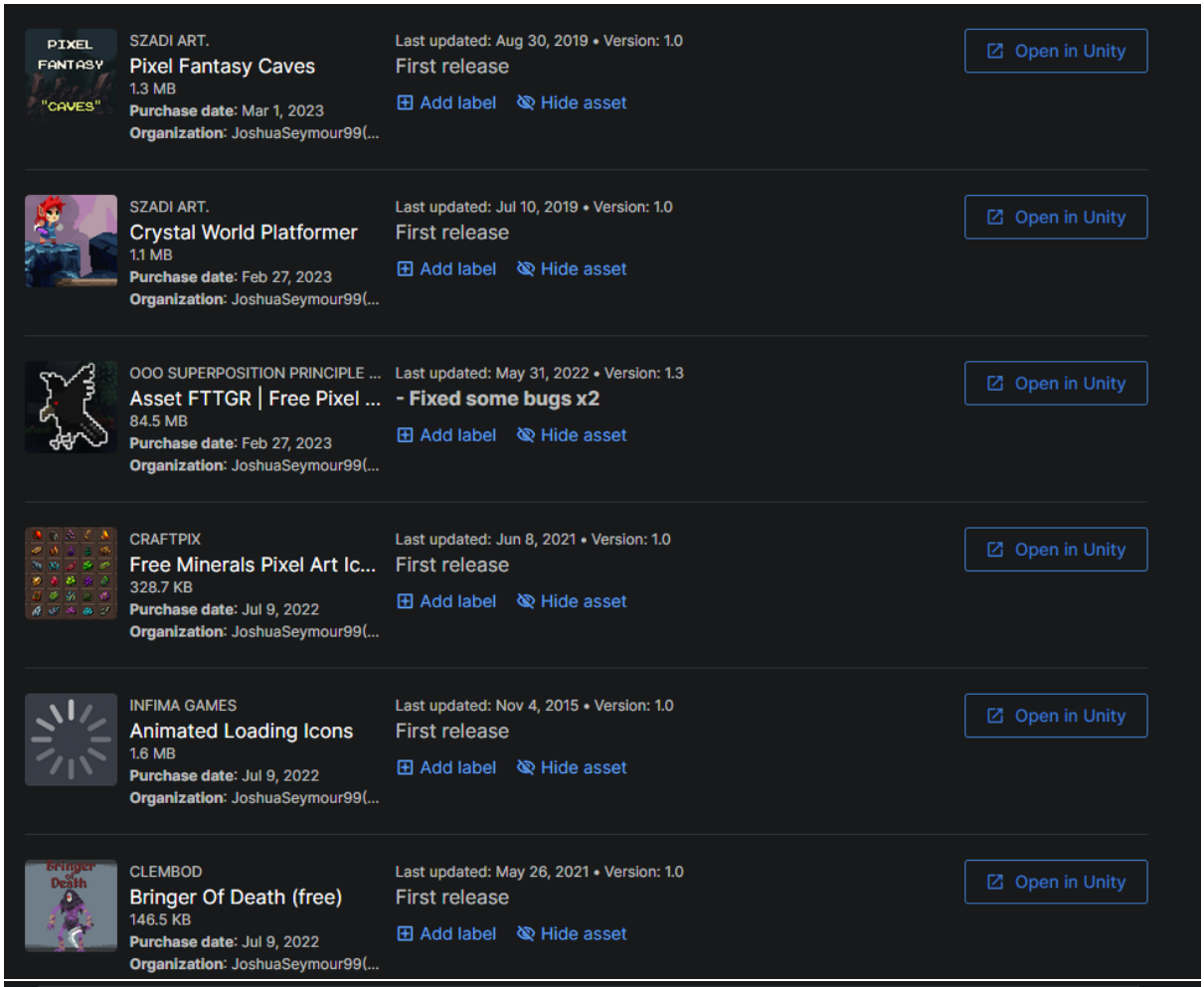
10.3 Implementation Assets

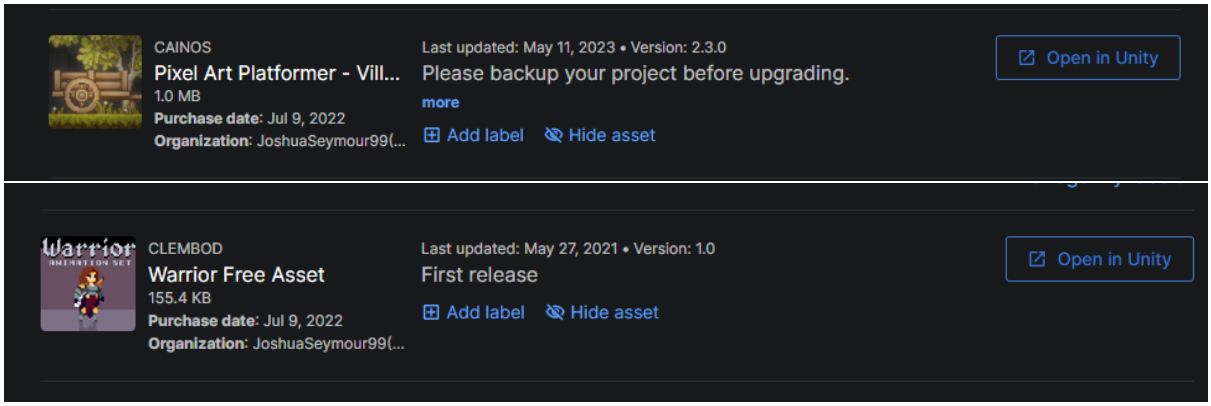
These are all from the Unity store for free




LEOHPAZ
RPG Essentials Sound Eff...
16.1 MB
Purchase date: Aug 2, 2023
Organization: JoshuaSeymour99(...
Last updated: Aug 16, 2022 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
OLIVIER GIRARDOT
Free Sound Effects Pack
284.3 MB
Purchase date: Aug 2, 2023
Organization: JoshuaSeymour99(...
Last updated: Oct 23, 2019 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
DUSTYROOM
FREE Casual Game SFX P...
8.2 MB
Purchase date: Aug 2, 2023
Organization: JoshuaSeymour99(...
Last updated: Mar 4, 2019 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
SWISHSWOOSH
Free UI Click Sound Pack
9.8 MB
Purchase date: Aug 1, 2023
Organization: JoshuaSeymour99(...
Last updated: Feb 23, 2023 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
SOUNDBITS
SoundBits | Free Sound F...
459.1 MB
Purchase date: Jul 31, 2023
Organization: JoshuaSeymour99(...
Last updated: Mar 26, 2020 • Version: 1.3
New Sounds from all till 2020 released SoundBits | Sound
more
Add label
Hide asset
Open in Unity


ANSIMUZ
Parallax Dusk Mountain B...
1.7 MB
Purchase date: Jul 21, 2023
Organization: JoshuaSeymour99(...
Last updated: Nov 11, 2022 • Version: 2
Added 16-bit Version
Add label
Hide asset
Open in Unity
SUPER BRUTAL ASSETS
Free 2D Adventure Beac...
2.2 MB
Purchase date: Jul 21, 2023
Organization: JoshuaSeymour99(...
Last updated: Mar 19, 2018 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
FEONY
Animated Pixel-Art Back...
77.5 MB
Purchase date: Jul 21, 2023
Organization: JoshuaSeymour99(...
Last updated: Jul 7, 2023 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
ANSIMUZ
GothicVania Town
273.3 KB
Purchase date: Jun 9, 2023
Organization: JoshuaSeymour99(...
Last updated: Oct 16, 2017 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
SEVERIN BACLET
Tasty Characters - Villag...
682.4 KB
Purchase date: Jun 9, 2023
Organization: JoshuaSeymour99(...
Last updated: Mar 20, 2019 • Version: 1.0
First release
Add label
Hide asset
Open in Unity


SZADI ART.
Pixel Fantasy Caves
1.3 MB
Purchase date: Mar 1, 2023
Organization: JoshuaSeymour99(...
Last updated: Aug 30, 2019 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
SZADI ART.
Crystal World Platformer
1.1 MB
Purchase date: Feb 27, 2023
Organization: JoshuaSeymour99(...
Last updated: Jul 10, 2019 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
OOO SUPERPOSITION PRINCIPLE ...
Asset FTTGR | Free Pixel ...
84.5 MB
Purchase date: Feb 27, 2023
Organization: JoshuaSeymour99(...
Last updated: May 31, 2022 • Version: 1.3
- Fixed some bugs x2
Add label
Hide asset
Open in Unity
CRAFTPIX
Free Minerals Pixel Art Ic...
328.7 KB
Purchase date: Jul 9, 2022
Organization: JoshuaSeymour99(...
Last updated: Jun 8, 2021 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
INFIMA GAMES
Animated Loading Icons
1.6 MB
Purchase date: Jul 9, 2022
Organization: JoshuaSeymour99(...
Last updated: Nov 4, 2015 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
CLEMBOD
Bringer Of Death (free)
146.5 KB
Purchase date: Jul 9, 2022
Organization: JoshuaSeymour99(...
Last updated: May 26, 2021 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
ANSIMUZ
Warped Caves
235.6 KB
Purchase date: Jul 9, 2022
Organization: JoshuaSeymour99(...
Last updated: Nov 8, 2017 • Version: 1.0
First release
Add label
Hide asset
Open in Unity
IPHIGENIA PIXELS
2D Platfrom Tile Set - Cave
3.4 MB
Purchase date: Jul 9, 2022
Organization: JoshuaSeymour99(...
Last updated: Jan 9, 2017 • Version: 1.2
Added a new 4th folder, content is the same, only the
more
Add label
Hide asset
Open in Unity
BLACK HAMMER
Fantasy Wooden GUI : Fr...
14.9 MB
Purchase date: Jul 9, 2022
Organization: JoshuaSeymour99(...
FREE GUI
Last updated: Feb 18, 2019 • Version: 2.1
First release
Add label
Hide asset
Open in Unity


CAINOS
Pixel Art Platformer - Vill...
1.0 MB
Purchase date: Jul 9, 2022
Organization: JoshuaSeymour99(...
Last updated: May 11, 2023 • Version: 2.3.0
Please backup your project before upgrading.
more
Add label
Hide asset
Open in Unity
Warrior
ANIMATION SET
CLEMBOD
Warrior Free Asset
155.4 KB
Purchase date: Jul 9, 2022
Organization: JoshuaSeymour99(...
Last updated: May 27, 2021 • Version: 1.0
First release
Add label
Hide asset
Open in Unity