



# Dungeon Scribbles: A Study of Procedural Generation and Pathfinding in 2D Game Design

Samuel Downey

N00212512

Project Supervisor

Timm Jeschawitz

Year 4 2024-25

## DL836 BSc (Hons) in Creative Computing

### **Declaration of Authorship**

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 Programme Chair.

**WARNING:** Take care when discarding program listings lest they be copied by some- one else, which may well bring you under suspicion. Do not to 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.

### **Declaration**

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Signed: \_\_\_\_\_

Date: \_\_\_\_\_

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

## **Abstract**

This thesis explores the full development cycle of a top-down 2D roguelike video game built using the Unity Engine. The project was guided by a strong focus on research-led design, incorporating both front-end and back-end planning to ensure the final product was stable, functional, and visually engaging. Early research played a critical role in shaping the project's technical foundations, particularly in the implementation of procedural generation and pathfinding, two key systems that created a dynamic and replayable game environment.

The development followed a modular approach, allowing systems such as player movement, combat, health, animations and much more to be implemented in isolation and tested independently. A structured project management process, using Miro boards and management strategies such as agile sprint tracking and kanban methodology, helped maintain consistent progress and supported clear documentation throughout development. Manual testing and player feedback informed several iterations of improvement, enhancing the game's playability and overall experience.

The result is a playable prototype that meets its original goals while leaving room for future content expansion. Key technical competencies in Unity, C#, system architecture, and game design were strengthened, and the project serves as a strong foundation for continued development portfolio presentation. This thesis provides a detailed overview of how thoughtful planning, research, and iterative testing can support the creation of a well-rounded independent game project.

### **Acknowledgements**

I would like to acknowledge and give thanks to my project supervisor Timm who encouraged me through this project and offered me genuine and thoughtful critiques to make sure this project was going in a good direction.

I would like to give a special thanks to my girlfriend and my family who have been cheering me on over the duration of this project. Their support means the world to me.

## Table of Contents

Dungeon Scribbles: A Study of Procedural Generation and Pathfinding in 2D Game Design.....	1
1 Introduction .....	12
1.1 Overall Aim .....	12
1.2 Application Area .....	12
1.3 Technologies .....	12
1.3.1 Project Management Technologies .....	12
1.3.2 Development Technologies .....	12
1.3.3 Design Technologies.....	12
1.4 Project Management.....	12
1.5 Requirements .....	13
1.5.1 Functional Requirements.....	13
1.5.2 Non-Functional Requirements .....	13
1.6 Design .....	13
1.6.1 Back-End Design.....	13
1.6.2 Front-End Design .....	13
1.7 Implementation.....	13
1.8 User Testing .....	14
2 Research.....	15
2.1 Front-End Research .....	15
2.1.1 The Binding of Isaac.....	15
2.1.2 Vagante .....	18
2.1.3 Hades .....	20
2.1.4 Hollow Knight.....	22
2.2 Back-End Research .....	22
2.2.1 Development Engine Research.....	22
2.2.2 Procedural Content Generation Research .....	24
2.2.3 Pathfinding Algorithm Research.....	26
3 Requirements.....	29
3.1 Introduction .....	29
3.2 Requirements Gathering .....	29
3.3 Requirements Modelling .....	29
3.3.1 Functional Requirements.....	29
3.3.2 Non-Functional Requirements .....	31
3.4 Feasibility .....	33
3.5 Conclusion .....	33

4 Design .....	33
4.1 Introduction .....	33
4.2 Programme Design .....	34
4.2.1 Unity Structure.....	34
4.2.2 Design Pattern .....	37
4.3 User Interface Design .....	37
4.3.1 Wireframes .....	37
4.3.2 User Flow Diagram .....	39
4.3.4 Level Design .....	42
4.4 Conclusion .....	42
5 Implementation .....	42
5.1 Introduction .....	42
5.2 Sprint 1 .....	42
5.2.1 Goals .....	42
5.2.2 Goal 1 – Functional Research .....	43
5.2.3 Goal 2 – Back-End System Research .....	43
5.2.4 Goal 3 – Gathering Applied Research .....	43
5.3 Sprint 2 .....	44
5.3.1 Goals .....	44
5.3.2 Goal 1 – Front-End Design Breakdown.....	44
5.3.3 Goal 2 – Wireframe Creation .....	44
5.4 Sprint 3 .....	44
5.4.1 Goals .....	44
5.4.2 Goal 1 – Creating a Development Area .....	45
5.4.4 Goal 2 – Camera Transition System .....	48
5.4.5 Goal 3 – Enemy Pathfinding System .....	51
5.4.6 Goal 4 – Player Combat System .....	54
5.4.7 Goal 5 – Health and Knockback system.....	57
5.5 Sprint 4 .....	61
5.5.1 Goals .....	61
5.5.2 Goal 1 – Room Prefab .....	63
5.5.3 Goal 2 – Procedural Map Generation .....	65
5.5.4 Goal 3 – Nav Mesh Integration .....	72
5.5.5 Goal 4 – Updated Camera Transition .....	74
5.5.6 Goal 5 – Handle Aim Update.....	75
5.5.7 Goal 6 – Enemy integration .....	76

5.5.9 Goal 7 – Pause Menu .....	79
5.6 Sprint 5 .....	80
5.6.1 Goals .....	80
5.6.2 Goal 1 – Loading Screen.....	81
5.6.3 Goal 2 – Finalise Room .....	83
5.6.5 Goal 3 – Health bar UI addition .....	89
5.6.6 Goal 4 – Health Item .....	91
5.6.7 Goal 5 – Enemy design implementation.....	93
5.6.8 Goal 6 – Fixing enemy in starting room bug .....	99
5.7 Sprint 6 .....	103
5.7.1 Goals .....	103
5.7.2 Goal 1 – Add Door Design to Room .....	104
5.7.3 Goal 2 – Add a Player Design .....	105
5.7.4 Goal 3 – Update Combat System .....	111
5.7.5 Goal 4 – Fix Player Clipping Bug .....	114
5.7.6 Goal 5 – Update User Interface .....	116
5.7.8 Goal 6 – Updating Pause Menu .....	117
5.7.9 Goal 7 – Main Menu Implementation .....	122
5.7.10 Goal 8 – Audio system Implementation .....	128
5.7.11 Goal 9 – Controller Support .....	139
5.8 Sprint 7 .....	141
5.8.1 Goals .....	141
5.8.2 Goal 1 – First Draft of Report.....	141
5.8.3 Goal 2 – Receive Feedback.....	142
6 Testing .....	143
6.1 Introduction .....	143
6.2 Functional Testing .....	143
6.2.1 Menu Navigation .....	143
6.2.2 Player Controls .....	144
6.2.3 Enemy Interaction .....	145
6.2.4 Map Navigation .....	145
6.3 User Testing .....	146
6.3.1 Player Controls .....	146
6.3.2 Enemy Interaction .....	148
6.3.3 Game World .....	149
6.3.4 Menu Interaction .....	150

6.4 Conclusion .....	151
6.4.1 Player Controls .....	151
6.4.2 Enemy Interaction .....	151
6.4.3 Game World .....	151
6.4.4 Menu Interaction .....	151
7 Conclusion .....	152
7.1 Introduction .....	152
7.2 Technologies .....	152
7.2.1 Figma .....	152
7.2.2 Photoshop .....	152
7.2.3 Unity .....	152
7.2.3 Visual Studio Code .....	153
7.2.4 Mirro .....	153
7.2.5 Unity Version Control.....	153
7.2.6 OneDrive .....	153
7.3 Project Phases .....	154
7.3.1 Research .....	154
7.3.2 Requirements .....	154
7.3.3 Design.....	155
7.3.4 Implementation .....	155
7.3.5 User Testing .....	155
7.6 Reflection .....	156
7.6.1 Project Management .....	156
7.6.2 Views on The Project .....	157
7.6.3 Working with a Supervisor .....	157
7.6.4 Further Competencies & Skills .....	157
7.6.5 How the Project Could be Developed Further .....	157
7.7 Conclusion .....	158
9 References.....	159



## Table of Figures

Figure 1 - Screenshot of The Binding of Issac Main Menu .....	15
Figure 2 - Screenshot of The Binding of Issac Gameplay / User Interface .....	16
Figure 3 - Screenshot of The Binding of Isaac Pause Menu .....	17
Figure 4 - Vagante Main Menu .....	18
Figure 5 - Screenshot of Vagante Gameplay / User Interface .....	19
Figure 6 - Screenshot of Hades Main Menu .....	20
Figure 7 - Screenshot of Hades Gameplay / User Interface .....	20
Figure 8 - Screenshot of Hades Pause Menu .....	21
Figure 9 - Hollow Knight Pause Menu .....	22
Figure 10 - Screenshot of Unity File Explorer .....	34
Figure 11 - Screenshot of Animation File Structure .....	34
Figure 12 - Screenshot of Audio File Structure .....	35
Figure 13 - Screenshot of Prefabs File Structure .....	35
Figure 14 - Screenshot of Scripts File Structure .....	36
Figure 15 - Screenshot of TileSets File Structure .....	36
Figure 16 - Screenshot of Main Menu Wireframe .....	37
Figure 17 - Screenshot of User Interface Wireframe .....	38
Figure 18 - Screenshot of Pause Menu Wireframe .....	38
Figure 19 - Main Menu User Flow Chart .....	39
Figure 20 - Player Controls User Flow Chart .....	40
Figure 21 - Pause Menu User Flow Chart .....	41
Figure 22 - User Interface User Flow Chart .....	41
Figure 23 - Snippet of back-end system research .....	43
Figure 24 - Screenshot of Test Level .....	45
Figure 25 - Code snippet of Player Movement script .....	46
Figure 26 - Code snippet of Input Manager for Player Movement .....	47
Figure 27 - Snippet of trigger points for Camera Transition script .....	48
Figure 28 - Code snippet of camera transition script .....	49
Figure 29 - Code snippet of camera transition script .....	50
Figure 30 - Snippet of test enemy game object .....	51
Figure 31 - Code Snippet of enemy movement AI .....	52
Figure 32 - Snippet of nav surface script for baking object detection .....	53
Figure 33 - Script to set collision layer as an object to detect .....	53
Figure 34 - Updated Player Movement script for Player Melee integration .....	54
Figure 35 - Code snippet of Player Melee Script .....	55
Figure 36 - Code snippet of Player Weapon script .....	56
Figure 37 - Code snippet of Player Health Script .....	57
Figure 38 - Updated Enemy script with health integration. ....	58
Figure 39 - Code snippet of updated player movement script for new aim mechanic and knockback integration. ....	59
Figure 40 - Updated Player movement script for knockback integration. ....	60
Figure 41 - Updated Enemy script for knockback integration. ....	61
Figure 42 - Code Snippet of Room script .....	63
Figure 43 - Screenshot of room prefab. ....	64
Figure 44 - Code snippet one of Room generation script .....	65
Figure 45 - Code snippet of Room generation script. ....	66
Figure 46 - Code snippet of Room generation script. ....	66
Figure 47 - Code snippet of Room generation script. ....	67
Figure 48 - Code snippet of Room generation script. ....	67
Figure 49 - Code snippet of Room generation script. ....	68

Figure 50 - Code snippet of Room generation script. ....	69
Figure 51 - Code snippet of Room generation script. ....	70
Figure 52 - Snippet of Room design iteration. ....	70
Figure 53 - Room generation script test. ....	71
Figure 54 - Snippet of Enemy nav mesh bug ....	72
Figure 55 - Update to enemy movement script to fix bug.....	72
Figure 56 - Demonstration of bug fix in action.....	73
Figure 57 - Updated camera transition logic. ....	74
Figure 58 - Update to player movement script. ....	75
Figure 59 - Snippet of enemy integration into procedural generation. ....	76
Figure 60 - Code snippet of enemy integration.....	76
Figure 61 - Screenshot of enemy spawning bug.....	77
Figure 62 - Enemy spawning bug solution.....	78
Figure 63 - Code Snippet of interface manager script. ....	79
Figure 64 - Snippet of pause menu integration.....	80
Figure 65 - Loading screen design.....	81
Figure 66 - Code snippet of loading screen logic.....	82
Figure 67 - Finalised room design.....	83
Figure 68 - Code snippet of door logic. ....	84
Figure 69 - Code snippet of door logic. ....	85
Figure 70 - Snippet of room script for smart door logic.....	86
Figure 71 - Snippet of room script for smart door logic.....	87
Figure 72 - Snippet of room script for smart door logic.....	87
Figure 73 - Snippet of room script for smart door logic.....	88
Figure 74 - First version of health bar added to user interface.....	89
Figure 75 - First version of dynamically moving health bar.....	89
Figure 76 - Additions to player health script for health bar functionality.....	90
Figure 77 - Code snippet of health item.....	91
Figure 78 - Health item in the game with new enemy designs implemented.....	92
Figure 79 - Additions to room manager script to implement health items.....	92
Figure 80 - Enemy walking sprite sheet.....	93
Figure 81 - Enemy animation controller.....	94
Figure 82 - Animation controller variables added to enemy movement.....	95
Figure 83 - Animation controller variables added to enemy damage.....	96
Figure 84 - Enemy designs implemented into game.....	97
Figure 85 - Addition of death animations to enemy animation controller.....	98
Figure 86 - Animation variables being implemented into enemy death logic.....	99
Figure 87 - Modifications to the room manager script.....	100
Figure 88 - Modifications to the room manager script.....	100
Figure 89 - Modifications to the room manager script.....	101
Figure 90 - Modifications to the room manager script.....	102
Figure 91 - Modifications to the room manager script.....	103
Figure 92 - Updated room design with door prefabs brought in.....	104
Figure 93 - New character design being implemented.....	105
Figure 94 - Player animation controller.....	106
Figure 95 - Player movement code updated with animator integrations.....	107
Figure 96 - Player movement code updated with animator integrations.....	108
Figure 97 - Player movement code updated with animator integrations.....	109
Figure 98 - Player movement code updated with animator integrations.....	110
Figure 99 - Updated handle aim function for easier animation controlling.....	111
Figure 100 - Updated player melee script with animation controller integration.....	112
Figure 101 - Updated player melee script with animation controller integration.....	113
Figure 102 - Updated player melee script with animation controller integration.....	114

<i>Figure 103 - Updated enemy script to fix clipping bug.</i>	115
<i>Figure 104 - Mini map added to user interface.</i>	116
<i>Figure 105 - Small code addition for room clearing icon.</i>	116
<i>Figure 106 - Updated health bar design.</i>	117
<i>Figure 107 - Updated pause menu design.</i>	118
<i>Figure 108 - Options menu addition.</i>	118
<i>Figure 109 - Updated Interface manager.</i>	119
<i>Figure 110 - Updated Interface manager.</i>	120
<i>Figure 111 - Updated Interface manager.</i>	121
<i>Figure 112 - Updated Interface manager.</i>	122
<i>Figure 113 - Main menu Added.</i>	123
<i>Figure 114 - Options menu Added.</i>	124
<i>Figure 115 - Main menu script.</i>	125
<i>Figure 116 - Main menu script.</i>	126
<i>Figure 117 - Main menu script.</i>	127
<i>Figure 118 - Game audio mixer.</i>	128
<i>Figure 119 - Volume settings script.</i>	129
<i>Figure 120 - Volume settings script.</i>	130
<i>Figure 121 - Audio manager script.</i>	131
<i>Figure 122 - Audio manager script.</i>	132
<i>Figure 123 - Additions to player movement script for player audio.</i>	132
<i>Figure 124 - Additions to player movement script for player audio.</i>	133
<i>Figure 125 - Additions to player movement script for player audio.</i>	134
<i>Figure 126 - Additions to player movement script for player audio.</i>	134
<i>Figure 127 - Additions to player melee script for player audio.</i>	135
<i>Figure 128 - Additions to player melee script for player audio.</i>	135
<i>Figure 129 - Additions to enemy script for enemy audio.</i>	135
<i>Figure 130 - Additions to enemy script for enemy audio.</i>	136
<i>Figure 131 - Additions to enemy script for enemy audio.</i>	137
<i>Figure 132 - Additions to enemy script for enemy audio.</i>	138
<i>Figure 133 - Additions to enemy script for enemy audio.</i>	138
<i>Figure 134 - New input system additions for controller support.</i>	139
<i>Figure 135 - Additions to player melee script for controller support.</i>	139
<i>Figure 136 - Additions to player melee script for controller support.</i>	140
<i>Figure 137 - Additions to interface manager script for controller support.</i>	140
<i>Figure 138 - Additions to interface manager script for controller support.</i>	141

# 1 Introduction

## 1.1 Overall Aim

The overall aim of this project is to successfully research and develop a 2D game with a similar gameplay loop to popular dungeon crawler roguelikes in the Unity 6 engine while furthering my knowledge into procedural generation systems and enemy artificial intelligence in video games.

## 1.2 Application Area

This project falls under the Entertainment and Media application area, with a specific focus on Game Development. Video games are a form of interactive media designed to entertain and engage players through immersive gameplay, storytelling, and interactive mechanics. The project explores the technical and creative aspects of game development, combining software engineering, digital art, sound design, and user experience design to create a cohesive and enjoyable game.

## 1.3 Technologies

### 1.3.1 Project Management Technologies

**Miro** will be used as the primary application for tracking the progress and development of this project. To maintain a clear record of game development updates, screenshots will be uploaded to a Miro board along with short breakdowns of these updates. This approach will provide a quick and accessible way to look back on previous versions of the game while also creating an easier method to collect these screenshots for the upcoming report.

The **Unity Version Control System** will be the primary version control system due to its ease of access within the Unity engine as well as its similarities with GitHub. The **OneDrive** cloud storage system will also be used for this project to store full project files after major implementations in case of a failure with the Unity Version Control System.

### 1.3.2 Development Technologies

The project will be developed using the **Unity 6 Engine** paired with **Visual Studio Code** as the IDE. This decision is explained in detail in Chapter 2.

### 1.3.3 Design Technologies

The project will be using **Figma** as the design application for the menu system wireframes as well as the user interface wireframes. **Photoshop CS6** will be used for asset design or image editing if necessary. The project will be using many assets found on the **Unity Asset Store** as well to save time on design while still having a high quality look.

## 1.4 Project Management

The project will use a **two-week Agile sprint** system, with each sprint focused on hitting specific implementation goals. Miro will be used to keep everything clear by showing tasks visually, along with regular sprint reviews with the project supervisor to make sure that the documentation stays up to date. To stay organized with tasks, a **Kanban** board will be set up with columns such as “To Do”, “In Progress”, and “Completed” to make it easier to track development progress.

## 1.5 Requirements

### 1.5.1 Functional Requirements

The functional requirements of this project will be guided by research into key game systems commonly found in games of similar genres. Researching how other games implement key features such as player systems, procedural room generation, and enemy systems will offer valuable insights into how these mechanics work and how they can be adapted for this project.

The goal is to understand how these systems operate and to use them as a reference point for structuring the codebase. Well-structured code is crucial for maintainability and efficient debugging, and studying proven examples will help ensure that each system is designed with these principles in mind.

### 1.5.2 Non-Functional Requirements

The non-functional requirements of this project will be informed by research into design principles of games within similar genres. Researching how other games handle menu design, player and enemy animations, and sound design will help shape the game's overall user experience and visual consistency.

Understanding how these design principles impact usability and player engagement will play a key role in guiding the design process. Factors such as menu layout, control scheme, visual feedback, and audio cues will be carefully considered to ensure the game feels intuitive, immersive, and enjoyable to play.

## 1.6 Design

### 1.6.1 Back-End Design

The back-end software design of this project will be focused on implementing each feature separately to ensure a well-organized and maintainable codebase. By keeping different systems, such as player movement, enemy AI, and room generation separated from each other, it will become easier to find and fix any bugs that are found. This approach will also improve the readability of the code, making it easier to understand and update in the future.

### 1.6.2 Front-End Design

The majority of the assets for the game will come from third-party platforms as well as the Unity Asset Store which offer a wide range of high quality sprites, animations and environment assets. This will make it easier to keep a consistent visual aesthetic across the game. Music and sound effects will be sourced from YouTube primarily as there are many royalty free soundtracks and sound effects available.

## 1.7 Implementation

The implementation phase will loosely adhere to the Agile sprint system which was explained previously in the chapter, primarily the purpose of keeping project progress and documentation organized. If goals are completed ahead of the sprint schedule, that will not be used as an excuse to not continue working. The implementation phase will be adhering heavily to the Kanban system that was also explained previously in the chapter. If utilized correctly, the Kanban system will provide a steady stream of implementation goals the project will need. With

these ideas in mind, there should be a steady stream of constant progress through the entire duration of the project, and the sprint goals can be rewritten to reflect the amount of work that was actually accomplished over that respective sprint period.

## 1.8 User Testing

The user testing phase will consist of three separate stages. The first stage involves manual testing of every change made to the game, ensuring that both modifications to existing features and newly implemented features function as intended. This process will focus on identifying and resolving errors while ensuring that no existing functionality is compromised. Testing will be thorough and extensive, especially for overlapping features, continuing until no bugs can be found.

The second phase involves user testing studies with friends and fellow students from Creative Computing and other departments. This phase aims to gather perspective and insight into how users interact with the game, including its menu systems, control scheme, and overall enjoyment. Feedback from this focus group will be used to implement changes, fix identified bugs and adjust features to ensure a fairer experience for the player before moving on to the final testing phase.

The third and final testing phase will involve large-scale user testing at Comic-Con Dublin, held at the Convention Centre in Dublin City Centre. A small group of students and lecturers has been invited to attend the event as ambassadors for the new Game Design course, providing an opportunity to gather extensive user testing data. This event will also offer the chance to receive feedback from major studios such as Black Shamrock and Larian Studios on ways to improve the game. With insights from industry professionals and public opinion, this phase is expected to be the most significant stage of testing.

## 2 Research

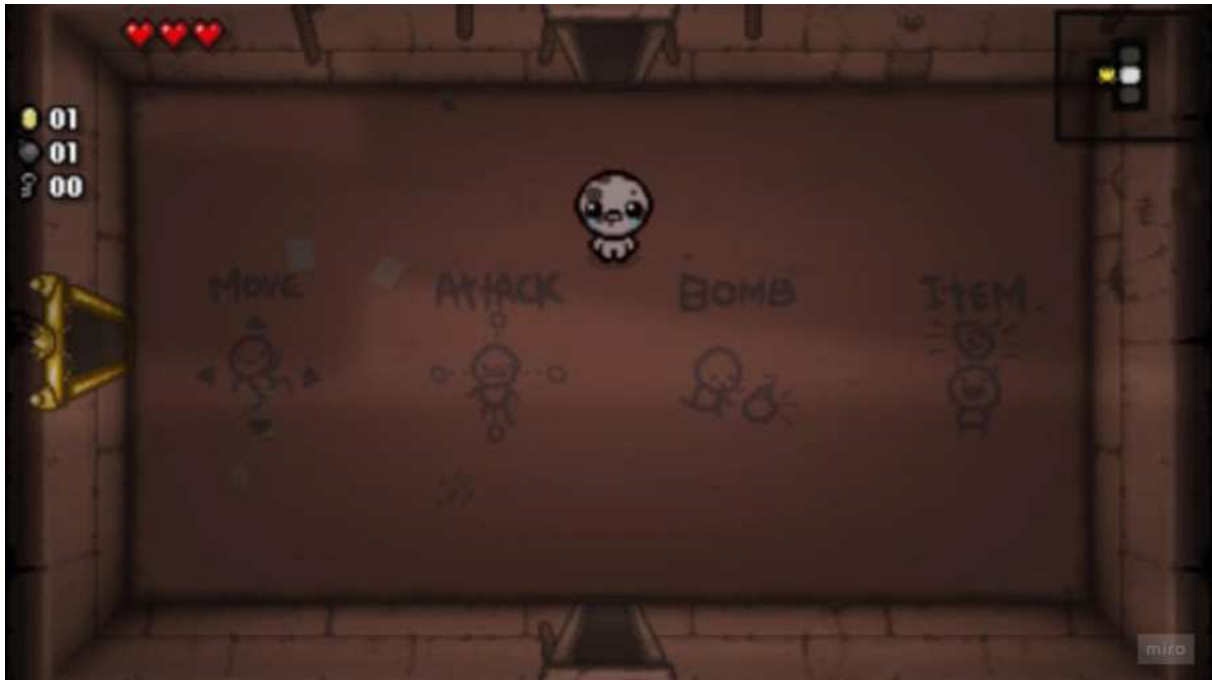
### 2.1 Front-End Research

#### 2.1.1 The Binding of Isaac



*Figure 1 - Screenshot of The Binding of Issac Main Menu.*

The Binding of Isaac's main menu (Seen in Figure 1) has a unique style, but it does not sacrifice its functionality. It has a basic list style menu, offering all the necessary buttons that a menu would need to provide such as the New Run, Continue and Options buttons. But it also has the Challenges and Stats buttons which are nice additions. The lack of a Quit button seems like a bit of an oversight in terms of overall user experience so I will be making sure to add one in my menu. Other than the lack of a Quit button, the menu does its job displaying the menu options to the player very well.



*Figure 2 - Screenshot of The Binding of Issac Gameplay / User Interface.*

When looking at The Binding of Issac's user interface (Seen in Figure 2), one of the first noticeable elements is the highly contrasting red hearts in the top-left corner of the screen, which, based on common gaming conventions, represents the player's current health. This is an effective way to communicate the health system without needing direct explanation, clearly implying that taking damage results in losing a heart.

Beneath the hearts is a simple inventory system, displaying coins, bombs, and keys. This layout clearly communicates the items that can be found but also removes any sense of mystery when discovering these items. While the system supports simplicity, it may not suit designs aiming for more surprise and exploration.

On the far right side of the screen is a mini-map, highlighting the player's current position while also displaying available paths and special rooms which are marked with a yellow crown to signify their importance. This is a simple and effective way to design a mini-map, and while the exact style may not be adopted, the focus on simplicity is a quality worth trying to emulate.



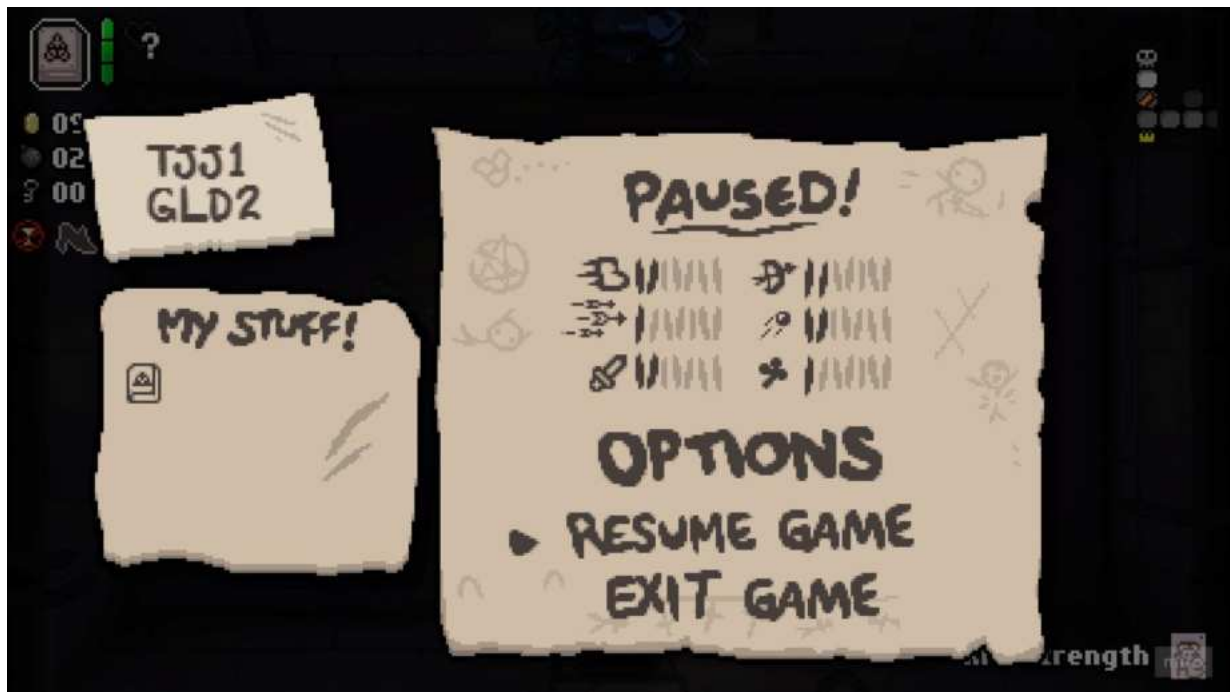


Figure 3 - Screenshot of The Binding of Isaac Pause Menu.

The Binding of Isaac's pause menu (Seen in Figure 3) shows a very simple and understandable design. The three main buttons, Resume, Options, and Exit are easy to see, helping users quickly find what they need. The options button stands out a little more due to the larger font, helping guide the user's attention to important buttons.

The player's stats are also shown in the pause menu, giving players a safe way to check their information without worrying about being attacked. The "My Stuff" section clearly lists the items collected by the user, keeping everything readable.

While the menu design works well and maintains visual consistency, the biggest takeaway is how it focuses on the most important buttons by simply making them a little bigger. Keeping the player's stats out of the pause menu might also be a better idea for the project in case the stats or upgrade system have to be scrapped.

### 2.1.2 Vagante



*Figure 4 - Vagante Main Menu.*

The Vagante main menu (Seen in Figure 4) is very simple and very easy to navigate, the artwork immediately familiarizes the player with the art style of the game. The marker over the selected menu button is great for showing what the player is currently selecting and removes any confusion about what they are doing. The lack of unnecessary buttons is great for keeping the menu system easy to understand, making it easier for the player to pick up and start playing.



*Figure 5 - Screenshot of Vagante Gameplay / User Interface.*

One of the first things to notice when breaking down Vagante's user interface (Seen in Figure 5) is the health bar which is positioned in the bottom left corner. It effectively shows the player's health through a number and bar format, making it easy for players to understand.

Above the health bar is a minimalistic simple inventory system, which gives players more flexibility in item type discovery, while also leaving more room for items to be added to the game later. However the alignment of the inventory should be updated to fit the rest of the menu which follows a left to right design philosophy.

Beneath the health bar is a larger inventory system along with a stats and level section. These UI features suggest that players can level up or discover better items, adding important context and keeping players engaged. A good example of a simple yet informative UI.

The mini map in the top left corner shows the player's current position and where they have already been. The viewing angle of the map gives the user a better sense of scale of the size of the level.

The currency display sits neatly in the top-right corner of the screen. It's out of the way but still easy to see, showing that currency is an important aspect of the game while not making the UI feel crowded. Overall the UI includes many smart decisions that could be used as a great point of reference.

### 2.1.3 Hades

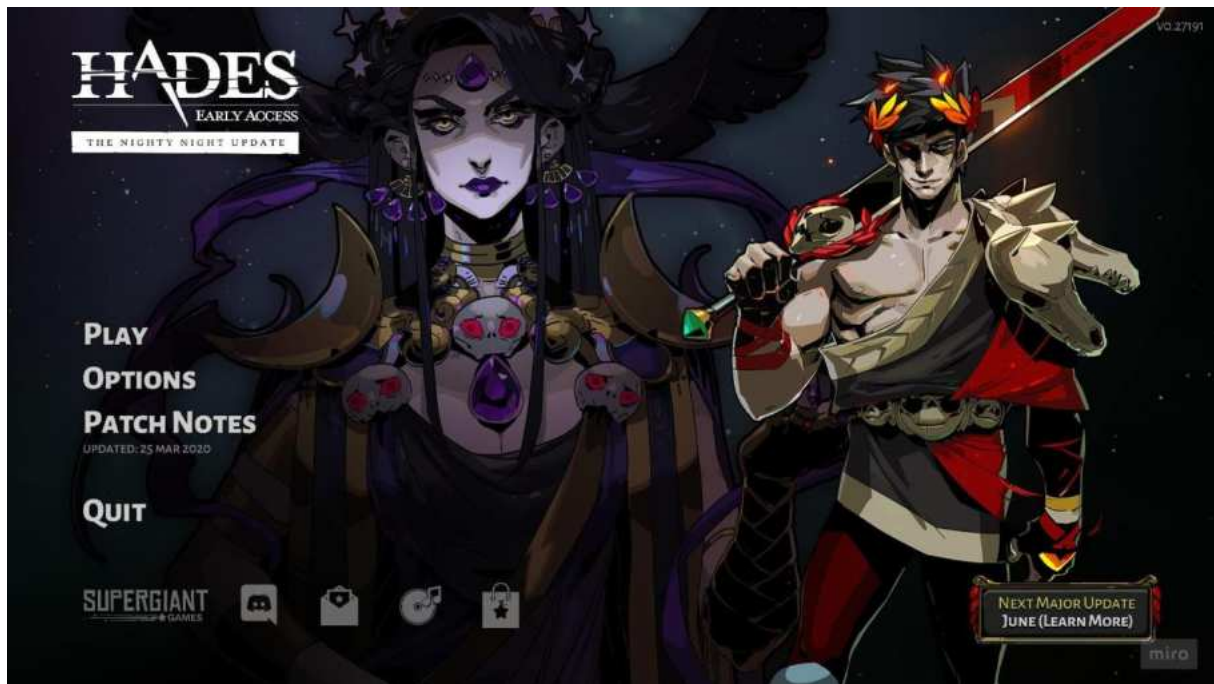


Figure 6 - Screenshot of Hades Main Menu.

The Hades main menu (Seen in Figure 6) is very simple and easy to understand. The game title is well placed in the top left corner with good spacing from the edges of the screen, making it feel well-balanced. The menu buttons are centred on the left side of the screen which works well because there aren't many button options, helping to keep the menu clean and straightforward for the user. One thing that is strange is the large gap between the Patch Notes button and the Quit button, the space seems too wide but aside from that the menu works well.



Figure 7 - Screenshot of Hades Gameplay / User Interface.



The Hades user interface (Seen in Figure 7) look overly technical can hard to understand without a tutorial. Without knowing the game, it's unclear what many of the symbols and items mean, which is something to avoid.

In the bottom right corner, there seems to be a simple inventory system with currency and potions, although their exact use isn't very clear. On the left side of the screen, there seems to be an ability system, with a health bar underneath and possibly a life counter above. However this is not explained, which makes the UI harder to use.

Overall, this design creates an unnecessary learning curve. A goal to keep in mind is to keep the interface simple and clear.



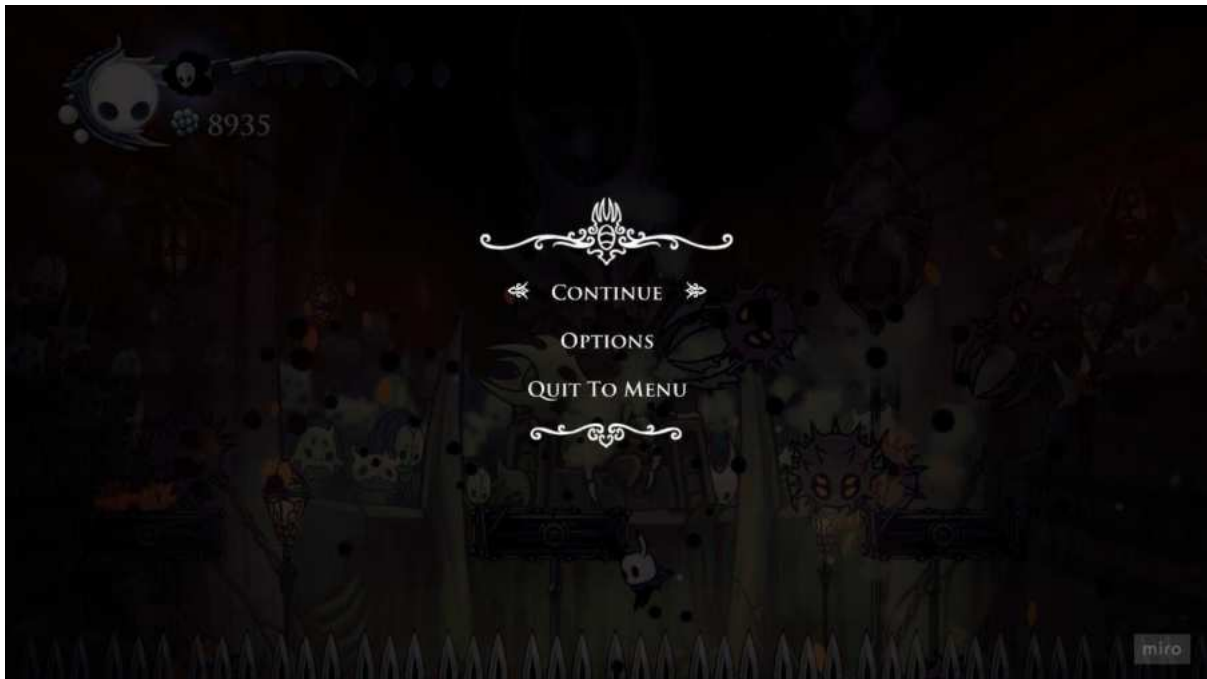
*Figure 8 - Screenshot of Hades Pause Menu.*

The Hades pause menu (Seen in Figure 8) has a simple design but could be made even cleaner. The main buttons (Continue, Settings, and Quit) are clearly listed, but menu options like Controls and Display could have been grouped under the Settings button to make the layout even simpler.

The design matches the games visual aesthetic, and the buttons are easy to see. Like many games, the menu is placed in the centre of the screen, which is a good layout choice to keep in mind for the project.

One nice detail is the small text at the bottom of the pause menu, reminding players about when their last save point was. This simple feature keeps the save point clear to the player without being intrusive, and would be a good addition if a save game system is implemented.

## 2.1.4 Hollow Knight



*Figure 9 - Hollow Knight Pause Menu.*

The pause menu in Hollow Knight (Seen in Figure 9) is simple, sleek, and well-designed, requiring no improvements. All major buttons are clearly visible, with a marker indicating the currently selected option to eliminate any player confusion. The aesthetic aligns perfectly with the game's theme, making it a strong design reference. A common pattern observed in other pause menus is the centre alignment, combined with a mid-opacity dark background and high-contrast menu elements to ensure buttons remain visible against the game backdrop.

## 2.2 Back-End Research

### 2.2.1 Development Engine Research

#### Unity

The Unity engine is a free, open-source game development engine. It uses the C# programming language and uses Visual Studio Code as its programming IDE but that can be changed if the user wishes to. It is capable of both 2D and 3D environment rendering and is most known for being the most accessible game development engine for independent game developers.

The biggest benefit of the Unity engine is how small the barrier to entry can be for new game developers. There are countless amounts of community forums and YouTube tutorials that can be freely accessed if you know the key terms to search for. The integration of Visual Studio Code is also great as it provides a platform that many programmers are familiar with. The number of third-party libraries that are available to use is also quite large which shortens the time that it takes to create assets or look for certain coding structures.

The downfalls of the Unity engine are mainly due to the price point behind certain assets or libraries which really lock some developers out of the creative space. There is also the file explorer structure which adds a lot of difficulty when communicating between different scripts, throwing many instantiation errors, but those are issues that can be solved.

## Unreal

The Unreal engine is a free game engine created by Epic Games in 1998 and is currently on its 5th iteration. It uses the C++ programming language and it is capable of rendering both 2D and 3D environments. It is most known for its stunning rendering capabilities which has led a lot of AAA game studios such as Halo Studios (Formerly 343) and CDProjektRed to abandon their home-made engines to use the newest iteration of Unreal Engine.

The biggest benefit that Unreal Engine has is simply how powerful of an engine it is and how the rendering output is done in such high detail. The Unreal blueprints are a great starting point to learn how Unreal works and some users have stated that after learning blueprints, the barrier to entry for learning C++ is much lower due to learning all the fundamental principles through the blueprint's functionality being explained.

Although Unreal blueprints is a great start to learn Unreal engine. There are also contradictory statements from other users who report that the Unreal blueprints system is quite restrictive as it only has a certain number of options for creating game objects. There are also reports that Unreal blueprints has significantly worse performance over its programming counterpart.

The lack of experience with the Unreal engine and the C++ programming language was already casting doubts on this engine being chosen, but the issues with the Unreal Blueprints system with performance and lack of creative freedoms has locked in the answer on whether this will be the chosen engine.

## Godot 4

The Godot 4 engine is a free, open-source game development engine. It has the option of being able to write games in the C# programming language or their own programming language which the developers of the engine have called GDScript. The layout of the engine is very similar to the Unity engine as well as its capabilities as an engine. It can render games in both 3D and 2D environments but seems to be more popular in the 2D pixel art space.

One of the biggest advantages of the Godot 4 engine is its built-in programming language, and the fact that all the coding is done directly inside the game engine without the use of an IDE. This setup makes coding much easier, thanks to the simple file structure that lets scripts and game objects communicate with each other more smoothly than in other engines. This structure also removes a lot of the instantiation errors that Unity developers often run into when trying to get game objects to reference each other.

One of the biggest downsides to Godot is the fact that it is a newer engine with less people using it and therefore having less resources and libraries to call upon when running into an error or trying to add a new feature to your game, therefore the learning curve could be considered much higher than the other engines.

## Conclusion

Through comprehensive research into various game development engines and an evaluation of the resources they provide, it was determined that the Unity engine would be the most suitable platform for this project. Other engines were considered heavily, however, the lack of extensive learning and troubleshooting resources presented a significant risk in development. This risk is further compounded by the absence of prior experience with these alternative engines, making Unity the most practical and reliable choice.

While each engine offers unique advantages and disadvantages, as outlined by the above breakdowns, Unity's benefits clearly outweigh the potential drawbacks of other considered platforms. The availability of comprehensive documentation, a supportive developer community, and familiarity with the engine significantly reduces the likelihood of encountering game breaking issues during development. Exploring other engines might have been viable with prior experience or better independent learning resources, but Unity remains the most effective option for achieving the project's goals.

## 2.2.2 Procedural Content Generation Research

### What is Procedural Content Generation?

“Procedural content generation (PCG) is an increasingly important area of technology within modern human-computer interaction (HCI) design. Personalization of user experience via affective and cognitive modelling, coupled with real-time adjustment of the content according to user needs and preferences are important steps toward effective and meaningful PCG. Games, Web 2.0, interface, and software design are among the most popular applications of automated content generation.” Yannakakis, G. N., & Togelius, J. (2011).

“Procedural content generation (PCG) refers to the practice of generating game content, such as levels, quests or characters, algorithmically. Motivated by the need to make games replayable, as well as to reduce authoring burden and enable particular aesthetics, many PCG methods have been devised. At the same time that researchers are adapting methods from machine learning (ML) to PCG problems.” Risi, S., & Togelius, J. (2020).

In summary, Procedural Content Generation (PCG) is the practice of using variable based algorithms to create large amounts of content for multiple different fields but is most popular in the software development industry. It provides a more streamlined output of projects by lessening the time needed for creating content from scratch, and thanks to introduction of machine learning and artificial intelligence, procedural generation has evolved even further in recent years.

### Fractal Terrain Generation

“It is obvious that fractal automatic terrain generation can save game developers a lot of time by reducing the amount of height data and they must generate themselves. During studying this algorithm, the reason fractals can be used to terrain is apparently. The main reason is that terrain is self-similar. This statement seems abstract; however, we could imagine that the magnified subsets of the objects look like the whole to each other. Taking the mountains as an example, the horizon of a mountain is not flat, but it is rugged. If we zoomed in on a part of the hillside, it would also look uneven, just like the surface of the hillside, a single rock or stone that is part of it. Therefore, using the fractal terrain generation as the article says above, a crossection of a rugged mountain is thus produced. One most typical case in today's game is Terraria, the terrain in Terraria is randomly generated by fractal terrain generation, which creates different terrain situations, such as mountains, riverbeds, and caves.” Shen, Z. (2022).

In summary, Fractal Terrain Generation is a technique that enables game developers to create realistic terrain efficiently by leveraging the self-similar nature of fractals. Natural landscapes, such as mountains, exhibit self-similar properties where smaller elements like hillsides or rocks resemble the larger structure. This characteristic makes fractals particularly suitable for simulating rugged and uneven terrain. By reducing the need for manually generated height data, this method facilitates the creation of varied and natural-looking environments. A notable



example of this technique is demonstrated in the game *Terraria*, where fractal terrain generation is used to produce randomized landscapes, including mountains, riverbeds, and caves.

### Bitmap Terrain Generation

“Unlike fractal terrain generation, this is not a completely random technique. There are already several first features on the map, and the process uses it to generate detailed data for each small feature, then the units that make up the map, which means the final terrain is something like the zooming terrain from the first terrain. Because of the unique features of zooming, bitmap terrain generation is usually the case for massively multiplayer game maps. Because the game designer may wish to have certain functions in several places, but do not care about the exact height of each square. Hence, they will indeed be very manually generating data for each square, which is very time-consuming. Thus, this technique helps to save plenty of time, otherwise, the game programmers need plenty of time consuming to generate code for each different terrain situation. Nevertheless, the generated data can be enhanced to make the terrain looks more exquisite the important thing is that bitmap terrain generation can be used to generate data other than the height of each tile and each pixel value can correspond to a specific terrain type, such as desert or jungle (which means dividing different types of terrain).” Shen, Z. (2022).

In summary, Bitmap Terrain Generation is a semi-random technique used to create game maps through predefined features. Unlike fractal terrain generation, which relies on complete randomness, this method begins with initial map parameters such as map size, the number of props, and specific generation rules. As a result, the generated terrain reflects the original map features while introducing slight variations with each iteration. This approach is particularly popular in the dungeon crawler genre, where it generates seemingly random rooms that share a similar structure and properties but rarely appear identical. Bitmap terrain generation significantly reduces development time by automating terrain creation, minimizing the need for manual coding. It also enhances detail in landscapes and supports the assignment of room or terrain types (e.g., desert or jungle areas), adding versatility to the method.

### Perlin Noise

“It is a common method to use a noise function to generate 2D-terrain, but normally we choose Perlin noise instead of normal noise. The reason we choose Perlin noise is easy to understand. Noise is a random number generator, and the random numbers generated by ordinary noise have no rules at all (Perlin noise is pseudorandom). Therefore, the cascading mountains in nature, the texture of marble, and the undulating waves on the sea surface seem to be chaotic, but there are inherent laws to follow. Normal noise cannot simulate these natural effects. The Perlin noise algorithm makes these possible. Therefore, a set of smoothly interpolated random numbers can be obtained by using the Perlin noise algorithm, which is correlated with each other and can be used to generate random values close to nature. By seeing the random texture generated by normal noise and the texture generated by Perlin noise, it can be found that the texture generated by Perlin noise is more natural and smoother, with obvious transition effects between random values.” Shen, Z. (2022).

In summary, Perlin Noise is a widely known and popular method for generating 2D terrain and simulating textures. Perlin noise produces pseudorandom values that are smoothly interpolated and correlated, resulting in natural-looking patterns with gradual transitions. This makes it particularly effective for mimicking chaotic natural phenomena within real environments. The smooth transitions provided by Perlin noise create a more visually appealing result compared to

standard noise, which often lacks structure and transition effects. Its versatility and realistic output have made it a common choice in game development for generating natural terrain and texture patterns.

### Key Findings

The above research explores Procedural Content Generation (PCG) by examining several prominent methods, including Bitmap Terrain Generation, Perlin Noise, and Fractal Terrain Generation. The initial focus was on getting an understanding about the theoretical foundations of these methods, including the underlying formulas and the historical evolution that shaped their current models. This approach provided a comprehensive understanding of how PCG operates and highlighted various ways these concepts can be applied in game development. Additionally, real-world examples were analysed to demonstrate practical application of these techniques in games, ensuring a deeper understanding of their functionality before exploring further.

The practical application of PCG methods were then investigated, with a particular emphasis on Perlin Noise and Bitmap Terrain Generation, as these are widely used and well-integrated techniques in game development. Unreal Engine 5 was examined for its PCG implementation due to its prominence among both AAA and independent game developers. The Unreal 5 Tech Demo in early 2023 showcased advanced PCG integration, demonstrating the extensive automation capabilities it offers to developers.

Further analysis was done on the use of Perlin Noise within the Unity engine, a major competitor to Unreal, which offers its own PCG system. Unity includes a built-in Perlin Noise function that can be directly linked to terrain generation scripts in both 2D and 3D environments. This integration is particularly notable for its optimization, as it avoids continuous terrain generation until memory limits are reached. Instead, Unity's method ties a single game object, minimizing memory usage while effectively creating randomized open spaces and maintaining performance.

Building on this research, principles and ideas from Bitmap Terrain Generation will be adapted and implemented into the project's procedural dungeon generation script. By applying these techniques, the project will aim to create a dynamic, efficient dungeon layouts that reflect the proven benefits of PCG systems, while also maintaining strong performance standards.

### 2.2.3 Pathfinding Algorithm Research

#### What is a Pathfinding Algorithm?

"Pathfinding refers to computing an optimal route between the specified start and goal nodes. It is an important research topic in the area of Artificial Intelligence with applications in fields such as GPS, Real-Time Strategy Games, Robotics, logistics while implemented in static or dynamic or real-world scenarios. Recent developments in pathfinding lead to more improved, accurate and faster methods and still captivates the researcher's attention for further improvement and developing new methods as more complex problems arise or being developed in AI. A great deal of research work is done in pathfinding for generating new algorithms that are fast and provide optimal path since the publication of the Dijkstra algorithm in 1959." Maurya Ananya, Yadav Aayushi, & Baiswar Ashish. (2022).

"Pathfinding is the searching technique for finding an optimal path from a starting location to a final(given) destination. The shortestpath problem is most studied in computer science.

Generally, to represent the shortest path problem we use graphs. A graph is a visual depiction of a collection of things in which some objects are linked together by links. The interconnected objects are represented by points termed vertices and the edges are the ties that connect the vertices. An optimal shortest path is defined as the minimum length criteria from a source to a destination. Pathfinding algorithm has become popular with the rise of gaming industries. Games with genres like survival, action-adventure, role-playing games, and real-time strategy games often have characters sent on missions from their current location to a predetermined destination. In these types of games, pathfinding algorithms have a dominant role. Some of the shortest path algorithms are namely as Dijkstra algorithm, Bellman-Ford algorithm, Floyd-Warshall algorithm, Genetic algorithm, A\* pathfinding algorithm, etc. Unity-3d is a game engine that is used by most of the gaming industries and indie game developers. This software is available for free which is one of the reasons for its high usage in the gaming industry. Unity-3d is a complete integrated development environment (IDE) with an asset workflow, scripting, integrated editor networking, scene builder, and more. It also includes a large community and forum where anyone interested in learning Unity can go and get answers to all of their questions. In unity-3D we use the c# programming language. Unity is a cross-platform developing software that is easy to learn for beginners and powerful enough for experts.” Iqbal, M. A., Panwar, H., & Singh, S. P. (2022).

In summary, pathfinding is the process of finding the most efficient and effective route to a destination. It is a key technique within artificial intelligence, with application in GPS, navigation and video games. Pathfinding algorithms such as Dijkstra’s, A\*, and Bellman-Ford are some of the most common algorithms when working with pathfinding but the A\* method is by far the most popular within game development. The development of faster and more accurate algorithms continues to be a focus of research, particularly a new challenge in AI and game development.

#### Breadth-First Search (BFS):

“Breadth-first search, in 1959. BFS explores equally in the directions until the goal is reached. Alternatively, we can say that it starts from a chosen node and examine its neighbour, the node which has been traversed is marked as visited. Breadth-first seeks is a graph traversal set of rules that begins of evolved by traversing the graph from the basis node and exploring all the neighbouring nodes. Then, it selects the closest node and explores all the unexplored nodes. While the usage of BFS for traversal, any node within the side of the graph may be taken into consideration as the basis node. BFS uses a queue (FIFO). BFS guarantees the shortest path. The data structure used to represent the graph determines BFS’s temporal complexity. The time complexity of the BFS algorithm is  $O(V+E)$ , where V is the number of vertices, whereas E is the number of edges. The space complexity is of BFS can be expressed as  $O(V)$ .” Iqbal, M. A., Panwar, H., & Singh, S. P. (2022).

#### Greedy Best First Search (Greedy Search):

“The greedy best-first search algorithm always selects the path that appears to be the most appealing at the time. It is defined as the combination of depth-first and breadth-first search algorithms. It uses both heuristics and search functions to perform its operations. We can use both methods while using the best-first search.

At each stage, we may use the best-first search algorithm to select the most promising node from the graph. We expand the node that is closest to the goal node in the best-first search

process, and the closest cost is determined using a heuristic function, i.e. For GreedyBFS the evaluation function  $f(n)$  is given as:

Where  $h(n)$  is the heuristic function which is defined as the distance of approximation of how close we are to the goal from a given node. The time complexity of the algorithm is given as  $O(n \cdot \log n)$ ." Iqbal, M. A., Panwar, H., & Singh, S. P. (2022).

#### A\* (A-Star):

"Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A\* Search Algorithm comes to the rescue.

What A\* Search Algorithm does is that at each step it picks the node according to a value- $f$  which is a parameter equal to the sum of two other parameters –  $g$  and  $h$ . At each step it picks the node/cell having the lowest  $f$ , and process that node/cell.

We define  $g$  and  $h$  as simply as possible below.  $g$  = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

$h$  = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this  $h$  which are discussed in the later sections." Belwariar, R. (2018, September 7).

#### Key Findings:

When researching pathfinding algorithms, the initial focus was on popular methods that were commonly used by game developers to gain a better understanding of how they worked and they were the preferred methods. This research highlighted methods such as Breadth-First Search (BFS), Greedy Best-First Search (Greedy Search), and A\* (A-Star) as widely used approaches to pathfinding within the game development community. All of these methods are based on the same core principles, calculating the shortest possible route using directional nodes and evaluating the cost to reach the target node.

After gaining an understanding of how these algorithms work, further investigation into their practical application in game engines showed that the functional implementations are limited, with most engines relying on built-in solutions like NavMeshComponents. The Nav Mesh system is based on the A\* pathfinding algorithm, which research confirms as the most popular and adaptable pathfinding method for game development. The A\* pathfinding algorithm's strength lies in its ability to work seamlessly across both 2D and 3D environments without needing large adjustments, making it highly versatile. Its integration into most modern game engines highlights its importance in the industry.

Overall, pathfinding algorithms play a critical role in game development, with A\* standing out as the most effective option due to its efficiency and adaptability. While algorithms like BFS and Greedy Search provide great insights, A\*'s usability has made it the industry standard for route planning. For these reasons, the chosen system will be the A\* pathfinding algorithm for this project.

## 3 Requirements

### 3.1 Introduction

The requirements phase is an important part of the development process, as it establishes the foundation for the project. This chapter outlines the key of functionality and performance goals needed to meet the project's aims. By researching similar games and considering design and user experience principles, the requirements were chosen to help make sure they were made to help make sure the final product is both technically strong and enjoyable to play, all while staying within a sizable scope.

### 3.2 Requirements Gathering

To ensure an in-depth understanding of the requirements needed for this project was gathered, research into several popular titles within the roguelike genre was conducted, these games include, The Binding of Isaac, Vagante, and Hades. Through analysing these games, key insights were gathered into essential gameplay elements, design philosophies, and structural frameworks commonly used in this genre. Most of the design research into these games was covered in detail in Chapter 2. Core gameplay elements like procedural generation, enemy behaviour and player progression systems were broken down to better understand how they work to create a strong gameplay experience. Based on this research, preliminary code structures were also planned to help guide the technical side of the project.

### 3.3 Requirements Modelling

#### 3.3.1 Functional Requirements

##### Main Menu

A script will be created to control the main menu, Unity's Canvas Panel will be used for a fast and responsive menu setup. Switching between panels will be managed using the `SetActive()` method from Unity's UI tools. The script will also use Unity's Scene Manage system to make the Play and Exit buttons work. The Options button will switch panels within the same scene, avoiding unnecessary scene loading and keep the menu performance smooth.

##### Player

A movement script will be created and will be integrated into the New Unity Input System to support both keyboard and controller inputs, the input system will make it far easier to create a control scheme for both input methods. This script will also handle the knockback functionality, allowing it to be triggered by enemy interactions. The knockback system will apply directional force based on the enemy's position, ensuring accurate and responsive player knockback.

A dedicated script will be created to manage the player's health, separated from the movement script to simplify future debugging and improve code maintenance. This health script will handle interactions with enemies, monitor health values, and remove the player from the scene when the player's health reaches zero.

A dedicated script will be created to manage the player's combat functionality. This script will handle interactions with enemies and will be responsible for removing an enemy from the scene once its health reaches zero. The combat will be built using a game object with a trigger that interacts with the enemy.

### Enemy

A script will be created to control the enemy's pathfinding. This pathfinding system must be capable of detecting obstacles and determining the shortest route to the player. The enemy will also need to interact with the player's movement script to trigger the knockback function. Within the same interaction, the script can also deduct a specified amount of health from the player's health script, allowing both the damage and knockback mechanics to be managed efficiently from the enemy's side.

### Room Generation

A script will be developed to control the generation of the game world, with the world being randomized each time the scene is loaded. The system must be capable of detecting neighbouring rooms to support door functionality. Integration with Unity's Tilemap system is necessary to enable enemy obstruction pathfinding and player collisions with closed doors and walls, ensuring that unplayable areas remain inaccessible.

The random generation system will also require configurable enemy parameters, including spawn positions and the number of enemies per room. Additionally, camera functionality will need to dynamically follow the player through different rooms by using trigger points to adjust the camera position as the player transitions between different areas.

### Player / Enemy Health System

A dedicated health system will be required for both the player and enemies. The player's health system must be able to interact with the enemy's combat system, triggering either deactivation or destruction of the player object when health reaches zero. Similarly, the enemy health system should function the same way, removing the enemy from the scene when reaching zero health, while maintaining compatibility with the player's combat system.

### Player / Enemy Combat System

Two independent scripts will be needed to handle the combat systems for both the player and enemies. Each script should be capable of interacting with the corresponding health system, player combat affecting enemy health, and enemy combat affecting player health. Both systems must also trigger knockback effects for their target. Following a decoupled structure will help maintain modularity and simplify integration between components.

### Player / Enemy Knockback System

The enemy will need a script designed to trigger the knockback function within the player's movement system. Alternatively, this trigger could be incorporated into the enemy's combat script to reduce the overall number of scripts. However, keeping the systems decoupled allows both the knockback and health systems to activate simultaneously, resulting in smoother gameplay. Similarly, the player's combat system must be able to trigger the enemy's knockback function. Maintaining consistency across both systems helps ensure a more polished and reliable gameplay experience.

### Pause Menu

A script will be required to manage the status of the pause menu. The Unity Scene Management library will be used to allow the pause menu to exit to the main menu, creating a loop that can reset the game level scene. This script will also handle the integration of an options screen by toggling between the pause and options menus. When either menu is active, the user interface will be deactivated to prevent overlap between the two interface systems.

## User Interface

The user interface will need to include a health bar and a mini map to help the player keep track of their health and explore potential areas. The health bar will reflect the player's current health by using variables from the player's health script, adjusting its size based on the player's remaining health. The mini map will be created using a render texture and a separate camera set to a different layer. The user interface will also need an experience level counter and a visual inventory system if those features get added into the project.

### 3.3.2 Non-Functional Requirements

#### Main Menu

A clear and user-friendly main menu layout is essential to ensure players can easily navigate the interface. The overall visual design should reflect the game's colour palette and aesthetic to maintain a consistent look and feel across all menus.

To inform the layout and functionality of the menu, research materials from similar games within the same genre will be gathered to help identify common design patterns and player expectations. The goal is to create an intuitive menu that requires no additional guidance to use effectively.

Once the main menu design is finalized, it can serve as a template for the options and pause menus. This approach ensures visual consistency and provides a better user experience throughout the game.

#### Player

A sprite sheet will be required for the player which includes idle animations facing all four cardinal directions. These animations will help indicate the player's last movement direction, which is important for determining the correct direction of the next attack.

The player's sprite sheet will also need to include running animations for all four cardinal directions. These animations will transition from the idle state to accurately reflect the player's movement direction. Special attention should be given to horizontal movement to avoid overlapping animation triggers and ensure smooth transitions.

The player's sprite sheet should also include attack animations for all four cardinal directions. These animations will work in conjunction with the idle and running animations to clearly indicate the direction of the player's attack. The attack animations will be triggered using the Any State feature in the animation controller, allowing them to run independently of the current animation state, provided their conditions are met. Exit times will be set to ensure each attack animation fully plays before another begins. Additionally, the player's attack hitboxes should be carefully aligned with the swing of each animation to give a clear visual indication of the attack's effective area.

The player character's sprite sheet should also include damage animations for all four cardinal directions. These animations can be integrated using the Any State feature in the animation controller, ensuring they play without overlap if the specified conditions are met. Including directional variations helps visually indicate the direction from which the player takes damage, reinforcing the accuracy of the knockback system. Exit times will be applied to these animations to ensure they fully complete before transitioning to other states.

The animation controller should be configured with appropriate Booleans and triggers to allow seamless integration of animations into the corresponding player scripts.

### Enemy

A sprite sheet for the enemy character should include a single idle animation, intended for use when the player is not present in the room. Although the idle animation may not be strictly necessary due to the player not seeing it very often, its inclusion ensures smooth and consistent transitions between animation states across various game components.

The same sprite sheet must also contain movement animations for all four cardinal directions, as these are essential for when the enemy is actively pursuing the player within a room.

Damage animations in all four directions are also required and should be linked to the enemy's current movement direction. When triggered, the appropriate damage animation will play based on the last direction of movement. These animations should be connected via the Any State node in the animation controller and configured with exit time to ensure clean transitions without overlap, allowing the animation to complete before returning to movement.

Death animations for all four cardinal directions should also be included to reflect the direction the enemy was facing at the time of defeat. These will also use the Any State node and be set with exit time to allow full completion of the animation before the enemy is removed or deactivated in the scene.

The animation controller must be set up with the necessary Booleans and triggers to enable integration of all animations into the appropriate enemy behaviour functions.

### Room Generation

An appropriate tile map from the Unity Asset Store will be required for the room design. This Tilemap should include walls from various perspectives, props for both open and closed doors that can be integrated into the walls, door hole props for wall integration, and obstacle props to diversify the rooms.

### User Interface

The User Interface design should include a portion of the screen dedicated to displaying the player's health bar, as this is a clear and informative way to communicate the player's current status. The design should align with the overall aesthetic of the game.

Additionally, a mini map will need to be designed to assist the player in navigating the world. Since the world is procedurally generated, it is important to ensure that the player can easily see available rooms, with a clear indication of whether each room has been explored or not.

A player hot bar would be great to add as well to have a visual indication of the player's inventory and what is currently selected in their arsenal if there is an inventory system brought into the project. A dedicated experience bar would also be great to have if there is time to add that to the game. When designing the user interface, this idea will be kept in mind in case there is time to make this idea into a reality.

### Pause Menu

An appropriate pause menu should be designed to clearly present the available options to the player while providing an easy and intuitive navigation system. Research will be conducted on



pause menus from popular games within the same genre to better understand how these systems feel and operate.

### 3.4 Feasibility

After breaking down all requirements and conducting extensive research, the completion of this project appears highly feasible, particularly given the larger development time allocated to this project. A significant portion of time will be dedicated to refining the procedural generation scripts to ensure the room generation system functions as intended without issues. Once this foundation is in place, the remaining aspects of the project should come together quickly, supported by extensive experience in building menu systems, scene navigation systems, and player mechanics.

A decoupled approach will be used, breaking each implementation into separate components to streamline development. This strategy not only speeds up progress but also simplifies debugging, leading to overall improved production. The front-end design process is not expected to be time-intensive, as many assets will be sourced from third-party suppliers and the Unity Asset Store. Additionally, integrating back-end systems with the user interface is a familiar process, further reinforcing confidence in the project's successful execution.

### 3.5 Conclusion

This chapter provided a clear breakdown of the project's requirements, covering both functional and non-functional needs. Research into existing games and design principles helped identify and plan key gameplay mechanics, UI elements, and system structures. A decoupled design approach was chosen to make development, testing and integration easier for the project. The feasibility breakdown showed that this project realistic, supported by a clear development timeline that was outlined in the project's proposal. The use of external assets will also help to speed up the work. With these requirements in place, development will be able to move forward with a strong technical foundation.

## 4 Design

### 4.1 Introduction

This chapter outlines the design process used to develop the project. It covers two main areas, user interface design and programme design. The first section will look at the back-end's architecture through the project's file structure to show what design patterns were used to keep the code maintainable. The second section will discuss the UI research conducted in Chapter 2 and how it guided the creation of the UI wireframes.

## 4.2 Programme Design

### 4.2.1 Unity Structure

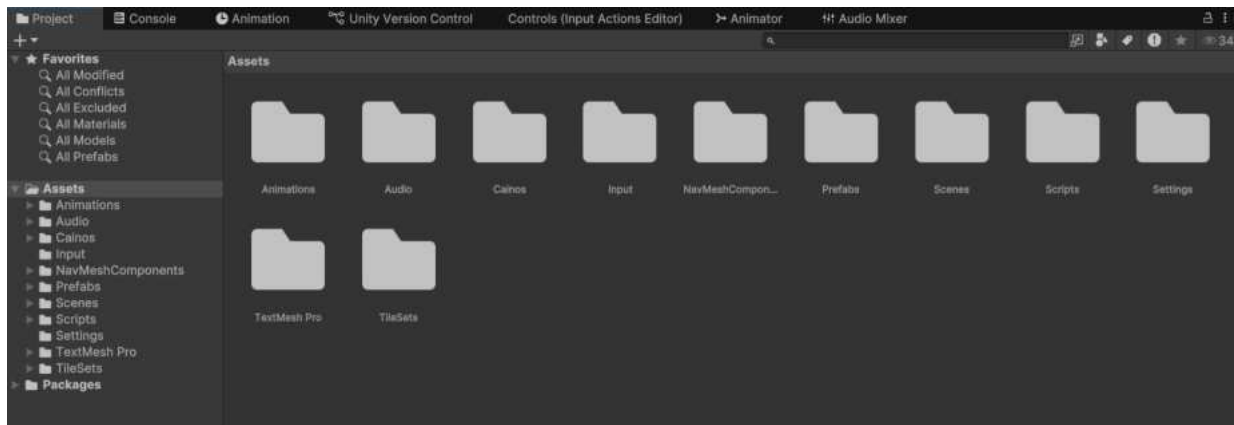


Figure 10 - Screenshot of Unity File Explorer.

This overall setup of folders (Seen in Figure 10) was made to ensure that everything created in, or imported into the project could be stored away but also found and accessed with ease. Having everything in dedicated folders was very beneficial, especially in later stages of development when the project was at its peak amount of assets.

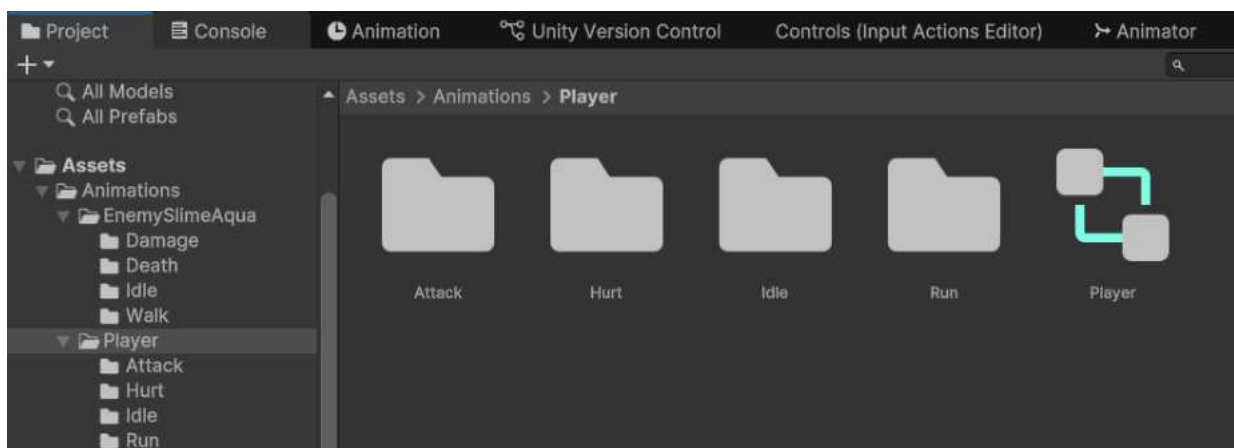


Figure 11 - Screenshot of Animation File Structure.

The structure of the Animation Files (Seen in Figure 11) is organized in a simple and logical way. All animation-related files are stored within a main “Animations” folder, which is then divided into categories based on the game objects they belong to. For example, the Player and the Enemy each have their own subfolders. Inside these folders, animations are further separated into specific actions, alongside the corresponding sprite sheets used to create them. This setup keeps the animation assets easy to manage and locate.

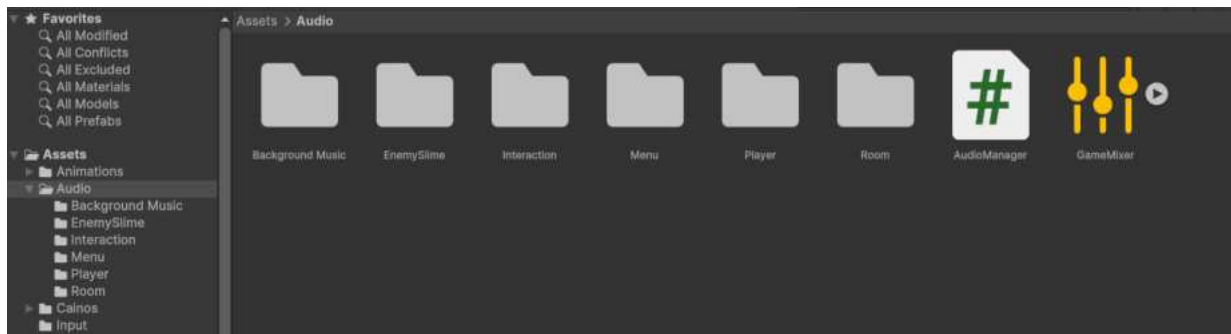


Figure 12 - Screenshot of Audio File Structure.

The structure of the Audio Files (Seen in Figure 12) is very similar to the structure of the Animations files. Every audio related file is placed within the Audio Folder, these files are then categorized into their respective subfolders. If the audio file is to do with the player's sound effects, then they will be placed in the Player category. This file structure made it very easy to differentiate between different sounds and what they were set to be dedicated to when doing the audio design for the game. Having the game mixer and the Audio Manager script in the overall audio file made sense as well as they were made to control all of the audio settings within the game.

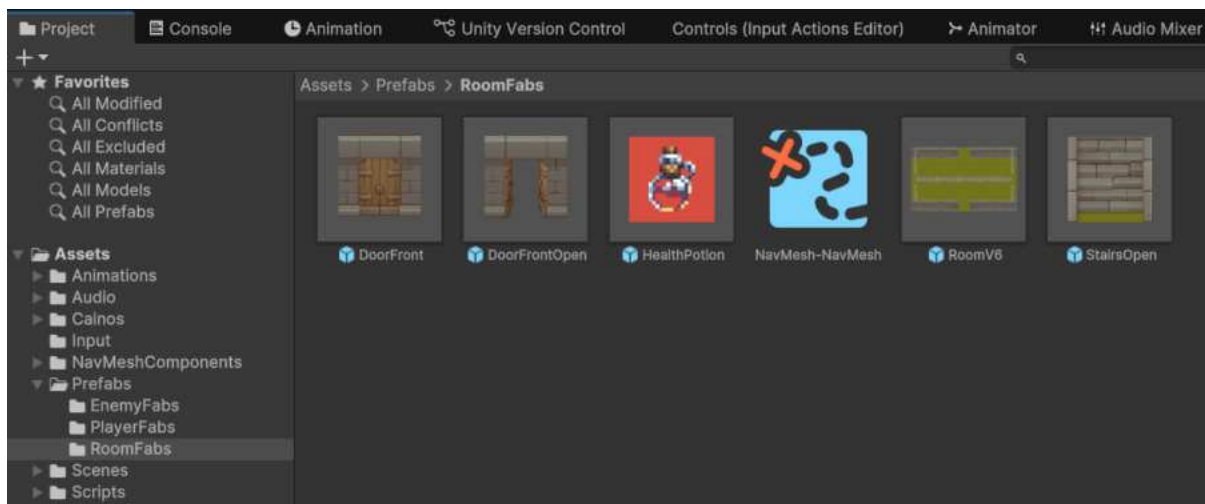


Figure 13 - Screenshot of Prefabs File Structure.

The structure of the Prefabs files (Seen in Figure 13) kept the same philosophy that was built with the Audio and Animations files. Everything prefab related gets brought into the Prefab folder and is then categorized into their respective subfolder. Everything related to the room prefab and how it operates is brought into the RoomFabs Folder. This is the same with the EnemyFabs and PlayerFabs folder. This file structure philosophy made it very easy to navigate, having everything neatly placed away into categories was very beneficial in later stages of development.

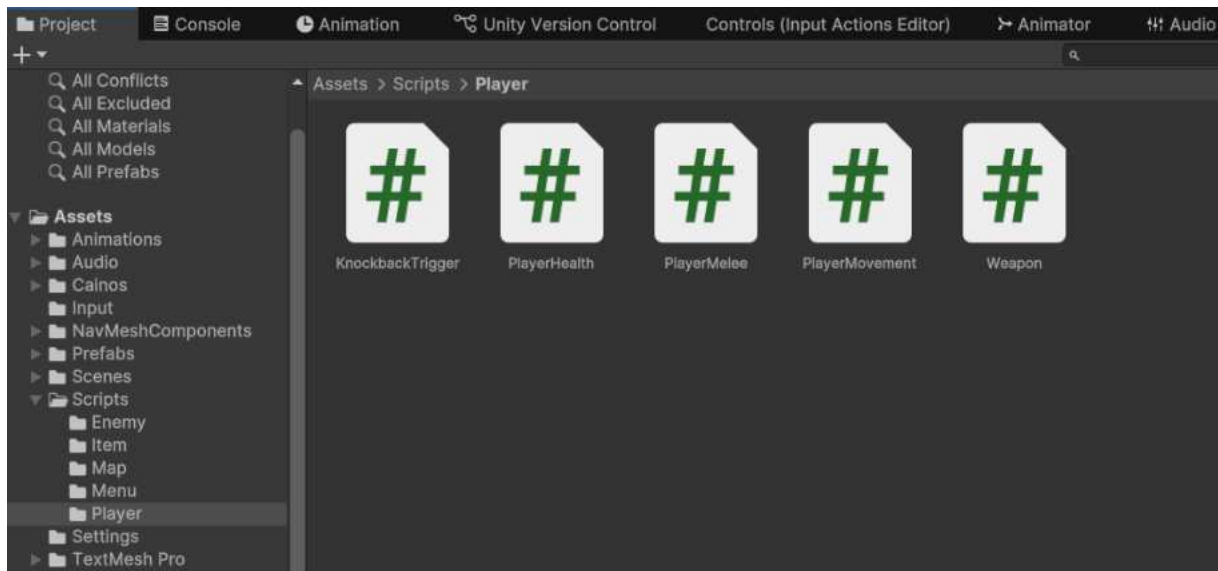


Figure 14 - Screenshot of Scripts File Structure.

The structure of the Scripts files (Seen in Figure 14) keeps the same structure philosophy as the file paths that have been previously mentioned. There is a main Scripts file which is home to the categories of which each script can be placed into, we have Player scripts, Enemy scripts, Map scripts a few others. This file structure gives the first glimpse of the decoupled back-end game design that was mentioned in Chapter 1. This file structure was particularly important for the scripts as there were so many written, finding where to modify functions would have taken far longer if the files were not organized this way.

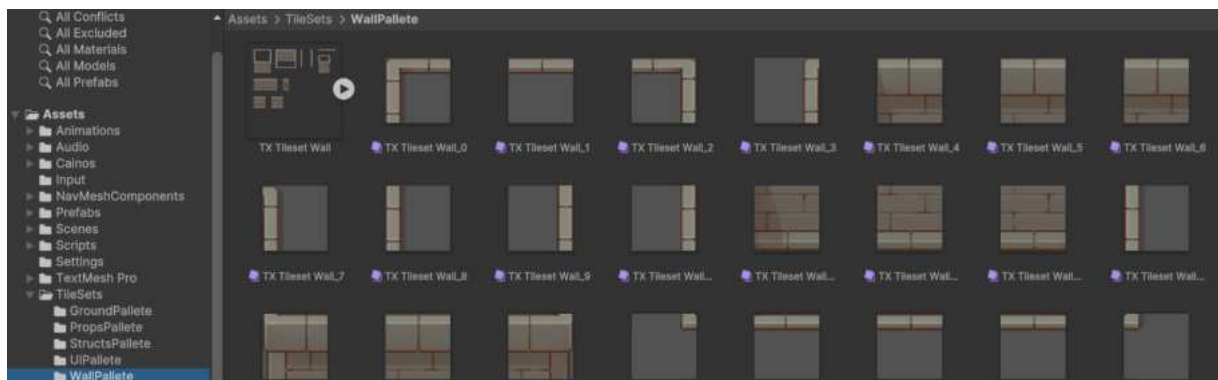


Figure 15 - Screenshot of TileSets File Structure

The structure of the TileSets folder (Seen in Figure 15) is very similar to the previously discussed file structures, which all follow the same structure philosophy. Each tile set that was brought into the project was brought into the TileSet folder and then categorized into each tile set they were for. Some of these TileSets include GroundPalette, and WallPalette which contain the ground tiles and wall tiles which were used to design the game. These folders were great for the creation of tile prefabs, being able to find the specific tile I need by number without losing them in a sea of irrelevant tiles. This file structure proved extremely useful for the duration of the project as there wasn't many tile sets brought in after the original batch.

Other files such as the NavMeshComponents files and Cainos files were brought in with other 3<sup>rd</sup> party assets and were kept to make sure that I wasn't removing important materials or scripts

for these 3rd party assets. These files don't reflect the structure philosophy I was going for when managing these files.

### 4.2.2 Design Pattern

The back-end development of the project followed a slow and methodical approach, with a strong focus on maintainability. Small, meaningful changes were introduced incrementally to ensure each addition was properly integrated and tested. Scripts and features were consistently separated to make debugging easier. For example, the Player was built using individual scripts for movement, combat, and health, allowing for a more focused development and quicker troubleshooting. This structure was maintained throughout the project, making final code adjustments easier as the game moved toward the build stage. Overall, this approach helped keep the codebase clear and improved the stability of the final product.

## 4.3 User Interface Design

### 4.3.1 Wireframes



Figure 16 - Screenshot of Main Menu Wireframe.

The project's main menu wireframe (Seen in Figure 16) takes heavy inspiration from *Vagante* (Seen in Figure 4), particularly in how it positions the game title and menu components in a unique and well-executed manner. The selected hover effect effectively highlights the currently selected option, enhancing usability. This menu design strikes a balance between uniqueness and simplicity by avoiding unnecessary functionality. To maintain design consistency across different parts of the game, the container concept used in the gameplay user interface has also been incorporated. This approach was not necessary for the pause menu, as its centred alignment already provides ample space.

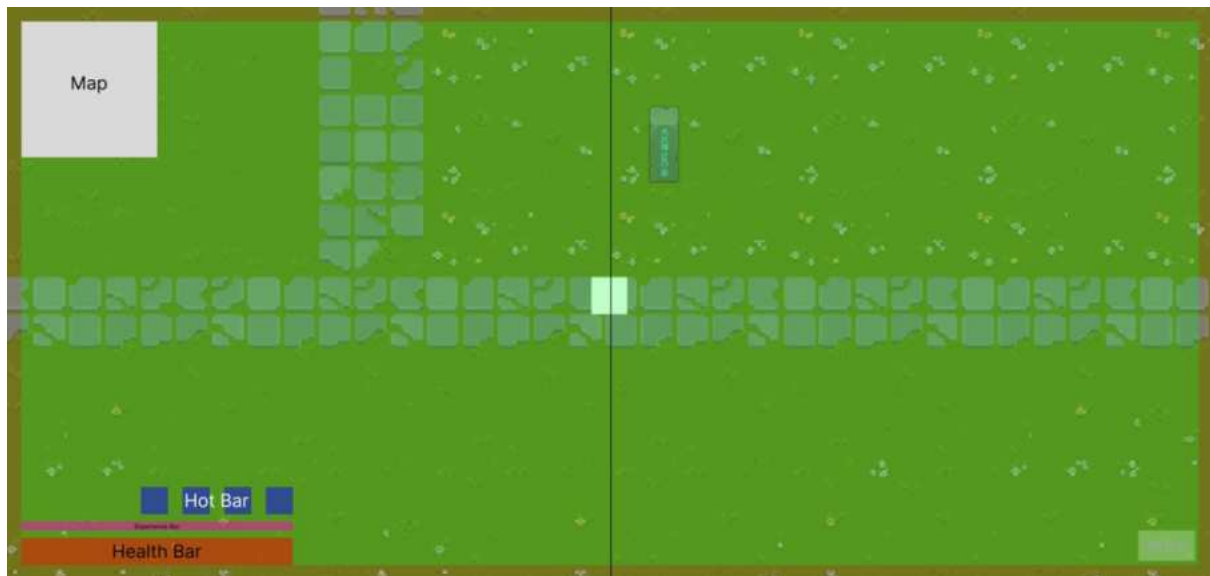


Figure 17 - Screenshot of User Interface Wireframe.

The user interface wireframe (Seen in Figure 17) was heavily inspired by Vagante's user interface (Seen in Figure 5). Vagante was chosen as a reference due to its extensive user interface mechanics, many of which align with the features which are planned to be implemented. Its design principles prioritize clarity, ensuring that all menu components are well-labelled and easy to understand, minimizing player confusion. These principles are being followed as closely as possible to create an intuitive experience for new players. Additionally, interface components are positioned along the outskirts of the screen, keeping the central gameplay area clear for better visibility of enemies. The container for the user interface takes inspiration from web design principles, maintaining a slight distance from the screen edges to enhance overall aesthetics and create a more polished, stylish layout.

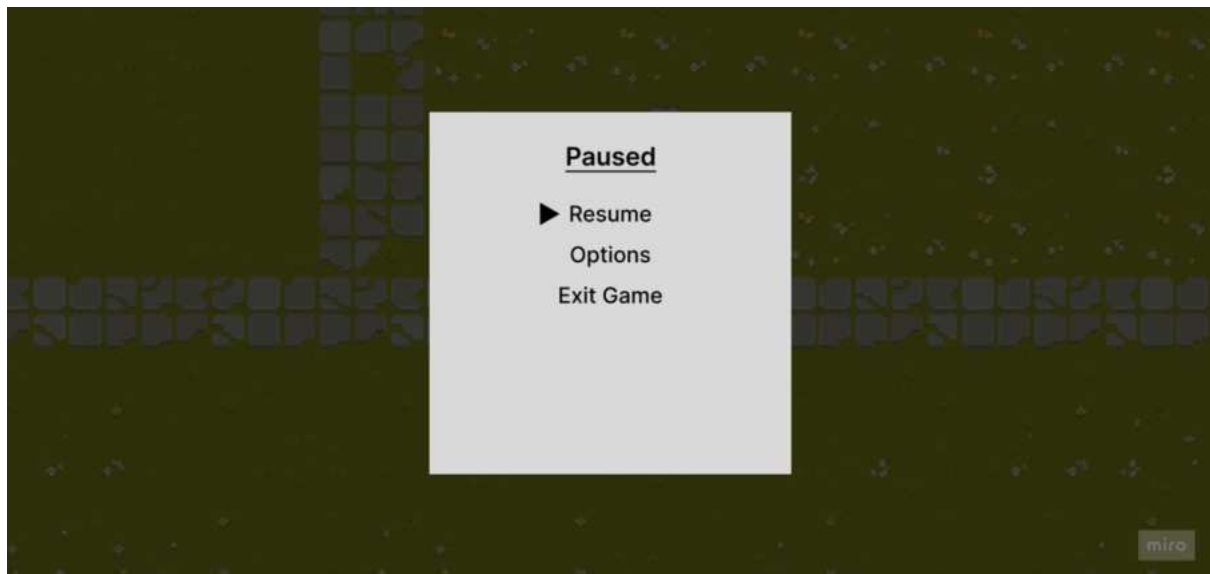


Figure 18 - Screenshot of Pause Menu Wireframe.

The pause menu wireframe (Seen in Figure 18) takes heavy inspiration from the Hollow Knight and Hades pause menu (Seen in Figure 8). Their pause menu layout maintains a simple, easy-to-understand structure while incorporating enough visual design to remain appealing. The low-

opacity background helps distinguish the menu from the gameplay, clearly indicating a change in game status. Additionally, placing the "Paused" title at the top centre of the screen eliminates any confusion about the player's action. The three main menu components are easily readable due to their large font size, while the hover effect enhances clarity by highlighting the currently selected option. To further improve readability, the pause container provides a contrasting background for the text, preventing it from blending into the semi-transparent game screen. This container approach is being explored based on previous experiences where pause menu text became difficult to read against the background.

### 4.3.2 User Flow Diagram

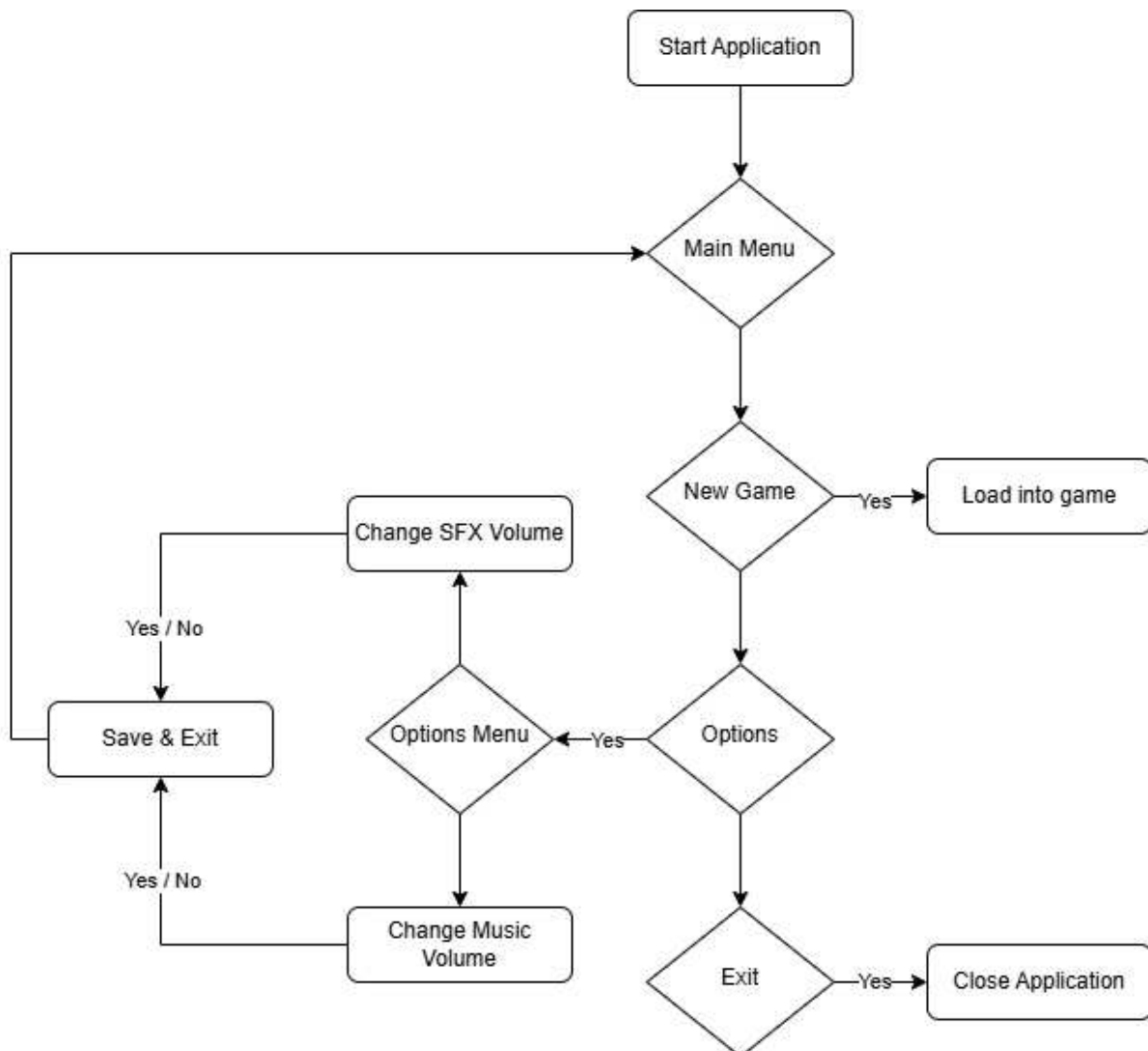


Figure 19 - Main Menu User Flow Chart

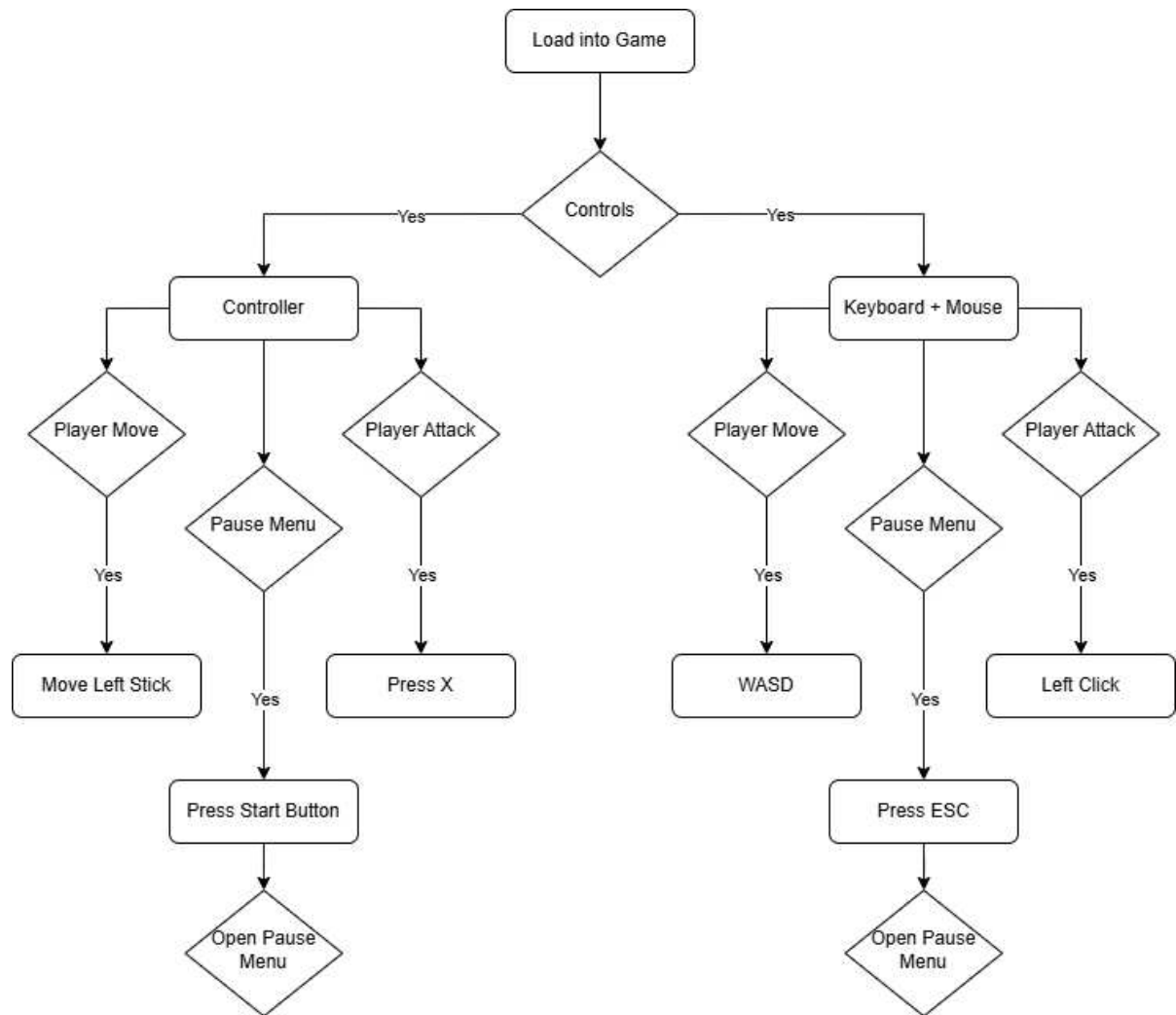


Figure 20 - Player Controls User Flow Chart



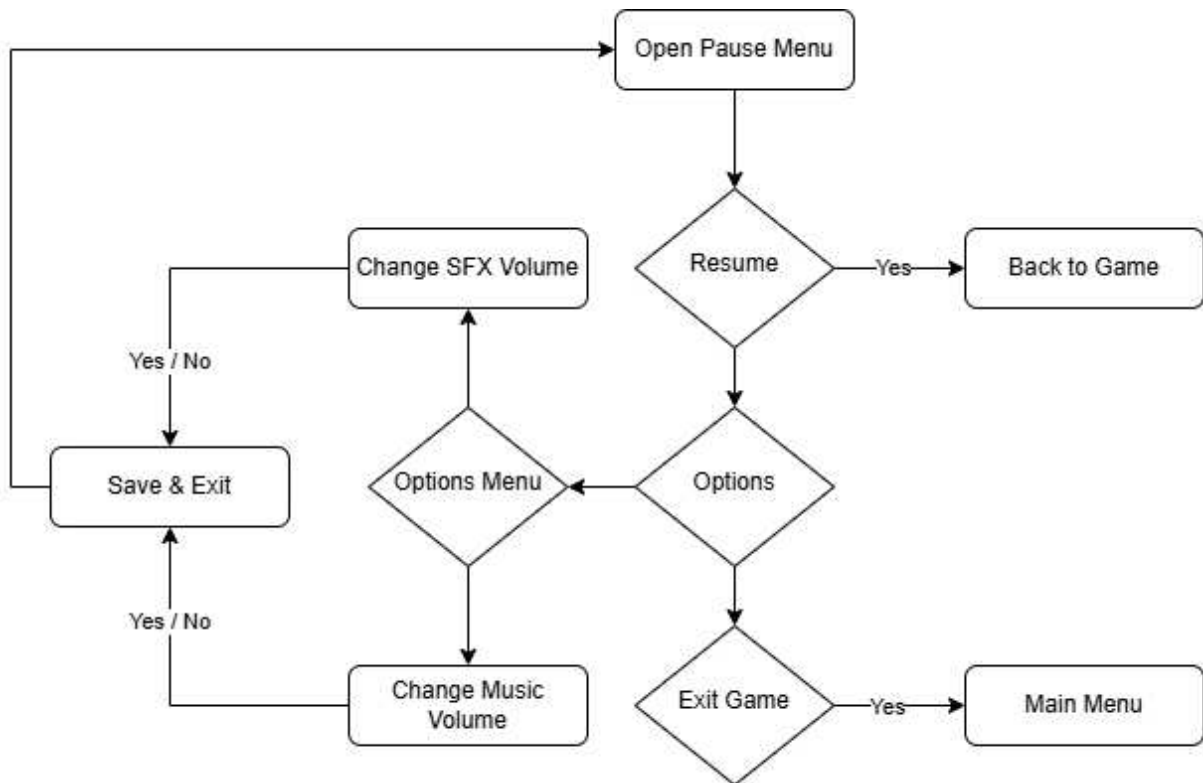


Figure 21 - Pause Menu User Flow Chart

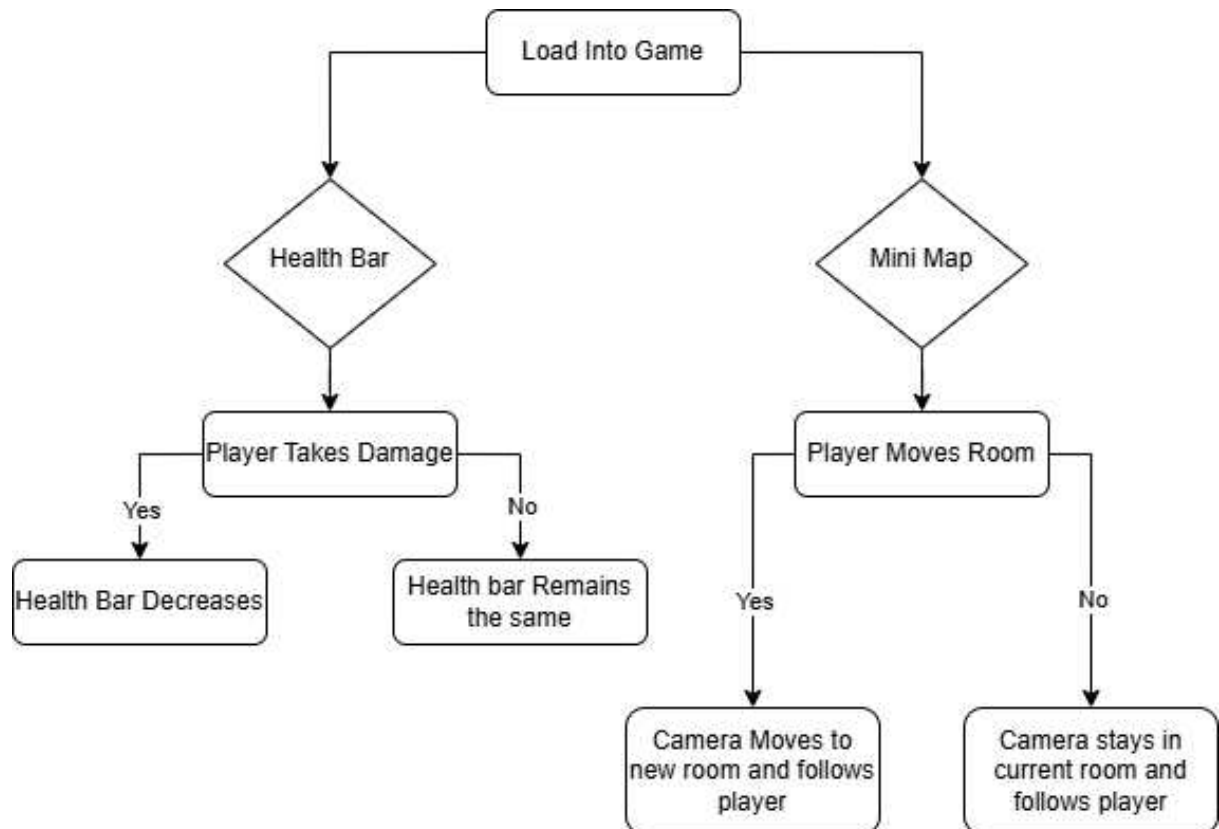


Figure 22 - User Interface User Flow Chart

### 4.3.4 Level Design

The level design for this project is heavily inspired by *The Binding of Isaac* (Seen in Figure 2). It uses a repeating room structure, where the contents and door positions are randomized each time the game is played. This keeps the experience fresh and makes each room feel different. The starting room is the only room that is generated the same way, giving the player a calm and familiar entry point. This design is easy to iterate on, allowing for a dynamic experience while also reusing assets to save time and focus more on development.

## 4.4 Conclusion

This chapter outlined the structured approach used to design both the system architecture and the user interface of the project. A well-organized file structure and clear separation of scripts helped streamline the development, improving the maintainability and debugging efficiency. The chosen design patterns also supported a modular approach, allowing flexibility for future updates.

Research into the user interface design across similar games provided valuable insights into the best design practices for improving the user experience. By analysing menus and interface elements, the user interface was crafted to balance clarity, usability, and visual appeal. The wireframing process reinforced these design decisions, ensuring the user interface aligned with the specific needs of the project.

Overall, the design choices made in this chapter created a strong foundation for a structured, intuitive, and a visually appealing game experience. Combining solid technical architecture with user-centred design principles ensures that the final product is both functional and engaging.

# 5 Implementation

## 5.1 Introduction

This chapter provides a comprehensive breakdown of the project's development process, detailing the implementation of all major systems across each sprint. It covers the full range of technical work undertaken, including feature development, feature modification, bug discovery and resolution, animation integration, audio system implementation and much more. Each section highlights the goals set and the challenges encountered during development and how each challenge was overcome, offering a clear view of how the project evolved from initial concepts to the functional final product.

## 5.2 Sprint 1

### 5.2.1 Goals

- Have a full breakdown of game requirements.
- Organize previous project research.
- Begin collecting design research materials.
- Begin collecting system tutorial breakdowns for back end.

### 5.2.2 Goal 1 – Functional Research

During this phase of the project, research was conducted on popular games within the same genre to identify both functional and non-functional requirements. These are discussed in detail in Chapter 3.

### 5.2.3 Goal 2 – Back-End System Research

Following the requirements breakdown, further research was conducted into game engines, procedural generation, and pathfinding algorithms to determine the most suitable technologies for the project. This research is explored in detail in Chapter 3.

### 5.2.4 Goal 3 – Gathering Applied Research

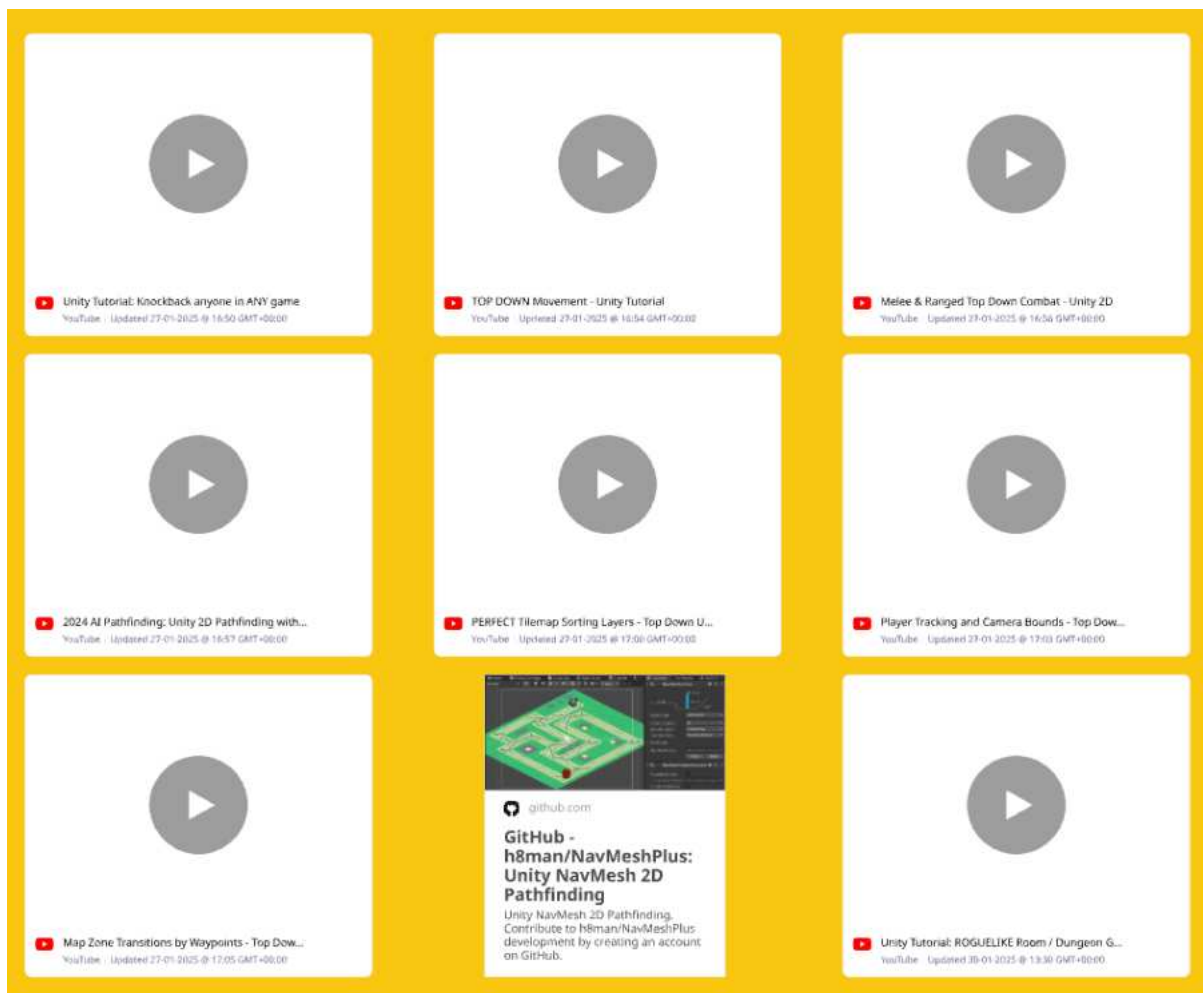


Figure 23 - Snippet of back-end system research.

Following the research into procedural generation and pathfinding algorithms, examples of applied systems were gathered to understand how they have can be implemented in games (Seen in Figure 23). Both simple and complex systems were explored to evaluate the feasibility of building and integrating them cohesively. This included foundational elements such as top-down 2D movement, as well as more advanced systems like stage-based camera tracking and basic random room generation.

Once the back-end systems researched was gathered, an initial design document was created. This included collecting and analysing references of user interfaces from games within the

same genre to inform the development of a cohesive design language for the project. Front-end research and breakdowns are discussed in detail in Chapter 3.

## 5.3 Sprint 2

### 5.3.1 Goals

- Begin to break down the design languages
- Start putting together a wireframe based on the breakdown

### 5.3.2 Goal 1 – Front-End Design Breakdown

During this stage of the project, the front-end research and breakdowns were conducted to better understand user interface design patterns within the genre. These breakdowns are spoken about in detail in Chapter 2.

### 5.3.3 Goal 2 – Wireframe Creation

The initial wireframes for the game were developed during this development phase. Using assets from the Unity Asset Store, I created a basic test environment to visualise interface layout and functionality. Screenshots from this environment were imported into Figma to begin designing the first iterations of the user interface and pause menu. A comprehensive breakdown of these wireframes and their design rationale is provided in Chapter 4.

## 5.4 Sprint 3

### 5.4.1 Goals

- Create a test level with a moveable player.
- Create a camera system that can follow the player through different stages.
- Create a basic enemy that follows the player.
- Create a basic player attack script.
- Create a basic knock back script.
- Create a basic player and enemy health script.

### 5.4.2 Goal 1 – Creating a Development Area



*Figure 24 - Screenshot of Test Level.*

The test level (Seen in Figure 24) was created using Unity's Tilemap functionality, supported by a 2D asset pack sourced from the Unity Asset Store. As this was the first time working with the Grid and Tilemap system, care was taken to build a strong understanding of the basics, including importing tile palettes and setting up a grid with multiple sorting layers. Particular attention was given to creating separate player and collision layers, ensuring they were stacked correctly so the player could interact with the environment properly rather than walking through the floors or walls. Applied research into Unity's Tilemap system proved highly valuable in gaining a solid understanding of its functionality.

```

1  using UnityEngine;
2
3  0 references
4  public class PlayerMovement : MonoBehaviour
5  {
6      1 reference
7      [SerializeField] private float _moveSpeed = 5f;
8
9      2 references
10     private Vector2 _movement;
11
12     2 references
13     private Rigidbody2D _rb;
14
15     0 references
16     void Awake()
17     {
18         _rb = GetComponent<Rigidbody2D>();
19     }
20
21     0 references
22     void Update()
23     {
24         _movement.Set(InputManager.Movement.x, InputManager.Movement.y);
25         _rb.linearVelocity = _movement * _moveSpeed;
26     }
27 }

```

Figure 25 - Code snippet of Player Movement script.

The Player Movement script (Seen in Figure 25) provides a basic 2D movement mechanic using Unity's physics system. The movement direction is determined by input values which are stored in the Unity Input Manager as a Vector2. This vector is multiplied by the `_moveSpeed` variable, and the result directly sets the Rigidbody2D's `linearVelocity`. This approach instantly moves the player in the desired direction at a consistent speed, with all movement handled by the physics engine.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.InputSystem;
5
6  0 references
7  public class InputManager : MonoBehaviour
8  {
9      1 reference
10     public static Vector2 Movement;
11     2 references
12     private PlayerInput _playerInput;
13     2 references
14     private InputAction _moveAction;
15
16     0 references
17     void Awake()
18     {
19         _playerInput = GetComponent<PlayerInput>();
20         _moveAction = _playerInput.actions["Move"];
21     }
22
23     0 references
24     void Update()
25     {
26         Movement = _moveAction.ReadValue<Vector2>();
27     }
28 }

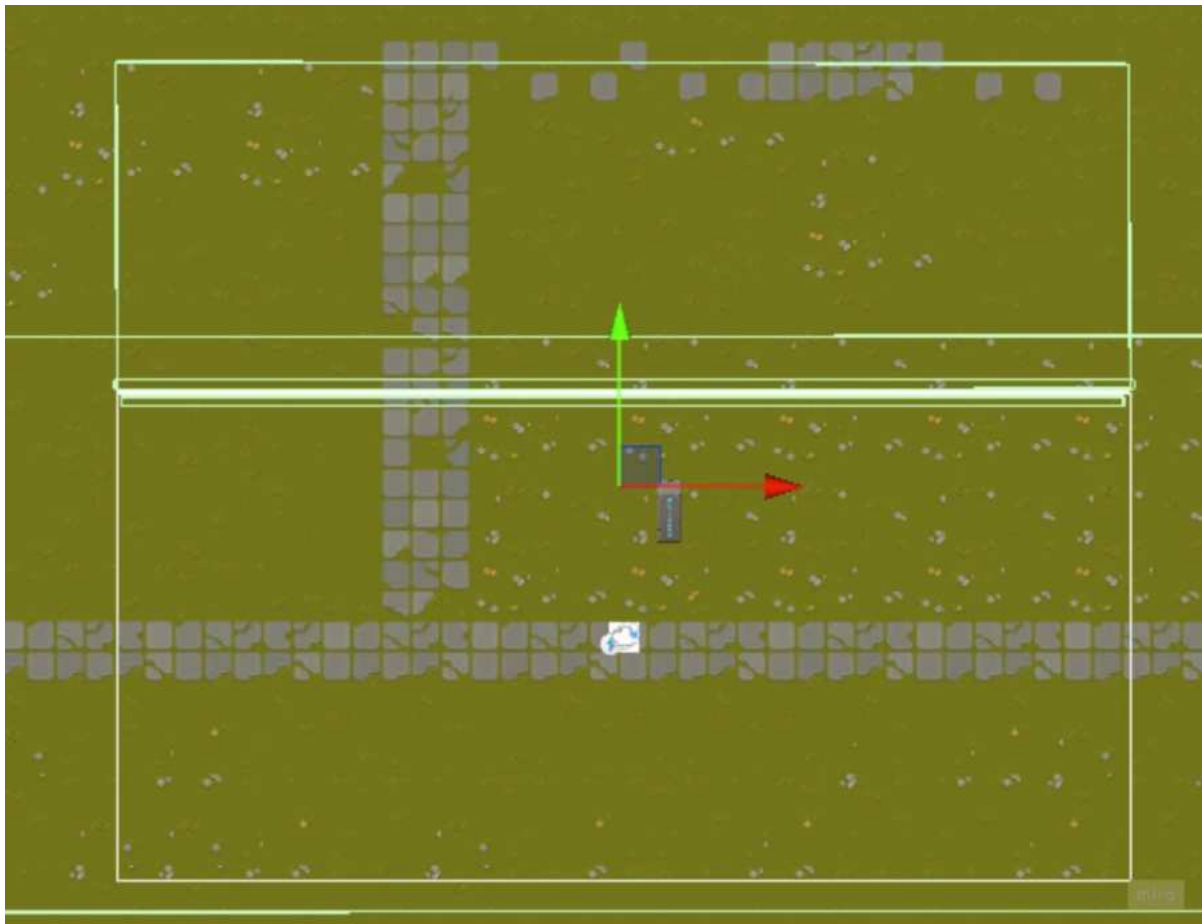
```

Figure 26 - Code snippet of Input Manager for Player Movement.

The Input Manager script (Seen in Figure 26) converts WASD key inputs into a normalized Vector2 value via Unity's input system, storing the direction in a static Movement variable. The Player Movement script accesses this value each frame, multiplying it by a speed parameter and applying the result to the player's Rigidbody. linearVelocity, creating immediate, physics driven movement.

Initial testing of the player movement script demonstrated quick and responsive control, thanks to the integration of the Unity Input System. Following the player testing, was testing the player's integration into the Tilemap that was previously added to the scene. The integration process was successful, thanks to the prior research and setup conducted during the Tilemap configuration phase.

#### 5.4.4 Goal 2 – Camera Transition System



*Figure 27 - Snippet of trigger points for Camera Transition script.*

A camera transition system (Seen in Figure 27) was required to adjust camera bounds based on the player's location within the level. Implementing this system at this stage of development would help to prevent potential complications during later stages of development, notably when developing procedurally generated environments. Unity's extensive community resources were instrumental in finding a suitable solution, which was brought into the test scene to gain a clearer understanding of its functionality.

As shown in Figure 27, the system was constructed using two polygon colliders, each serving as a camera boundary. These boundaries constrain the camera to follow the player only when within their respective zones. Additionally, two trigger points were placed, one in each boundary area, to signal the camera to switch its confining boundary. At the same time, the player's position was adjusted slightly during the transition to prevent repeated triggering or overlap between zones.



```

public class MapTransition : MonoBehaviour
{
    1 reference
    [SerializeField] PolygonCollider2D mapBoundry;
    2 references
    CinemachineConfiner confiner;
    1 reference
    [SerializeField] Direction direction;
    5 references | 1 reference | 1 reference | 1 reference | 1 reference
    enum Direction {Up, Down, Left, Right}

    0 references
    void Awake()
    {
        confiner = FindObjectOfType<CinemachineConfiner>();
    }

    0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if(collision.gameObject.CompareTag("Player"))
        {
            confiner.m_BoundingShape2D = mapBoundry;
            UpdatePlayerPosition(collision.gameObject);
        }
    }
}

```

Figure 28 - Code snippet of camera transition script.

```

1 reference
private void UpdatePlayerPosition(GameObject player)
{
    Vector3 newPos = player.transform.position;

    switch(direction)
    {
        case Direction.Up:
            newPos.y += 2;
            break;

        case Direction.Down:
            newPos.y -= 2;
            break;

        case Direction.Left:
            newPos.x += 2;
            break;

        case Direction.Right:
            newPos.x -= 2;
            break;
    }

    player.transform.position = newPos;
}

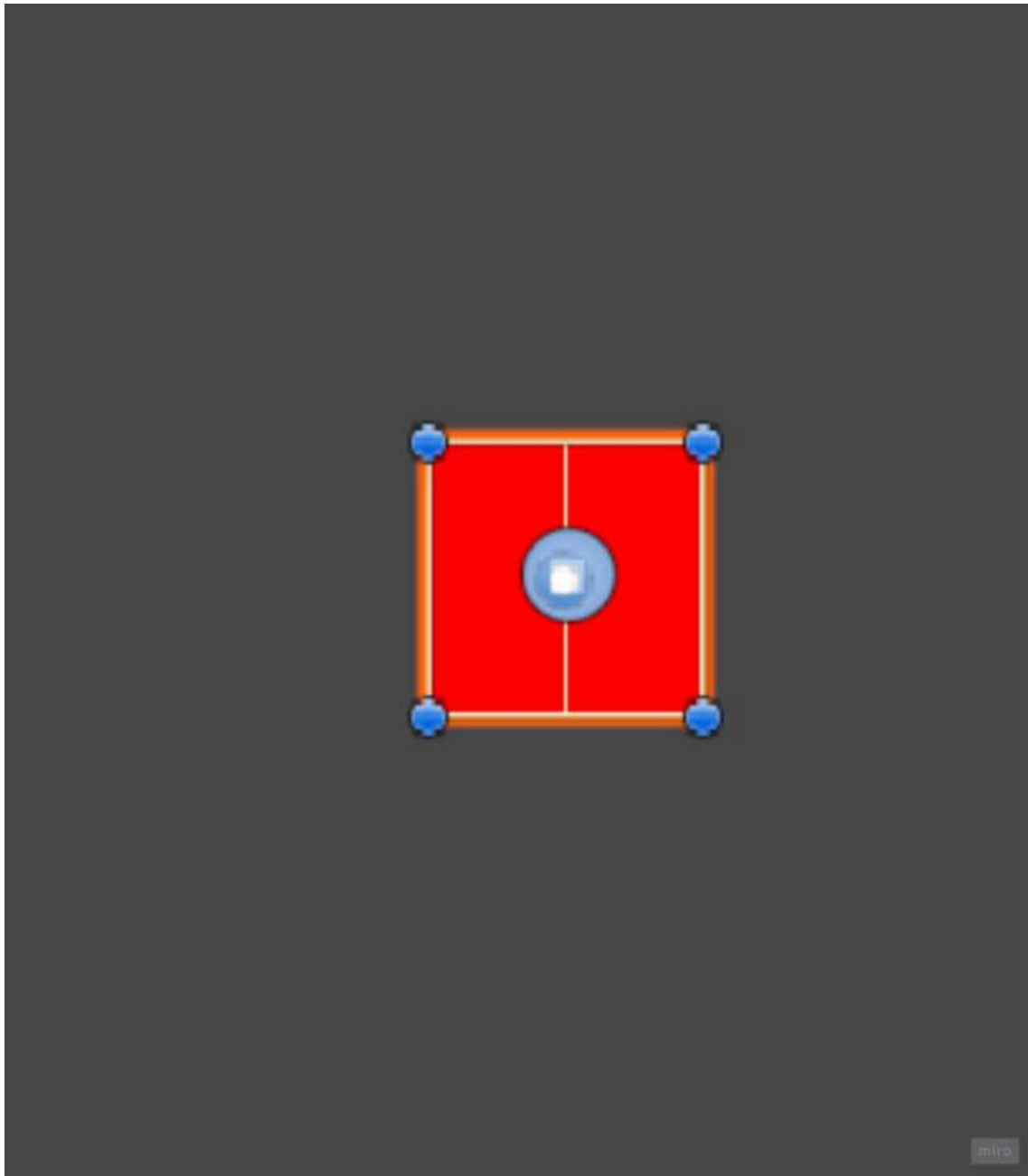
```

Figure 29 - Code snippet of camera transition script.

The Map Transition script (Seen in Figure 28 and Figure 29) manages camera boundary transitions using Cinemachine's Confiner2D system. When the player enters a trigger zone, the script updates the camera's bounding shape to a new PolygonCollider2D boundary (mapBoundary), seamlessly transitioning the camera's constrained view area. This prevents the camera from showing out-of-bounds areas while maintaining smooth movement. The script also slightly adjusts the player's position after transition (via UpdatePlayerPosition) to prevent accidental re-triggering of the zone, ensuring stable camera behavior. An enum defines transition directions (Up, Down, Left, Right) for organized level design. The system leverages Unity's trigger colliders and Cinemachine's dynamic confiner modification to create polished scene transitions.

Through testing it was found that this system would need some modifications as the trigger points that changed the camera bounds were not as reliable as originally desired, sometimes leading the player to be hidden from the camera in the wrong zone or sometimes ignoring trigger points completely and causing the camera to be stuck in the previous zone.

#### 5.4.5 Goal 3 – Enemy Pathfinding System



*Figure 30 - Snippet of test enemy game object.*

```

1  using UnityEngine;
2  using UnityEngine.AI;
3
4  public class Enemy : MonoBehaviour
5  {
6      [SerializeField] Transform target;
7      NavMeshAgent agent;
8
9      private void Awake()
10     {
11         target = GameObject.FindWithTag("Player").transform;
12     }
13
14     void Start()
15     {
16         agent = GetComponent<NavMeshAgent>();
17         agent.updateRotation = false;
18         agent.updateUpAxis = false;
19     }
20
21     void Update()
22     {
23         agent.SetDestination(target.position);
24     }
25 }
26

```

Figure 31 - Code Snippet of enemy movement AI

The Enemy script (Seen in Figure 31) implements pathfinding using Unity's NavMesh navigation with support from the NavMeshPlus GitHub repository. This solution enables nav mesh to be baked directly onto the Tilemap's collision layer, creating walkable surfaces for pathfinding calculations. The Enemy script identifies the player's transform as the target destination, while the NavMeshAgent component handles movement by continuously recalculating paths along the baked mesh. The enemy's rigidbody follows these calculated paths toward the player position, automatically navigating around obstacles. Essential 2D configuration includes disabling updateRotation and updateUpAxis to maintain proper movement alignment. This approach provides efficient pathfinding through pre-baked navigation data without requiring manual waypoint systems.

The NavMeshPlus GitHub repository was discovered while gathering the applied research during the first sprint. Through investigation into how this works it looks as though this repository works using the A\* (A-Star) pathfinding algorithm which was discussed at length in Chapter 2. Upon further investigation it was also found that the Nav Mesh component built into the Unity engine was also powered by the A\* pathfinding algorithm.

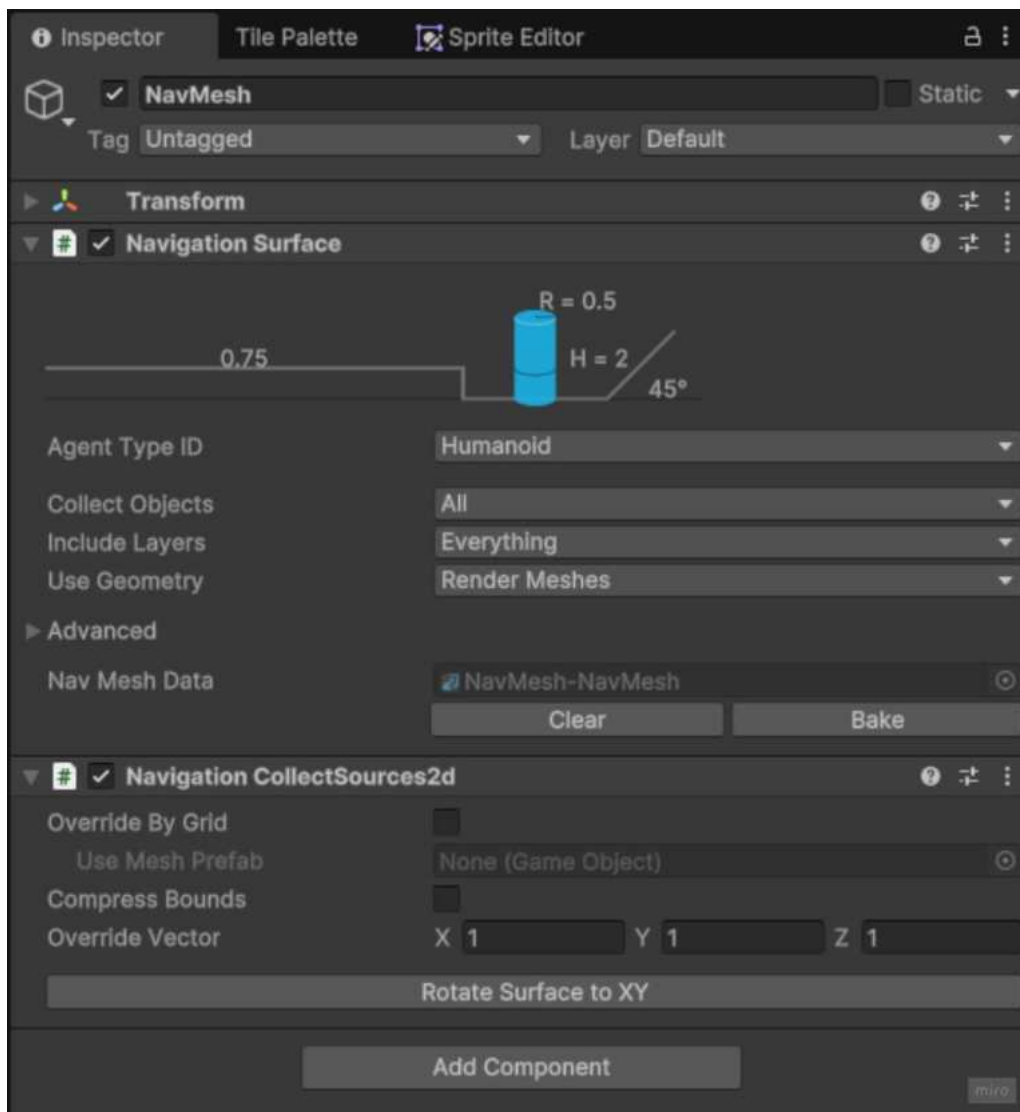


Figure 32- Snippet of nav surface script for baking object detection.

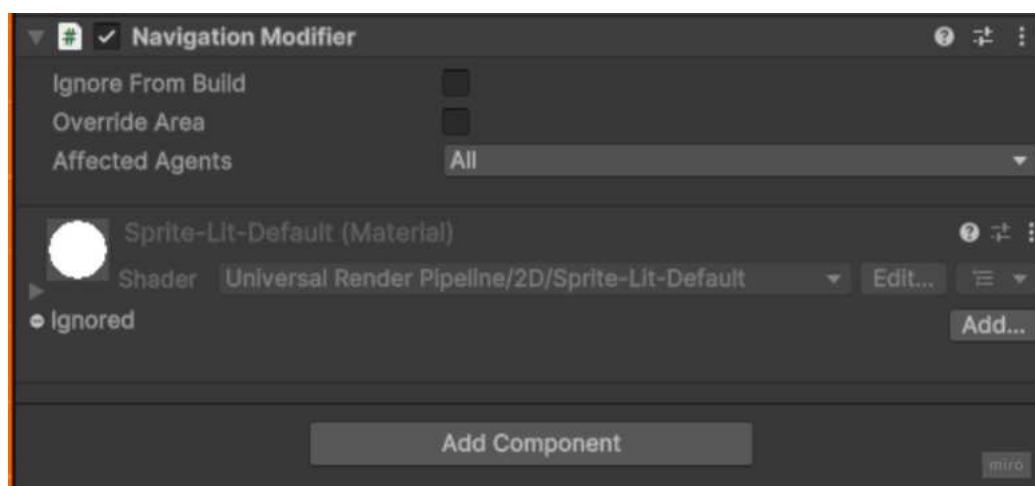


Figure 33 - Script to set collision layer as an object to detect.

## 5.4.6 Goal 4 – Player Combat System

```

using UnityEngine;
using System.Collections;

public class PlayerMovement : MonoBehaviour
{
    [SerializeField] private float _moveSpeed = 5f;

    private Vector2 _movement;
    private Rigidbody2D _rb;
    public Transform _aim;

    void Awake()
    {
        _rb = GetComponent<Rigidbody2D>();
    }

    void Update()
    {
        _movement.Set(InputManager.Movement.x, InputManager.Movement.y);
        _rb.linearVelocity = _movement * _moveSpeed;

        if (_movement.sqrMagnitude > 0.01f)
        {
            float angle = Mathf.Atan2(_movement.y, _movement.x) * Mathf.Rad2Deg;
            _aim.rotation = Quaternion.Euler(0, 0, angle + 90);
        }
    }
}

```

Figure 34 - Updated Player Movement script for Player Melee integration.

The Player Movement script (Seen in Figure 34) was updated to now include aiming functionality to create a melee combat system. A new public Transform reference (\_aim) tracks the weapon object requiring directional rotation. When movement input exceeds a minimal threshold, the script calculates a target angle using `Mathf.Atan2` with the movement vector's Y/X components, converting the result from radians to degrees. This calculated rotation is then applied to the aim object using `Quaternion.Euler`, creating smooth directional facing aiming mechanic that matches the player's movement input.

```

1  using UnityEngine;
2
3  public class PlayerMelee : MonoBehaviour
4  {
5      public GameObject Melee;
6      bool _isAttacking = false;
7      float _attackDuration = 0.3f;
8      float _attackCooldown = 0f;
9
10     void Update()
11     {
12         CheckMeleeTimer();
13
14         if(Input.GetKey(KeyCode.Mouse0))
15         {
16             onAttack();
17         }
18     }
19
20     void onAttack()
21     {
22         if (!_isAttacking)
23         {
24             Melee.SetActive(true);
25             _isAttacking = true;
26
27             // Animator goes here
28         }
29     }
30
31     void CheckMeleeTimer()
32     {
33         if (_isAttacking)
34         {
35             _attackCooldown += Time.deltaTime;
36             if (_attackCooldown >= _attackDuration)
37             {
38                 _attackCooldown = 0;
39                 _isAttacking = false;
40                 Melee.SetActive(false);
41             }
42         }
43     }
44 }

```

Figure 35 - Code snippet of Player Melee Script.

The Player Melee script (Seen in Figure 35) implements timed melee attacks through a child GameObject (Melee) that's parented to the Aim GameObject which is controlled inside the



Player Movement script, ensuring proper directional alignment. The system uses three key variables: a boolean `_isAttacking`, a 0.3 second `_attackDuration`, and a cumulative `_attackCooldown` timer. When the mouse button is pressed, the `OnAttack()` function activates the Melee GameObject which contains the player's attack box and sets the attacking state to true, while `CheckMeleeTimer()` automatically deactivates the GameObject after the attack duration expires. This creates a self-contained attack system where the melee hitbox follows the player's aim direction through its hierarchy placement, with built-in cooldown prevention through the attacking state boolean.

```

1  using UnityEngine;
2
3  public class Weapon : MonoBehaviour
4  {
5      public float _damage = 1;
6
7      private void OnTriggerEnter2D(Collider2D collision)
8      {
9          Enemy _enemy = collision.GetComponent<Enemy>();
10
11         if(_enemy != null)
12         {
13             _enemy.TakeDamage(_damage);
14         }
15     }
16 }
17

```

Figure 36 - Code snippet of Player Weapon script.

The Weapon script (Seen in Figure 36) serves as the damage-dealing component attached to the Melee GameObject. When active, it detects collisions through Unity's trigger system and applies damage to any encountered Enemy objects. The script features a modifiable `_damage` value that gets passed to enemies via their `TakeDamage()` method. Using `OnTriggerEnter2D`, it efficiently checks for Enemy components on colliding objects before executing damage calls, preventing unnecessary operations on non-enemy collisions. This creates a lightweight damage system that leverages Unity's physics callbacks, where the weapon's activation/deactivation is controlled by the Player Melee script's timing system. The script's placement on the Melee GameObject ensures damage only occurs during active attack frames while maintaining proper directional alignment through the aim system.



## 5.4.7 Goal 5 – Health and Knockback system

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Collections.Generic;
4
5  public class PlayerHealth : MonoBehaviour
6  {
7      [SerializeField] private float _health = 100;
8      private GameObject _player;
9
10     void Awake()
11     {
12         _player = GameObject.Find("Player");
13     }
14
15     void Update()
16     {
17         if (_health <= 0)
18         {
19             Debug.Log("You're dead!");
20             _player.SetActive(false);
21         }
22     }
23
24     void OnCollisionEnter2D(Collision2D collision)
25     {
26         if (collision.gameObject.CompareTag("Enemy"))
27         {
28             _health = _health - 10;
29         }
30     }
31 }
32
```

Figure 37 - Code snippet of Player Health Script.

The Player Health script (Seen in Figure 37) manages the player's survival state through a damage system and death check. Attached directly to the player prefab, it maintains a serialized health value that gets reduced by ten points whenever the player collides with objects tagged "Enemy". The script continuously monitors health in Update(), deactivating the player GameObject. The collision system uses Unity's physics callbacks for efficient damage detection without per-frame checks, creating a straightforward health management solution that integrates with enemy interactions.

```

1  using UnityEngine;
2  using UnityEngine.AI;
3
4  public class Enemy : MonoBehaviour
5  {
6      [SerializeField] Transform _target;
7
8      NavMeshAgent _agent;
9      float _health = 3f;
10
11     private void Awake()
12     {
13         _target = GameObject.FindWithTag("Player").transform;
14     }
15
16     void Start()
17     {
18         _agent = GetComponent<NavMeshAgent>();
19         _agent.updateRotation = false;
20         _agent.updateUpAxis = false;
21     }
22
23     void Update()
24     {
25         _agent.SetDestination(_target.position);
26     }
27
28     public void TakeDamage(float _damage)
29     {
30         _health -= _damage;
31
32         if (_health <= 0)
33         {
34             Destroy(gameObject);
35         }
36     }
37 }
38
39

```

Figure 38 - Updated Enemy script with health integration.

The updated Enemy script (Seen in Figure 38) now includes a health system that interacts directly with the player's Weapon script. A `_health` variable tracks the enemy's vitality, while the `TakeDamage()` method processes incoming damage from the Weapon script. When the weapon's hitbox contacts the enemy, it calls `TakeDamage()` with its `_damage` value, decrementing the enemy's health. If health reaches zero, the enemy `GameObject` is immediately removed from the scene via the `Destroy()` method. This creates a clean interaction where the Weapon script detects collisions and calls the damage function, and the Enemy

script handles its own health state and destruction with no additional communication needed between the systems beyond the initial damage call.

```

void Update()
{
    if (!_knocked)
    {
        HandleMovement();
        HandleAim();
    }
    else
    {
        ApplyKnockbackDeceleration();
    }
}

private void HandleMovement()
{
    // Movement logic
    _movement.Set(InputManager.Movement.x, InputManager.Movement.y);
    _rb.linearVelocity = _movement * _moveSpeed;
}

private void HandleAim()
{
    // Get mouse position
    Vector3 mouseWorldPosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);

    // Calculate direction from player to mouse
    Vector2 aimDirection = (mouseWorldPosition - _center.position).normalized;

    // Determine the direction
    if (Mathf.Abs(aimDirection.x) > Mathf.Abs(aimDirection.y))
    {
        // Horizontal direction
        aimDirection = new Vector2(Mathf.Sign(aimDirection.x), 0);
    }
    else
    {
        // Vertical direction
        aimDirection = new Vector2(0, Mathf.Sign(aimDirection.y));
    }

    // Calculate the angle and set the rotation
    float angle = Mathf.Atan2(aimDirection.y, aimDirection.x) * Mathf.Rad2Deg;
    _aim.rotation = Quaternion.Euler(0, 0, angle + 90);
}

```

Figure 39 - Code snippet of updated player movement script for new aim mechanic and knockback integration.

The refactored Player Movement script (Seen in Figure 39) now implements three key systems in a state-driven architecture. When in the UnKnocked state, the script processes movement through the existing physics-based velocity system while introducing a new mouse-driven aiming mechanic. This aiming system converts screen coordinates into game world space,

calculating the closest appropriate cardinal direction by comparing the axis dominance, prioritizing horizontal or vertical based on closest input direction, and applies proper 2D rotation with a 90-degree offset.

```
private void ApplyKnockbackDeceleration()
{
    // Gradually reduce velocity during knockback
    _rb.linearVelocity = Vector2.Lerp(_rb.linearVelocity, Vector2.zero, Time.deltaTime * 3);
}

public void Knockback(Transform t)
{
    var direction = _center.position - t.position;
    _knocked = true;
    _rb.linearVelocity = direction.normalized * _knockbackVel;
    StartCoroutine(Unknocked());
}

private IEnumerator Unknocked()
{
    yield return new WaitForSeconds(_knockedTime);
    _knocked = false;
}
```

Figure 40 - Updated Player movement script for knockback integration.

When knockback is triggered, the script switches to the `_knocked` state where movement input is disabled and physics take over, applying force away from the impact source at a configured velocity, then smoothly decelerating using the `Vector2.Lerp` method until the timed recovery period ends with the `UnKnocked()` coroutine. The state management ensures clean transitions between these modes, with the `_knocked` boolean preventing movement/aiming during recovery while maintaining all existing physics interactions. This creates responsive combat feedback while preserving the original movement feel, with the cardinal-direction aiming complementing melee systems by providing clear directional intent. All of this can be seen in Figure 40.

```

public void Knockback(Transform t)
{
    Vector3 _direction = (_center.position - t.position).normalized;

    _knocked = true;
    _agent.isStopped = true;

    _agent.velocity = _direction * _knockbackVel;

    StartCoroutine(UnKnocked());
}

private IEnumerator UnKnocked()
{
    yield return new WaitForSeconds(_knockTime);

    _knocked = false;
    _agent.isStopped = false;
}
}

```

Figure 41 - Updated Enemy script for knockback integration.

The enemy and player both use a knockback system that activates when they collide. When the player weapon hitbox collides with an enemy, the enemy's `Knockback()` method figures out the direction from the player's centre, then pushes the enemy back using that direction and a force value. It also turns off the enemy's pathfinding for a short time so the knockback works properly. After a short delay, the enemy goes back to normal behavior. The enemy knockback function can be seen in Figure 41.

In the same way, when an enemy hits the player, it triggers the player's `Knockback()` method, which works in a similar way. Both systems use the same idea. Knockback based on direction, a short delay, and temporary changes to how they move. The player's knockback works with their movement system, while the enemy's knockback pauses their pathfinding. This makes the combat feel fair and reactive on both sides.

## 5.5 Sprint 4

### 5.5.1 Goals

- Create a room prefab.
- Add a design to the room.
- Create a procedural generation script.
- Update enemy nav mesh to work with map generation.
- Update camera transition script to work with dynamically generated map.

## Dungeon Scribbles

- Add a pause menu.



## 5.5.2 Goal 1 – Room Prefab

```
1  using UnityEngine;
2
3  public class Room : MonoBehaviour
4  {
5      [SerializeField] GameObject _topDoor;
6      [SerializeField] GameObject _bottomDoor;
7      [SerializeField] GameObject _leftDoor;
8      [SerializeField] GameObject _rightDoor;
9
10     public Vector2Int RoomIndex { get; set; }
11
12     public void OpenDoor(Vector2Int direction)
13     {
14         if(direction == Vector2Int.up)
15         {
16             _topDoor.SetActive(true);
17         }
18
19         if(direction == Vector2Int.down)
20         {
21             _bottomDoor.SetActive(true);
22         }
23
24         if(direction == Vector2Int.left)
25         {
26             _leftDoor.SetActive(true);
27         }
28
29         if(direction == Vector2Int.right)
30         {
31             _rightDoor.SetActive(true);
32         }
33     }
34 }
35
```

Figure 42 - Code Snippet of Room script.

The Room script (Seen in Figure 42) serves as a modular door control system for the procedural room generation scripts, attached to each room prefab. It contains four serialized GameObject references representing each cardinal exit point. The public `OpenDoor()` method accepts a `Vector2Int` direction parameter and activates the corresponding door GameObject when called. This activation system integrates with the Room Manager script which determines neighbouring rooms, only doors leading to valid adjacent rooms will be triggered via this method. The script also includes a `Vector2Int` Room Index property for grid-based room tracking in the map generation system. This creates a clean system where the Room Manager script handles level generation logic and each Room instance manages its own door states.

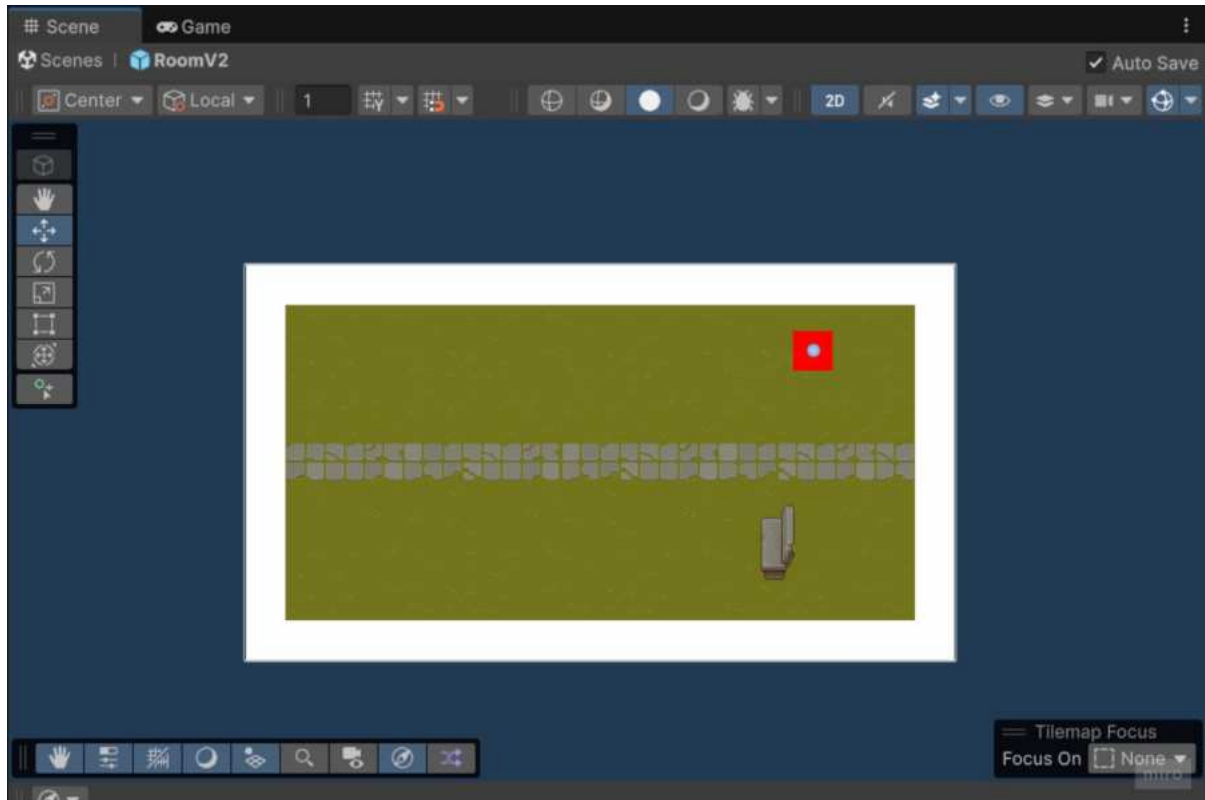


Figure 43 - Screenshot of room prefab.



### 5.5.3 Goal 2 – Procedural Map Generation

```

C: > Users > downe > Documents > Thesis > Thesis > Assets > RoomManager.cs
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class RoomManager : MonoBehaviour
6  {
7      [SerializeField] GameObject _roomPrefab;
8      [SerializeField] private int _maxRooms = 15;
9      [SerializeField] private int _minRooms = 10;
10
11     int _roomWidth = 20;
12     int _roomHeight = 12;
13
14     [SerializeField] int _gridSizeX = 15;
15     [SerializeField] int _gridSizeY = 15;
16
17     private List<GameObject> _roomObjects = new List<GameObject>();
18
19     private Queue<Vector2Int> _roomQueue = new Queue<Vector2Int>();
20
21     private int[,] _roomGrid;
22
23     private bool generationComplete = false;
24
25     private int _roomCount;
26
27     private void Start()
28     {
29         _roomGrid = new int[_gridSizeX, _gridSizeY];
30         _roomQueue = new Queue<Vector2Int>();
31
32         Vector2Int initialRoomIndex = new Vector2Int(_gridSizeX / 2, _gridSizeY / 2);
33         StartRoomGenerationFromRoom(initialRoomIndex);
34     }
35

```

Figure 44 - Code snippet one of Room generation script.

The Room Manager's Start() function in the (Seen in Figure 44) sets up the main systems needed for procedural generation. It first creates a 2D grid called `_roomGrid` to keep track of where rooms are placed. Then, it sets up a queue called `_roomQueue` to manage the order in which rooms are generated. The starting room's position is calculated and placed in the centre of the grid. Finally, it calls `StartRoomGenerationFromRoom()` to begin generating the dungeon, making sure it always starts from a consistent and expected location.

```

36     private void Update()
37     {
38         if (_roomQueue.Count > 0 && _roomCount < _maxRooms && !generationComplete)
39         {
40             Vector2Int roomIndex = _roomQueue.Dequeue();
41             int gridX = roomIndex.x;
42             int gridY = roomIndex.y;
43
44             TryGenerateRoom(new Vector2Int(gridX - 1, gridY));
45             TryGenerateRoom(new Vector2Int(gridX + 1, gridY));
46             TryGenerateRoom(new Vector2Int(gridX, gridY + 1));
47             TryGenerateRoom(new Vector2Int(gridX, gridY - 1));
48         } else if (_roomCount < _minRooms)
49         {
50             Debug.Log("Invalid Room Count, Regenerating...");
51             RegenerateRooms();
52         }
53         else if (!generationComplete)
54         {
55             Debug.Log($"Generation Complete, {_roomCount} rooms created");
56             generationComplete = true;
57         }
58     }
59

```

Figure 45 - Code snippet of Room generation script.

The Room Manager's Update() function (Seen in Figure 45) runs the room generation process every frame. It checks the queue of room indices and tries to spawn new rooms in all four cardinal directions. If the number of rooms drops below the minimum limit (\_minRooms), it calls the RegenerateRooms() function to restart the process. When the maximum number of rooms (\_maxRooms) is reached, the generation ends and a message is logged to show it's complete.

```

59     private void StartRoomGenerationFromRoom(Vector2Int roomIndex)
60     {
61         _roomQueue.Enqueue(roomIndex);
62         int x = roomIndex.x;
63         int y = roomIndex.y;
64         _roomGrid[x, y] = 1;
65         _roomCount++;
66
67         var initialRoom = Instantiate(_roomPrefab, GetPositionFromGridIndex(roomIndex), Quaternion.identity);
68         initialRoom.name = $"Room-{_roomCount}";
69         initialRoom.GetComponent<Room>().RoomIndex = roomIndex;
70         _roomObjects.Add(initialRoom);
71     }
72

```

Figure 46 - Code snippet of Room generation script.

The Room Manager's StartRoomGenerationFromRoom() function (Seen in Figure 46) begins the generation process by placing the first room into the queue and marking its spot on the grid as taken. It then creates the starting room, gives it a unique name and index number, and stores it in the \_roomObjects array to keep track of it. This sets up a clear starting point for the rest of the dungeon to build from.

```

74     private bool TryGenerateRoom(Vector2Int roomIndex)
75     {
76         int x = roomIndex.x;
77         int y = roomIndex.y;
78
79         if (_roomCount >= _maxRooms ) return false;
80
81         if (Random.value < 0.5f && roomIndex != Vector2Int.zero) return false;
82
83         if (CountAdjacentRooms(roomIndex) > 1) return false;
84
85         _roomQueue.Enqueue(roomIndex);
86         _roomGrid[x, y] = 1;
87         _roomCount++;
88
89         var newRoom = Instantiate(_roomPrefab, GetPositionFromGridIndex(roomIndex), Quaternion.identity);
90         newRoom.GetComponent<Room>().RoomIndex = roomIndex;
91         newRoom.name = $"Room-{_roomCount}";
92         _roomObjects.Add(newRoom);
93
94         OpenDoors(newRoom, x, y);
95
96         return true;
97     }
98

```

Figure 47 - Code snippet of Room generation script.

The Room Manager's TryGenerateRoom() function (Seen in Figure 47) handles procedural room generation by validating potential new rooms through four checks. The maximum room limit, random 50% chance, adjacent room density, and grid availability. If valid all of these checks are valid, it updates the grid/queue, instantiates the prefab with proper positioning/naming, and connects doors to neighbours via the OpenDoors() function. This ensures balanced dungeon layouts while maintaining generation rules.

```

99     // Clear all rooms and try again
100     private void RegenerateRooms()
101     {
102         _roomObjects.ForEach(Destroy);
103         _roomObjects.Clear();
104         _roomGrid = new int[_gridSizeX, _gridSizeY];
105         _roomQueue.Clear();
106         _roomCount = 0;
107         generationComplete = false;
108
109         Vector2Int initialRoomIndex = new Vector2Int(_gridSizeX / 2, _gridSizeY / 2);
110         StartRoomGenerationFromRoom(initialRoomIndex);
111     }
112

```

Figure 48 - Code snippet of Room generation script.

The Room Manager's RegenerateRooms() functions (Seen in Figure 48) acts as a safeguard in case of room generation failure, such as when not enough rooms are placed. It clears everything by deleting all existing rooms, resetting the grid and queue, and setting all counters back to zero. It then starts the generation again from the centre point. This ensures that the game always creates a working dungeon layout, even if something goes wrong with the first attempt.

```

113 void OpenDoors(GameObject room, int x, int y)
114 {
115     Room newRoomScript = room.GetComponent<Room>();
116
117     // Neighbours
118     Room leftRoomScript = GetRoomScriptAt(new Vector2Int(x - 1, y));
119     Room rightRoomScript = GetRoomScriptAt(new Vector2Int(x + 1, y));
120     Room topRoomScript = GetRoomScriptAt(new Vector2Int(x, y + 1));
121     Room bottomRoomScript = GetRoomScriptAt(new Vector2Int(x, y - 1));
122
123     // Determine which doors to open based on the direction
124     if (x > 0 && _roomGrid[x - 1, y] != 0)
125     {
126         // Neighbour Left
127         newRoomScript.OpenDoor(Vector2Int.left);
128         leftRoomScript.OpenDoor(Vector2Int.right);
129     }
130     if (x < _gridSizeX - 1 && _roomGrid[x + 1, y] != 0)
131     {
132         // Neighbour Right
133         newRoomScript.OpenDoor(Vector2Int.right);
134         rightRoomScript.OpenDoor(Vector2Int.left);
135     }
136     if (y > 0 && _roomGrid[x, y - 1] != 0)
137     {
138         // Neighbour below
139         newRoomScript.OpenDoor(Vector2Int.down);
140         bottomRoomScript.OpenDoor(Vector2Int.up);
141     }
142     if (y < _gridSizeY - 1 && _roomGrid[x, y + 1] != 0)
143     {
144         // Neighbour above
145         newRoomScript.OpenDoor(Vector2Int.up);
146         topRoomScript.OpenDoor(Vector2Int.down);
147     }
148 }

```

Figure 49 - Code snippet of Room generation script.

The Room Manager's `OpenDoors()` function (Seen in Figure 49) is used to connect rooms together. It checks each cardinal direction to see if there is a neighbouring room in the `_roomGrid`. If there is, it calls the `OpenDoor()` function for both rooms, making sure that the doors line up properly. For example, one room's right door connects to the neighbour's left door. This keeps the layout easy to move through and follows the rules set by the procedural generation system.

```

150 Room GetRoomScriptAt(Vector2Int index)
151 {
152     GameObject roomObject = _roomObjects.Find(r => r.GetComponent<Room>().RoomIndex == index);
153     if (roomObject != null) return roomObject.GetComponent<Room>();
154     return null;
155 }
156
157 private int CountAdjacentRooms(Vector2Int roomIndex)
158 {
159     int x = roomIndex.x;
160     int y = roomIndex.y;
161     int count = 0;
162
163     if (x > 0 && _roomGrid[x - 1, y] != 0) count++; // Left Neighbour
164     if (x < _gridSizeX - 1 && _roomGrid[x + 1, y] != 0) count++; // Right Neighbour
165     if (y > 0 && _roomGrid[x, y - 1] != 0) count++; // Neighbour Below
166     if (y < _gridSizeY - 1 && _roomGrid[x, y + 1] != 0) count++; // Neighbour Above
167
168     return count;
169 }
170
171 private Vector3 GetPositionFromGridIndex(Vector2Int gridIndex)
172 {
173     int gridX = gridIndex.x;
174     int gridY = gridIndex.y;
175
176     return new Vector3(_roomWidth * (gridX - _gridSizeX / 2), _roomHeight * (gridY - _gridSizeY / 2));
177 }
178

```

Figure 50 - Code snippet of Room generation script.

The Room Manager's `GetRoomScriptAt()` function (Seen in Figure 50) is a helpful tool that finds a room's Room script by looking through `_roomObjects` for a matching room index. It's mainly used by the `OpenDoors()` function to let rooms communicate with each other—such as keeping door states in sync.

The `CountAdjacentRooms()` function (Seen in Figure 50) checks how many rooms are directly next to the current one (up, down, left, or right) by looking at `_roomGrid`. This is important for the `TryGenerateRoom()` method, as it helps avoid placing too many rooms too close together and keeps the dungeon layout clear and balanced.

The `GetPositionFromGridIndex()` (Seen in Figure 50) function takes grid coordinates like `[3,2]` and turns them into actual world-space positions using the set room width and height (`_roomWidth` and `_roomHeight`). It also centres the whole dungeon by adjusting the positions based on the middle of the grid.



```

179     private void OnDrawGizmos()
180     {
181         Color gizmoColor = new Color(0, 1, 1, 0.05f);
182         Gizmos.color = gizmoColor;
183
184         for (int x = 0; x < _gridSizeX; x++)
185         {
186             for (int y = 0; y < _gridSizeY; y++)
187             {
188                 Vector3 position = GetPositionFromGridIndex(new Vector2Int(x, y));
189                 Gizmos.DrawWireCube(position, new Vector3(_roomWidth, _roomHeight, 1));
190             }
191         }
192     }
193 }
194

```

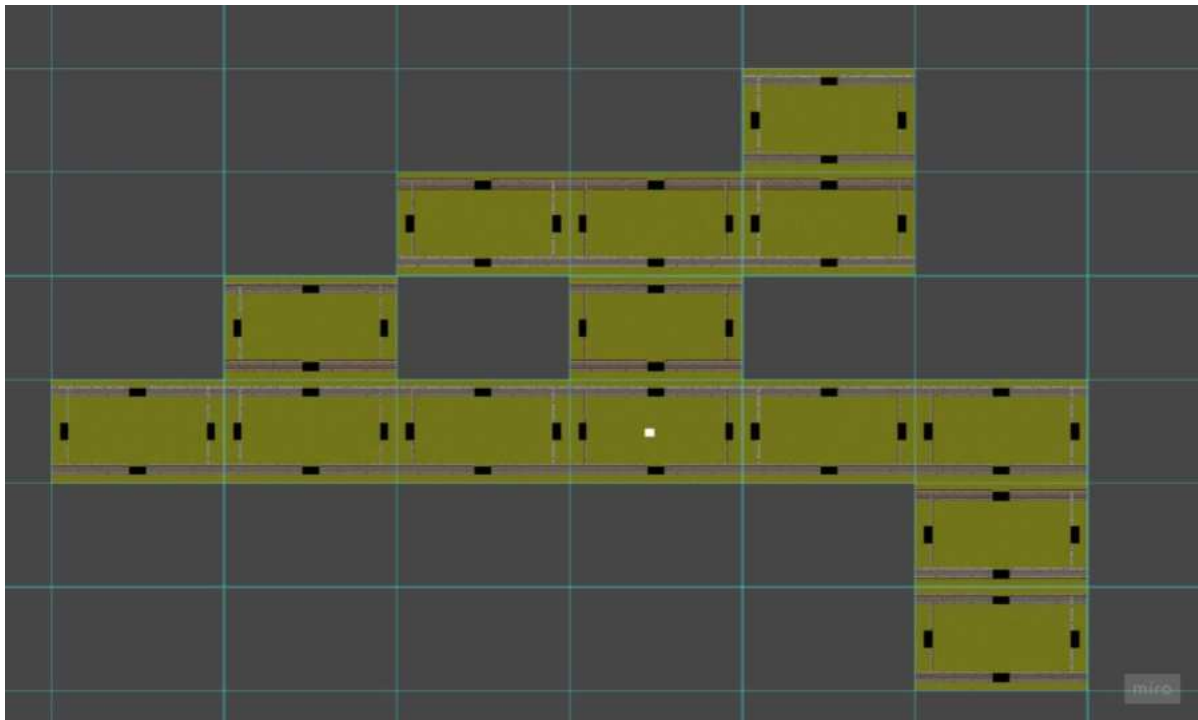
Figure 51 - Code snippet of Room generation script.

The OnDrawGizmos() function (Seen in Figure 51) is a helpful debug tool used in the Unity Editor. It shows the generation grid by drawing see-through cyan cubes at each possible room location. This makes it easier to check the grid size and room placement while building and testing the game.



Figure 52 - Snippet of Room design iteration.

Integrating designs into the procedural generation environment required a few different iterations. Initial attempts involved layering designs over the grid system (Seen in Figure 52), but this approach proved inefficient due to the constantly regenerating map layout. The dynamic nature of door GameObjects also presented additional challenges with this approach, as pre-made designs couldn't accommodate their changing positions. This idea was ultimately scrapped, but can be seen demonstrated by the results shown in Figure 52.



*Figure 53 - Room generation script test.*

The final approach adopted a more streamlined solution by designing directly within the room prefab (Seen in Figure 53). This method significantly reduced design time, as a single template could be replicated across all room instances while maintaining consistency. Testing confirmed that the Tilemap collision system continued to function correctly with this implementation. The dynamic door system challenge was addressed through a neighbour-state detection mechanism, where door GameObjects would be activated or deactivated based on adjacent room connections. However, full implementation and testing of this door system occurred during later stages of project development.

## 5.5.4 Goal 3 – Nav Mesh Integration

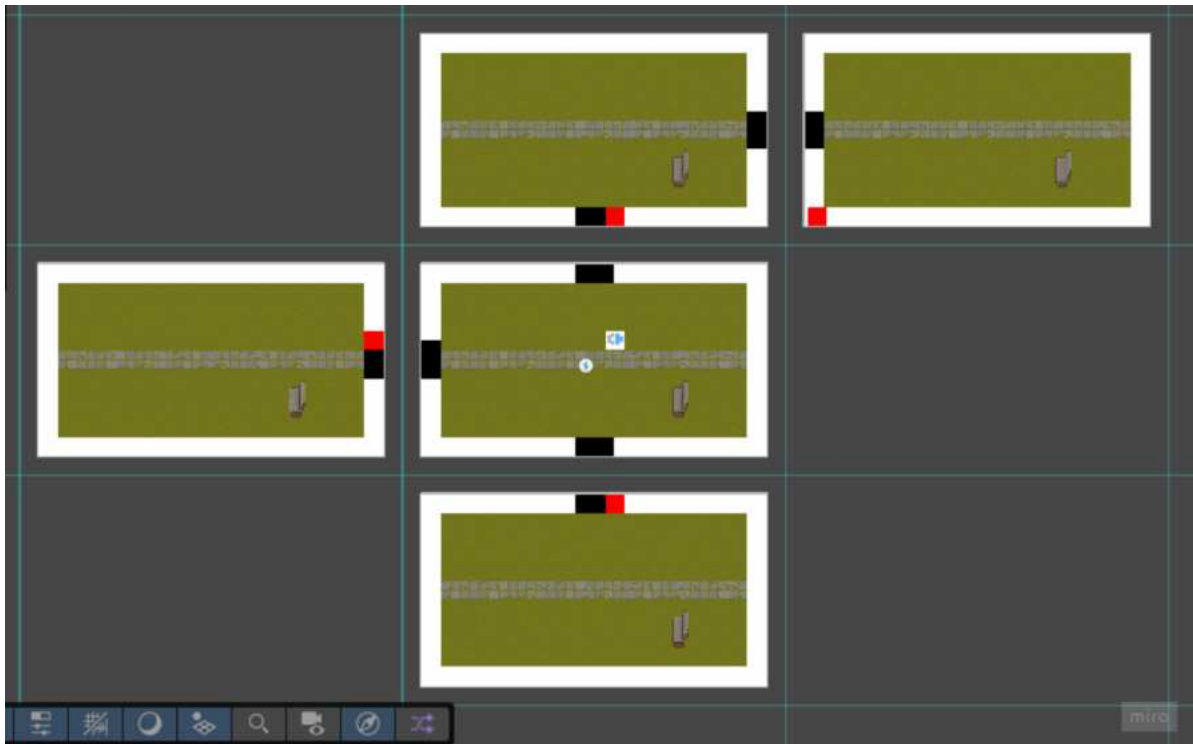


Figure 54 - Snippet of Enemy nav mesh bug.

```

21 void Start()
22 {
23     _agent = GetComponent<NavMeshAgent>();
24     _agent.updateRotation = false;
25     _agent.updateUpAxis = false;
26 }
27
28 void Update()
29 {
30     if (!_knocked && _target != null && IsPlayerOnSameNavMesh())
31     {
32         _agent.SetDestination(_target.position);
33     }
34     else
35     {
36         _agent.ResetPath(); // Stop moving if the player is not reachable
37     }
38 }
39
40 private bool IsPlayerOnSameNavMesh()
41 {
42     if (_target == null) return false;
43
44     NavMeshPath path = new NavMeshPath();
45     bool hasPath = _agent.CalculatePath(_target.position, path);
46     return hasPath && path.status == NavMeshPathStatus.PathComplete;
47 }
48

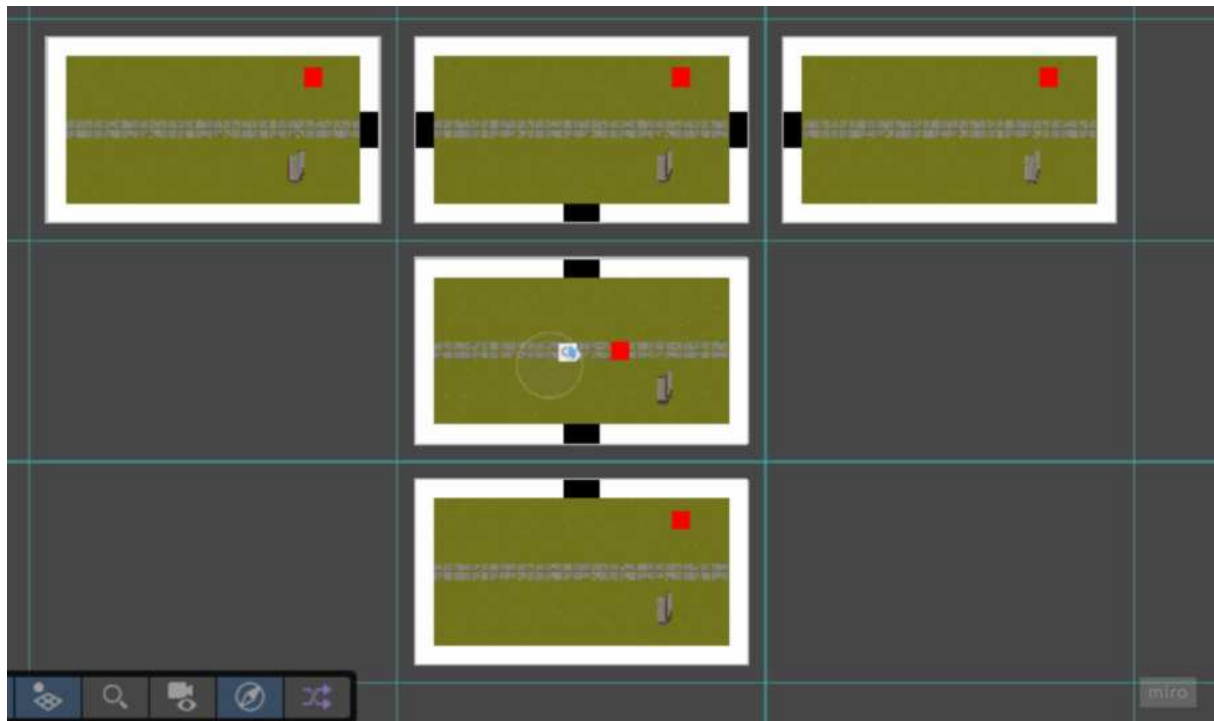
```

Figure 55 - Update to enemy movement script to fix bug.



The enemy script (Seen in Figure 55) was updated with a NavMesh validation mechanic to address pathfinding issues that occurred when enemies tracked players across rooms the player wasn't in. The key addition was the `IsPlayerOnSameNavMesh()` method which performs three critical checks.

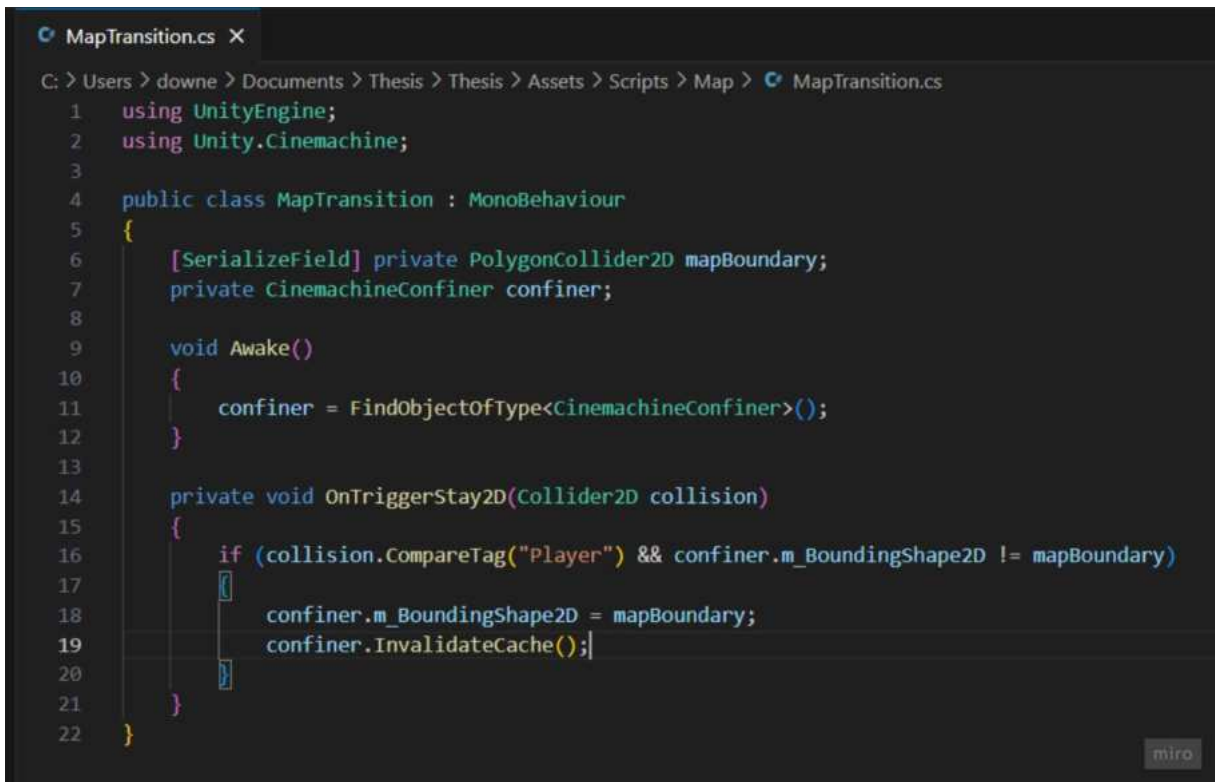
Firstly, it verifies that the target exists. Secondly, It calculates a potential path using the `CalculatePath()` method, and finally, it confirms the path is fully traversable. This validation stops the `SetDestination()` call in the `Update()` if these conditions aren't met, preventing movement attempts when the player is in disconnected rooms.



*Figure 56 - Demonstration of bug fix in action.*

The system now automatically resets the current path when either the player becomes unreachable or the enemy is in a knocked-back state. These changes directly address the original issue where enemies would cluster near room transitions, blocking player movement, while maintaining the existing knockback system's functionality. The solution pairs with the prefab modification where rooms now bake their NavMesh before generation, ensuring proper pathfinding segmentation between rooms. Results can be seen in Figure 56.

### 5.5.5 Goal 4 – Updated Camera Transition



```

1  using UnityEngine;
2  using Unity.Cinemachine;
3
4  public class MapTransition : MonoBehaviour
5  {
6      [SerializeField] private PolygonCollider2D mapBoundary;
7      private CinemachineConfiner confiner;
8
9      void Awake()
10     {
11         confiner = FindObjectOfType<CinemachineConfiner>();
12     }
13
14     private void OnTriggerStay2D(Collider2D collision)
15     {
16         if (collision.CompareTag("Player") && confiner.m_BoundingShape2D != mapBoundary)
17         {
18             confiner.m_BoundingShape2D = mapBoundary;
19             confiner.InvalidateCache();
20         }
21     }
22 }

```

Figure 57 - Updated camera transition logic.

The reworked Map Transition script (Seen in Figure 57) implements a dynamic camera boundary system that automatically adapts to procedurally generated game world. Each room prefab now contains its own PolygonCollider2D boundary, eliminating the need for static trigger points. When the player enters any room area, the OnTriggerStay2D callback continuously checks if the CinemachineConfiner's bounding shape needs updating. Upon detection, it immediately switches to the new room's boundary and invalidates the camera's cache to ensure smooth transitions. This solution provides several key improvements, firstly, it reduces scene complexity by removing dedicated trigger objects, instead using the room colliders themselves as activation zones. Secondly, it maintains performance efficiency by only executing boundary checks when the player is actively crossing room thresholds. Finally, the system now works seamlessly with procedurally generated layouts since each room instance carries its own preconfigured boundary data. The cache invalidation ensures proper camera recalculation when switching between differently shaped rooms, preventing visual glitches during transitions.

## 5.5.6 Goal 5 – Handle Aim Update

```
17
18 void Awake()
19 {
20     _rb = GetComponent<Rigidbody2D>();
21     _playerMelee = FindObjectOfType<PlayerMelee>();
22 }
23
24 void Update()
25 {
26     if (!_knocked)
27     {
28         HandleMovement();
29
30         // Prevent aiming while attack cooldown is active
31         if (_playerMelee._cooldownTimer <= 0f)
32         {
33             HandleAim();
34         }
35     }
36     else
37     {
38         ApplyKnockbackDeceleration();
39     }
40 }
```

Figure 58 - Update to player movement script.

The Player Movement script was updated to integrate with the melee combat system by adding an aiming restriction during the player's attack (Seen in Figure 58). The modification introduces a reference to the Player Melee script in `Awake()`, then checks the `_cooldownTimer` value within the Player Melee script before processing aim updates. This prevents the `HandleAim()` function from executing while an attack is in progress, effectively locking the attack direction throughout the entire melee animation cycle. The change addresses two key issues, firstly, it maintains combat consistency by preventing mid-attack direction changes that could create visual or gameplay discrepancies. Secondly, it establishes proper animation system integration by ensuring attack directions remain stable throughout the animation timeline. The knockback system remains unaffected, as it operates independently of both the aiming and melee cooldown systems.

## 5.5.7 Goal 6 – Enemy integration

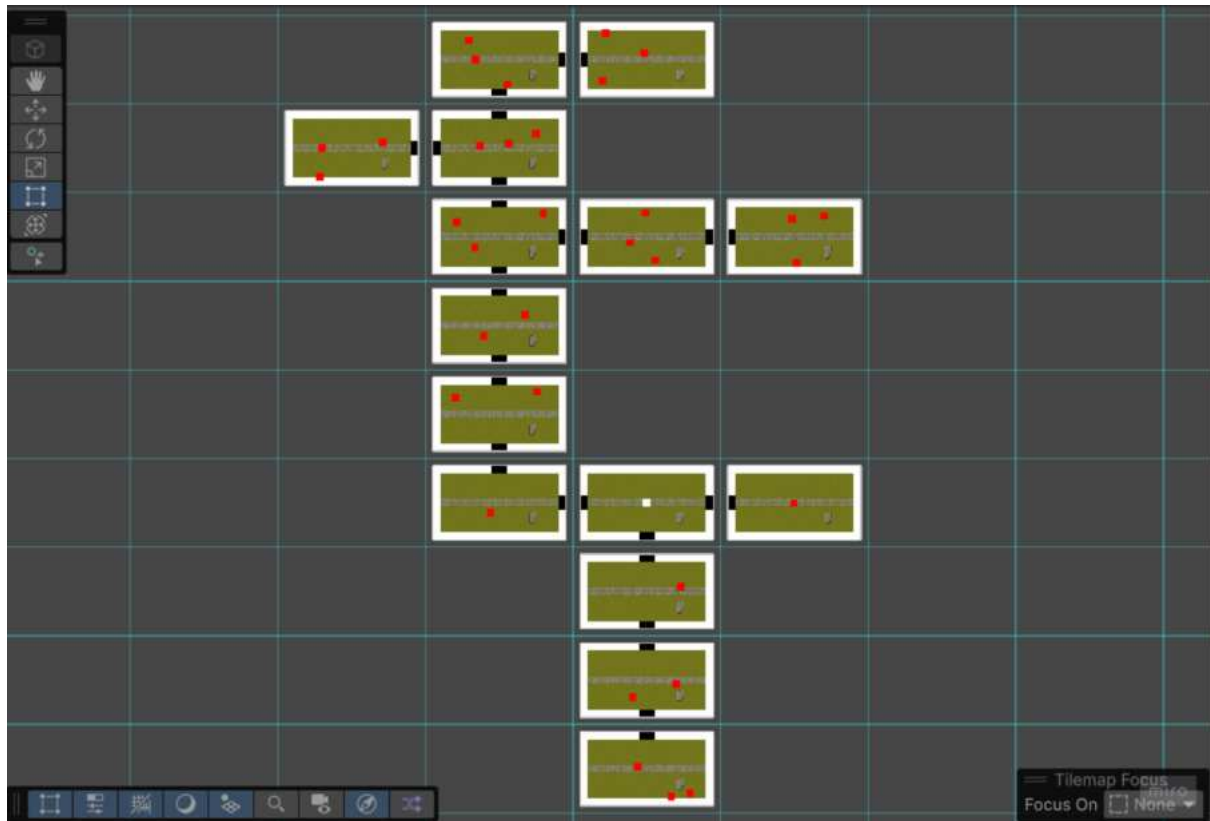


Figure 59 - Snippet of enemy integration into procedural generation.

```

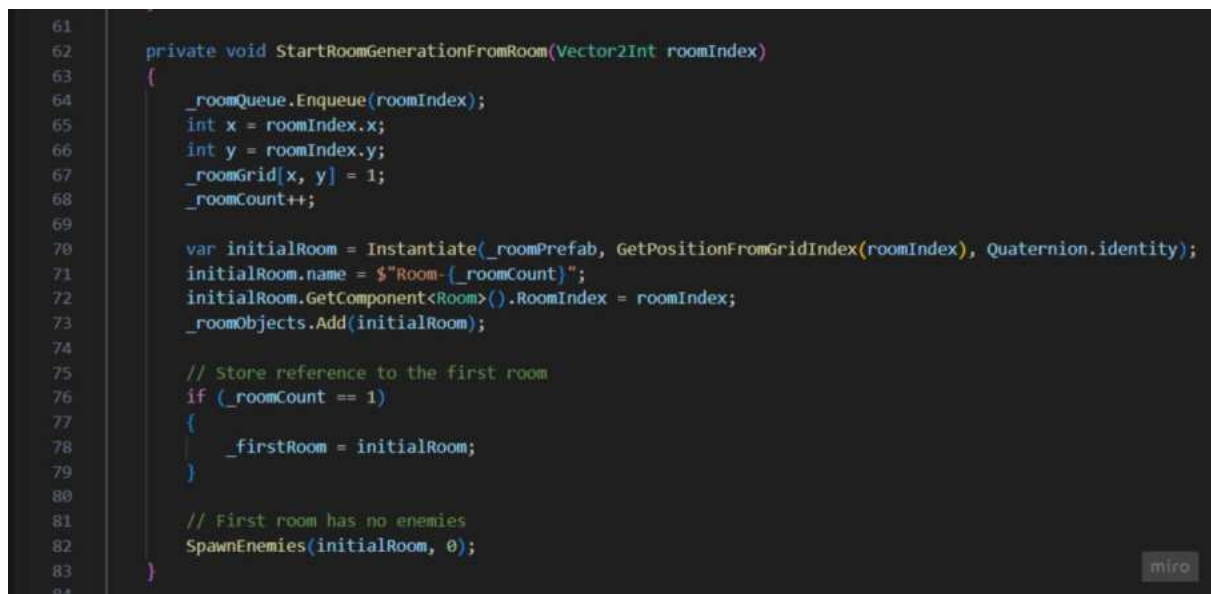
102 // Function to determine how many enemies should spawn per room
103 private int GetEnemyCountForRoom(int roomNumber)
104 {
105     if (roomNumber == 1) return 0; // First room has no enemies
106     if (roomNumber <= 4) return 1; // Rooms 2-4 have 1 enemy
107     if (roomNumber <= 7) return 2; // Rooms 5-7 have 2 enemies
108     return 3; // Rooms 8+ have 3 enemies
109 }
110
111 private void SpawnEnemies(GameObject room, int enemyCount)
112 {
113     if (_enemyPrefab == null) return;
114
115     for (int i = 0; i < enemyCount; i++)
116     {
117         Vector3 spawnPosition = GetRandomPositionInRoom(room.transform.position);
118         GameObject enemy = Instantiate(_enemyPrefab, spawnPosition, Quaternion.identity, room.transform);
119
120         // Ensure the enemy is on the correct sorting layer
121         SpriteRenderer enemyRenderer = enemy.GetComponent<SpriteRenderer>();
122         if (enemyRenderer != null)
123         {
124             enemyRenderer.sortingLayerName = "Player";
125         }
126     }
127 }
128
129 // Get a random position within the room bounds
130 private Vector3 GetRandomPositionInRoom(Vector3 roomPosition)
131 {
132     float offsetX = Random.Range(-_roomWidth / 3f, _roomWidth / 3f);
133     float offsetY = Random.Range(-_roomHeight / 3f, _roomHeight / 3f);
134     return new Vector3(roomPosition.x + offsetX, roomPosition.y + offsetY, 0);
135 }

```

Figure 60 - Code snippet of enemy integration.

The Room Manager script (Seen in Figure 60) was extended with three functions to manage procedural enemy placement. The `GetEnemyCountForRoom()` method implements a progressive difficulty curve by varying spawn counts based on room generation order, starting with 0 enemies in the first room, scaling up to 3 enemies in later rooms. The `SpawnEnemies()` function handles instantiation, creating each enemy at randomized positions within room boundaries while parenting them to their respective rooms for organizational clarity. It includes validation for the enemy prefab reference and automatically configures sprite rendering settings to ensure proper visual layering. Position randomization is managed by `GetRandomPositionInRoom()`, which calculates spawn points within the central area of each room using the predefined room dimensions. This system creates controlled enemy distribution that maintains gameplay balance while working seamlessly with the procedural generation pipeline.

A bug was then identified in the room regeneration system where enemies would incorrectly spawn in the starting room during regeneration cycles. The issue stemmed from the original implementation only enforcing the "no enemies" rule during initial generation. During regeneration, the system would recreate all rooms, including the starting room, without reapplying this rule. This inconsistency meant the starting room remained enemy-free only on the first generation attempt, disrupting the intended difficulty progression where early rooms should be safer. The bug was particularly noticeable during failed generation attempts when the system automatically regenerated rooms below the `_minRooms` threshold.



```

61
62     private void StartRoomGenerationFromRoom(Vector2Int roomIndex)
63     {
64         _roomQueue.Enqueue(roomIndex);
65         int x = roomIndex.x;
66         int y = roomIndex.y;
67         _roomGrid[x, y] = 1;
68         _roomCount++;
69
70         var initialRoom = Instantiate(_roomPrefab, GetPositionFromGridIndex(roomIndex), Quaternion.identity);
71         initialRoom.name = $"Room-{_roomCount}";
72         initialRoom.GetComponent<Room>().RoomIndex = roomIndex;
73         _roomObjects.Add(initialRoom);
74
75         // Store reference to the first room
76         if (_roomCount == 1)
77         {
78             _firstRoom = initialRoom;
79         }
80
81         // First room has no enemies
82         SpawnEnemies(initialRoom, 0);
83     }
84

```

Figure 61 - Screenshot of enemy spawning bug.

To address this, a two-part system was introduced (Seen in Figure 61 and Figure 62). First, the script now tracks the starting room persistently by storing it in a `_firstRoom` variable during initial generation, while explicitly calling `SpawnEnemies(initialRoom, 0)` to enforce the enemy-free state. Second, the `RegenerateRooms()` function was modified to preserve this room: it destroys all rooms except `_firstRoom`, clears the generation queue, resets the room grid while re-registering the starting room's position, and restarts generation from this preserved room. This ensured the starting room maintained its correct state across regeneration cycles while allowing other rooms to follow standard spawning rules.

```

// Clear all rooms except the first room and try again
private void RegenerateRooms()
{
    // Destroy all rooms except the first room
    foreach (var room in _roomObjects)
    {
        if (room != _firstRoom)
        {
            Destroy(room);
        }
    }

    // Clear the room list and add the first room back
    _roomObjects.Clear();
    if (_firstRoom != null)
    {
        _roomObjects.Add(_firstRoom);
    }

    // Reset the room grid and queue
    _roomGrid = new int[_gridSizeX, _gridSizeY];
    _roomQueue.Clear();
    _roomCount = 1; // Reset room count to 1 since we are keeping the first room
    generationComplete = false;

    // Reinitialize the first room in the grid
    Vector2Int initialRoomIndex = _firstRoom.GetComponent<Room>().RoomIndex;
    _roomGrid[initialRoomIndex.x, initialRoomIndex.y] = 1;
    _roomQueue.Enqueue(initialRoomIndex);

    // Start generating rooms from the first room
    StartRoomGenerationFromRoom(initialRoomIndex);
}

```

Figure 62 - Enemy spawning bug solution.

While this solution fixed the immediate issue of enemies spawning in the starting room, testing revealed unresolved cases where enemies would still appear. Connected rooms occasionally inherited incorrect spawn counts during regeneration, and the system didn't account for post-generation modifications to `_firstRoom`. The partial fix highlighted the need for a more robust spawning rule system that would consistently apply room-specific logic during both initial generation and regeneration. These refinements were deferred for later development to prioritize core gameplay testing, with the understanding that the current implementation provided a stable foundation for further iteration. The solution successfully prevented starting-room enemy spawns but would require additional work to fully harmonize the procedural generation and enemy placement systems.



## 5.5.9 Goal 7 – Pause Menu

```
1  using UnityEngine;
2
3  public class InterfaceManager : MonoBehaviour
4  {
5      [SerializeField] private GameObject pauseMenu;
6      public bool isPaused = false;
7
8      void Start()
9      {
10         pauseMenu.SetActive(isPaused);
11     }
12
13     void Update()
14     {
15         if (Input.GetKeyDown(KeyCode.Escape))
16         {
17             TogglePauseMenu();
18         }
19     }
20
21     private void TogglePauseMenu()
22     {
23         isPaused = !isPaused;
24         pauseMenu.SetActive(isPaused);
25
26         // Pause the game when the menu is open
27         Time.timeScale = isPaused ? 0f : 1f;
28     }
29
30     public void ResumeGame()
31     {
32         isPaused = false;
33         pauseMenu.SetActive(false);
34         Time.timeScale = 1f;
35     }
36 }
```

Figure 63 - Code Snippet of interface manager script.

The Interface Manager script (Seen in Figure 63) adds pause functionality through a Canvas-based UI pause menu panel that toggles visibility in response to player input. The system initializes by deactivating the pause menu panel in `Start()`, then monitors for ESC key presses in `Update()`. When triggered, `TogglePauseMenu()` switches the `isPaused` state to the opposite of its

current state to activate/deactivate the pause menu GameObject accordingly. The script controls game time using `Time.timeScale`, setting it to 0 when paused which freezes the gameplay and restoring it to 1, unfreezing the gameplay when resumed. A dedicated `ResumeGame()` method allows button-triggered unpausing, ensuring consistency between key and UI interactions. The pause menu reference is serialized for easy assignment in the Unity Editor, linking to a Canvas panel containing pause menu elements like buttons and text. This implementation creates a lightweight but functional pause system that can be extended with additional menu features while maintaining clear state management through the `isPaused` flag.

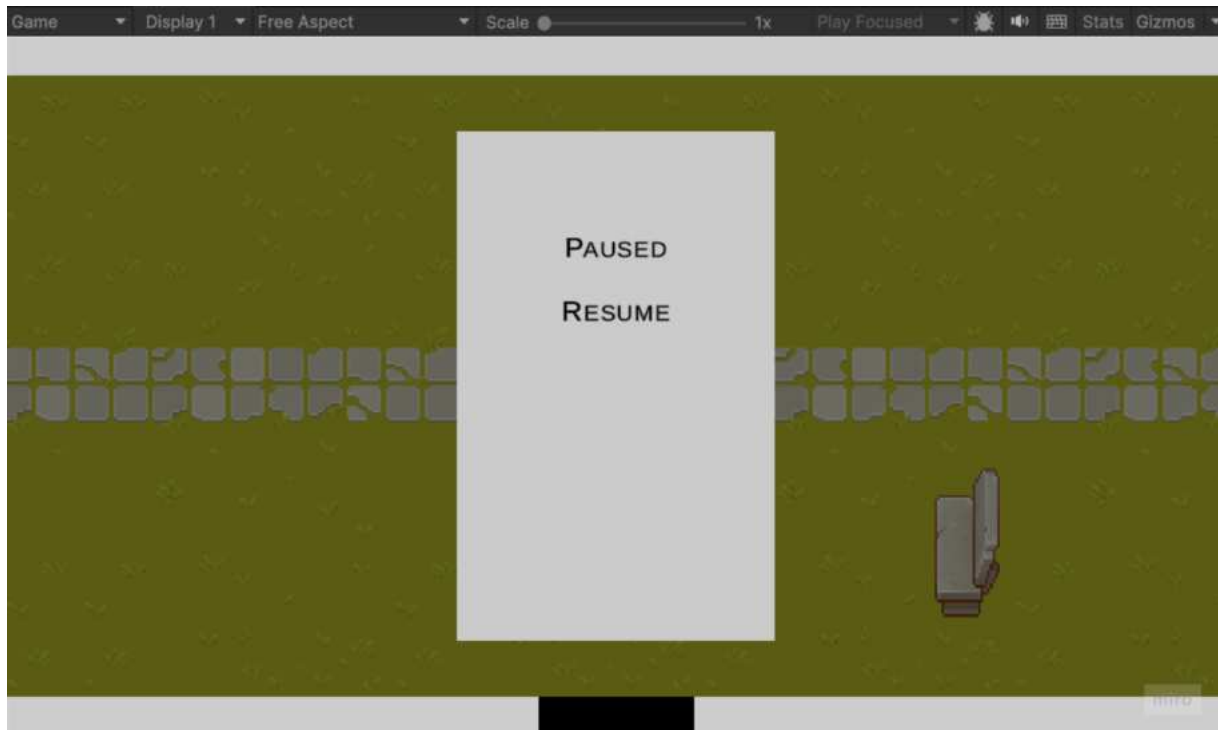


Figure 64 - Snippet of pause menu integration.

## 5.6 Sprint 5

### 5.6.1 Goals

- Add a loading screen.
- Finalize room design.
- Add door logic to the rooms.
- Add a visible health bar.
- Add a health item.
- Add an enemy design and animator.
- Fix enemy spawning in starter room bug.
- Add player into room generation.



### 5.6.2 Goal 1 – Loading Screen



*Figure 65 - Loading screen design.*

```

public class InterfaceManager : MonoBehaviour
{
    [SerializeField] private GameObject pauseMenu;
    [SerializeField] private GameObject loadingScreen;

    public bool isPaused = false;
    private RoomManager roomManager;

    void Start()
    {
        pauseMenu.SetActive(isPaused);

        roomManager = FindObjectOfType<RoomManager>();

        if (roomManager == null)
        {
            Debug.LogError("RoomManager not found in the scene!");
        }
    }

    void Update()
    {
        if (roomManager != null)
        {
            loadingScreen.SetActive(!roomManager.generationComplete);
        }

        if (Input.GetKeyDown(KeyCode.Escape))
        {
            TogglePauseMenu();
        }
    }
}

```

Figure 66 - Code snippet of loading screen logic.

The Interface Manager was extended to include a loading screen system that masks procedural generation processes (Seen in Figure 66). A new GameObject named “loadingScreen” was added, controlled by the Room Manager’s generationComplete boolean. During scene initialization, the script has a new reference to the Room Manager and implements error handling if missing. In the Update() loop, the loading screen’s active state directly mirrors the inverse of generationComplete boolean, remaining visible while rooms are generating and hiding when complete. This addresses the visual issue where players could see rooms being deleted and regenerated during failed generation attempts. The system operates independently from the existing pause functionality, with both features coexisting through separate GameObject controls. The loading screen consists of a full-screen Canvas element with static text and is activated during these key moments: initial dungeon generation, failed generation recovery, and any future scene regenerations.

### 5.6.3 Goal 2 – Finalise Room

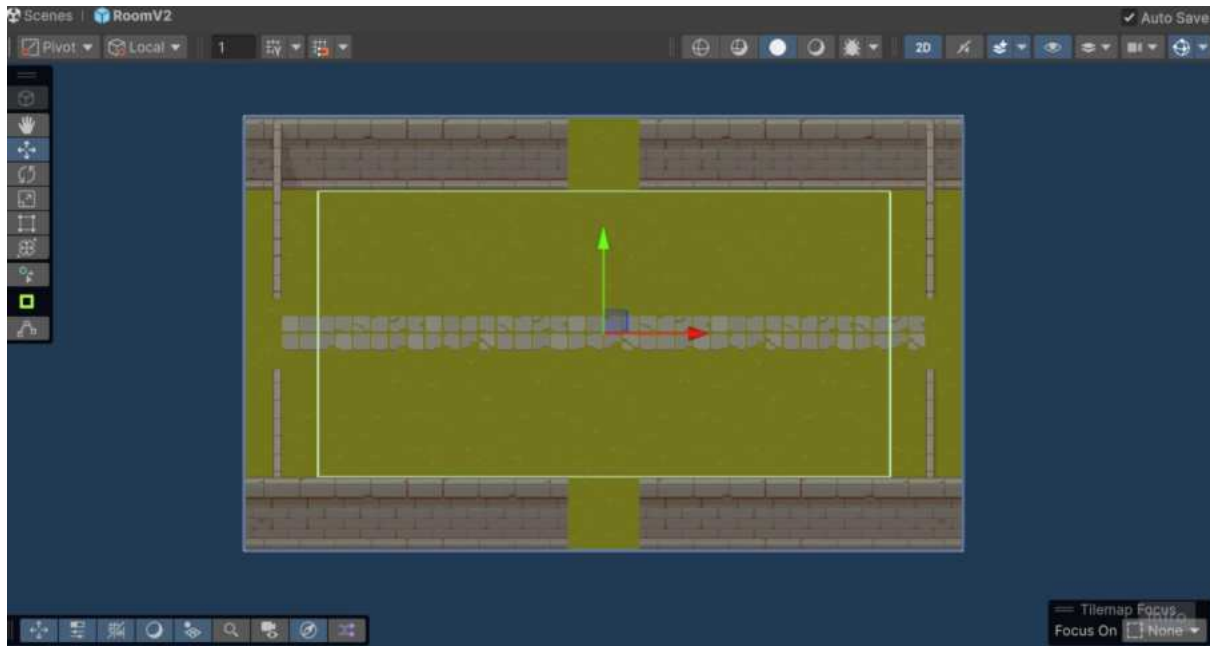


Figure 67 - Finalised room design.

The final room design (Seen in Figure 67) took a handful of iterations and failed versions before getting to this stage. There were many issues with the room baking incorrectly or the tile palette not being set up correctly which led to issues with designing proportions. Thankfully the issues were able to be rectified and a final design was created. The spaces at the four cardinal exit points is for the implementation of the smart door system where game objects will be conditionally swapped out depending on if the room has a neighbouring room or if there are enemies in the room with the player. This system should provide dynamic visual cues for the player to be able to understand what directional options they have available to them at different states of the game.

```
84
85 // Open the door in the specified direction
86 public void OpenDoor(Vector2Int direction)
87 {
88     if (direction == Vector2Int.up)
89     {
90         _topDoor.SetActive(true);
91         _topWall.SetActive(false); // Deactivate the wall when the door is open
92         Debug.Log("Opened top door and deactivated top wall.");
93     }
94
95     if (direction == Vector2Int.down)
96     {
97         _bottomDoor.SetActive(true);
98         _bottomWall.SetActive(false); // Deactivate the wall when the door is open
99         Debug.Log("Opened bottom door and deactivated bottom wall.");
100     }
101
102     if (direction == Vector2Int.left)
103     {
104         _leftDoor.SetActive(true);
105         _leftWall.SetActive(false); // Deactivate the wall when the door is open
106         Debug.Log("Opened left door and deactivated left wall.");
107     }
108
109     if (direction == Vector2Int.right)
110     {
111         _rightDoor.SetActive(true);
112         _rightWall.SetActive(false); // Deactivate the wall when the door is open
113         Debug.Log("Opened right door and deactivated right wall.");
114     }
115 }
```

Figure 68 - Code snippet of door logic.

The `OpenDoor()` method has been modified to implement a new paired door/wall system, creating more dynamic room transitions (Seen in Figure 68). For each cardinal direction, the function now manages two related `GameObjects`: a door and a wall. This creates proper spatial awareness where doorways become physically passable only when open. The system uses simple boolean activation states, setting doors active while disabling their corresponding walls, which provides immediate visual and collision feedback. This implementation serves as the first step for smarter door management.

```

// Place a wall in a specified direction only if there is no door
public void InstantiateWall(Vector2Int direction)
{
    if (direction == Vector2Int.up && !_topDoor.activeSelf)
    {
        _topWall.SetActive(true);
        Debug.Log("Activated top wall.");
    }

    if (direction == Vector2Int.down && !_bottomDoor.activeSelf)
    {
        _bottomWall.SetActive(true);
        Debug.Log("Activated bottom wall.");
    }

    if (direction == Vector2Int.left && !_leftDoor.activeSelf)
    {
        _leftWall.SetActive(true);
        Debug.Log("Activated left wall.");
    }

    if (direction == Vector2Int.right && !_rightDoor.activeSelf)
    {
        _rightWall.SetActive(true);
        Debug.Log("Activated right wall.");
    }
}

```

Figure 69 - Code snippet of door logic.

The `InstantiateWall()` function works in parallel with the existing `OpenDoor()` method by implementing conditional wall activation logic (Seen in Figure 69). This new function only activates walls in directions where no door currently exists, creating a fail-safe mechanism that prevents walls and doors from occupying the same space. The system maintains four directional checks, mirroring the door system's structure, ensuring consistent behaviour across all room boundaries. When called by the Room Manager during generation, it automatically creates sealed boundaries in unconnected directions while preserving open pathways where doors exist. This creates a complete doorway management system where doors open or close pathways when connecting rooms and walls automatically fill gaps where no connections exist. The function's conditional activation ensures it works harmoniously with the procedural generation process without overwriting manually placed doors.

```

3  public class Room : MonoBehaviour
4  {
5      [SerializeField] GameObject _topDoor;
6      [SerializeField] GameObject _bottomDoor;
7      [SerializeField] GameObject _leftDoor;
8      [SerializeField] GameObject _rightDoor;
9
10     public Vector2Int RoomIndex { get; set; }
11     public int _enemyCount;
12
13     void OnTriggerEnter2D(Collider2D other)
14     {
15         if (other.CompareTag("Enemy"))
16         {
17             _enemyCount += 1;
18             Debug.Log($"Enemy Count: {_enemyCount}");
19         }
20         else if (other.CompareTag("Player"))
21         {
22             // Close the doors if the room has enemies
23             if (_enemyCount > 0)
24             {
25                 CloseDoors();
26             }
27             else
28             {
29                 // If there are no enemies, open the neighboring doors
30                 FindObjectOfType<RoomManager>().OpenDoors(gameObject, RoomIndex.x, RoomIndex.y);
31             }
32         }
33     }
34 }

```

Figure 70 - Snippet of room script for smart door logic.

The Room script now implements dynamic door management system that responds to enemy presence, creating risk-reward exploration mechanics (Seen in Figure 70). An `_enemyCount` variable tracks enemies entering through `OnTriggerEnter2D`, incrementing when enemies spawn inside the room's collider. When the player enters which is detected via tag comparison, the system evaluates this count. If the enemy count is higher than zero, it triggers the `CloseDoors()` method to temporarily seal the room, forcing combat encounters. In cleared rooms where the enemy count is equal to zero, it requests the Room Manager to open all doors to neighbouring rooms, maintaining progression flow. This creates a gameplay loop where the player must clear rooms to advance, enemy encounters become mandatory challenges, and the environment actively responds to combat states.



```

35 void OnTriggerExit2D(Collider2D other)
36 {
37     if (other.CompareTag("Enemy"))
38     {
39         _enemyCount -= 1;
40         Debug.Log($"Enemy Count: {_enemyCount}");
41
42         // If no more enemies remain, open doors to immediate neighbors
43         if (_enemyCount == 0)
44         {
45             FindObjectOfType<RoomManager>().OpenDoors(gameObject, RoomIndex.x, RoomIndex.y);
46         }
47     }
48     else if (other.CompareTag("Player"))
49     {
50         // If the player leaves and there are no enemies, keep the doors open
51         if (_enemyCount == 0)
52         {
53             FindObjectOfType<RoomManager>().OpenDoors(gameObject, RoomIndex.x, RoomIndex.y);
54         }
55     }
56 }

```

Figure 71 - Snippet of room script for smart door logic.

The OnTriggerExit2D method (Seen in Figure 71) completes the dynamic door management system on the Room side of things by handling two scenarios. When enemies exit the room when being destroyed, the enemy counter decreases, and upon reaching zero it automatically triggers the Room Manager to reopen all available doors, creating a satisfying gameplay loop of clearing rooms and exploring more of the game. Simultaneously, the system monitors player exits as a safeguard, if the player leaves an empty room, it reconfirms the door states with the Room Manager to prevent accidental lockdowns. This two-way tracking system ensures rooms maintain accurate enemy counts while providing appropriate accessibility, with the Room Manager serving as the central coordinator for all door state changes. This implementation creates robust room behaviour where doors only remain locked during active combat encounters, automatically resolving their states when either all enemies are eliminated.

```

62
63 // Close the doors in the current room
64 void CloseDoors()
65 {
66     _topDoor.SetActive(false);
67     _bottomDoor.SetActive(false);
68     _leftDoor.SetActive(false);
69     _rightDoor.SetActive(false);
70 }

```

Figure 72 - Snippet of room script for smart door logic

```

181 public void OpenDoors(GameObject room, int x, int y)
182 {
183     Room newRoomScript = room.GetComponent<Room>();
184
185     // Only open doors if the current room has 0 enemies
186     if (newRoomScript._enemyCount == 0)
187     {
188         // Neighbors
189         Room leftRoomScript = GetRoomScriptAt(new Vector2Int(x - 1, y));
190         Room rightRoomScript = GetRoomScriptAt(new Vector2Int(x + 1, y));
191         Room topRoomScript = GetRoomScriptAt(new Vector2Int(x, y + 1));
192         Room bottomRoomScript = GetRoomScriptAt(new Vector2Int(x, y - 1));
193
194         // Open doors to immediate neighbors based on available neighbors
195         if (x > 0 && _roomGrid[x - 1, y] != 0 && leftRoomScript != null)
196         {
197             newRoomScript.OpenDoor(Vector2Int.left);
198             leftRoomScript.OpenDoor(Vector2Int.right);
199         }
200         if (x < _gridSizeX - 1 && _roomGrid[x + 1, y] != 0 && rightRoomScript != null)
201         {
202             newRoomScript.OpenDoor(Vector2Int.right);
203             rightRoomScript.OpenDoor(Vector2Int.left);
204         }
205         if (y > 0 && _roomGrid[x, y - 1] != 0 && bottomRoomScript != null)
206         {
207             newRoomScript.OpenDoor(Vector2Int.down);
208             bottomRoomScript.OpenDoor(Vector2Int.up);
209         }
210         if (y < _gridSizeY - 1 && _roomGrid[x, y + 1] != 0 && topRoomScript != null)
211         {
212             newRoomScript.OpenDoor(Vector2Int.up);
213             topRoomScript.OpenDoor(Vector2Int.down);
214         }
215     }
216 }
217

```

Figure 73 - Snippet of room script for smart door logic.

The updated `OpenDoors()` method in Room Manager script (Seen in Figure 73) now implements a coordinated door management system that works in tandem with the Room script's enemy tracking functionality. The method begins by verifying the requesting room has no remaining enemies before proceeding with any door operations, creating a critical dependency between the two systems. When safe to proceed, it identifies all valid neighbouring rooms through grid position checks and null validation, then executes bidirectional door opening, simultaneously activating doors in both the current room and connected neighbours. This creates symmetrical pathways where doors only open when the initiating room is cleared of enemies and adjacent rooms exist in the generated layout. The system handles all four cardinal directions independently, with each check verifying grid boundaries, room existence in the generation matrix, and successful component retrieval. The implementation specifically corrects directional matching to maintain proper pathing logic. This enhanced of the method directly supports the Room script's combat-driven door states by ensuring automatic reopening of cleared rooms while respecting the procedural generation constraints. The tight integration between these systems creates emergent gameplay where players must strategically clear rooms to progress, with the environment dynamically responding to their combat performance through coordinated door states across the entire dungeon layout.



### 5.6.5 Goal 3 – Health bar UI addition

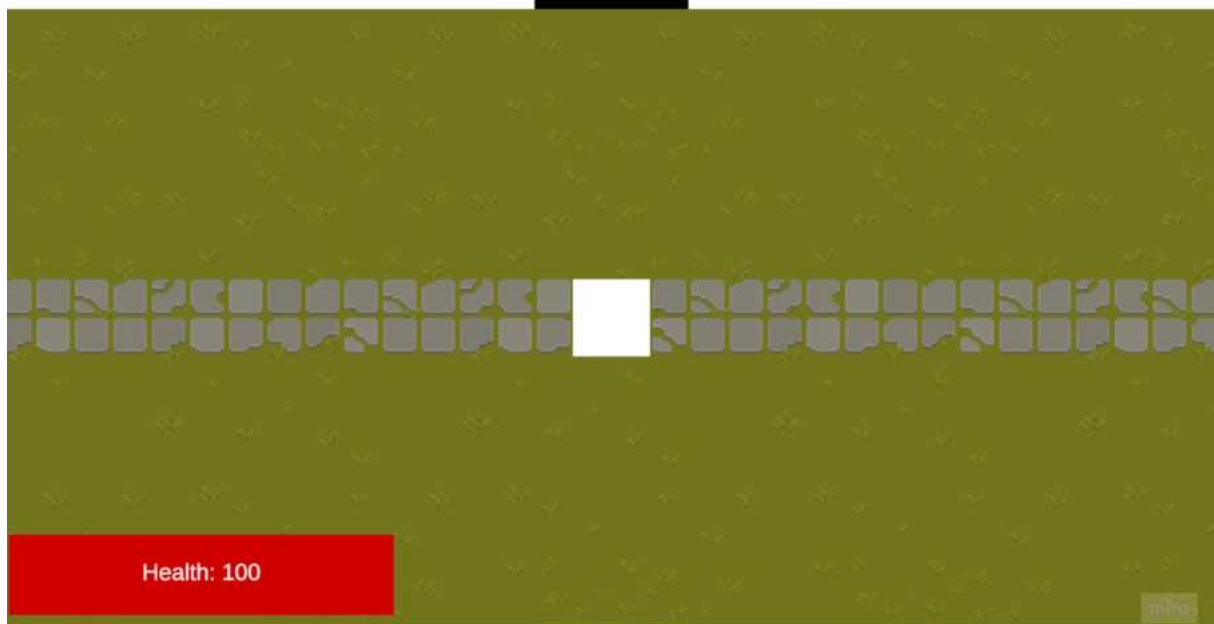


Figure 74 - First version of health bar added to user interface.

```

24
25 void Update()
26 {
27     // Update the health text display
28     _numHealth.text = $"Health: {_health}";
29
30     // Calculate health percentage for the health bar
31     float healthPercent = Mathf.Clamp01(_health / _maxHealth);
32
33     // Dynamically adjust the health bar's width based on health percentage
34     _healthBar.rectTransform.sizeDelta = new Vector2(MaxHealthBarWidth * healthPercent, HealthBarHeight);
35
36     // Player death logic
37     if (_health <= 0)
38     {
39         _health = Mathf.Max(_health, 0);
40         gameObject.SetActive(false);
41     }
42 }
43

```

Figure 75 - First version of dynamically moving health bar.

The modified Update() method inside the Player Health script (Seen in Figure 75) implements health visualization through a numeric and graphical display, though initial implementation caused a bug that caused strange height behaviour in the health bar. The TextMeshPro component reliably displays the current health value of the player, while the UI Image experienced an unexpected issue where its height would fluctuate despite being explicitly set to a constant health bar height. This occurred because the sizeDelta property was modifying both dimensions when only the width should have been adjusted. The health bar system calculates a clamped health percentage to prevent visual overflow, but the original implementation accidentally scaled the full Vector2 dimensions rather than just the x-component for width. The death check properly enforces health minimums and handles player deactivation, while the health bar now correctly maintains its height by only modifying the x-value in the sizeDelta Vector2.

```

public class PlayerHealth : MonoBehaviour
{
    [SerializeField] private float _health = 100;
    [SerializeField] private float _maxHealth = 100f;

    [SerializeField] private TextMeshProUGUI _numHealth;
    [SerializeField] private Image _healthBar;

    void Update()
    {
        // Text Output for Health Bar
        _numHealth.text = $"Health: {_health}";

        // Calculating health percentage for health bar
        float healthPercent = _health / _maxHealth;

        // Update the health bar fill amount or scale
        _healthBar.rectTransform.anchorMin = new Vector2(0, 0);
        _healthBar.rectTransform.anchorMax = new Vector2(healthPercent, 1);

        // Fixing the height and ensuring the image width is dynamic
        _healthBar.rectTransform.sizeDelta = new Vector2(_healthBar.rectTransform.sizeDelta.x, 25f);
        _healthBar.rectTransform.sizeDelta = new Vector2(500f * healthPercent, _healthBar.rectTransform.sizeDelta.y);

        // Ensuring the pivot point for the health bar is correct
        _healthBar.rectTransform.pivot = new Vector2(0, 0.5f);

        // Player Death Logic
        if (_health <= 0)
        {
            _health = Mathf.Max(_health, 0);
            gameObject.SetActive(false);
        }
    }
}

```

Figure 76 - Additions to player health script for health bar functionality.

The updated health bar implementation (Seen in Figure 76) introduces several key RectTransform adjustments to refine the health bar's visual behaviour. The anchor points are now dynamically controlled, with anchorMin fixed at (0,0) and anchorMax's x-value scaling with health percentage while maintaining a y-value of 1 - this creates smooth left-anchored shrinking. A new pivot point setting (0,0.5f) ensures the bar contracts from the left edge while staying vertically centred. The sizeDelta modifications now explicitly separate width and height adjustments, with the y-value permanently locked to 25f to prevent any height fluctuations. The width calculation uses a fixed base value multiplied by healthPercent, maintaining proper aspect ratio. These changes collectively create more polished visual feedback where the health bar now smoothly decreases from left to right while maintaining perfect dimensional stability.

## 5.6.6 Goal 4 – Health Item

```
1  using UnityEngine;
2
3  public class HealthPotion : MonoBehaviour
4  {
5      void OnTriggerEnter2D(Collider2D other)
6      {
7          var _player = other.GetComponent<PlayerHealth>();
8
9          if (_player != null && _player._health < 100)
10         {
11             _player._health = _player._health + 10;
12             Destroy(gameObject);
13         }
14     }
15 }
16
```

Figure 77 - Code snippet of health item.

The Health Potion script (Seen in Figure 77) implements a basic health restoration system through 2D collision detection. When a GameObject with a Collider2D enters the potion's trigger area, the script attempts to get the Player Health component from the colliding object. If successful and the player's health is below 100, it increments the player's health by 10 points before destroying the potion GameObject. This creates a straightforward pickup system with built-in validation that only affects the player and respects maximum health limits while minimizing confusion about used potions by removing used potions from the game world. The implementation provides immediate gameplay impact while maintaining balance through its conditional healing check.

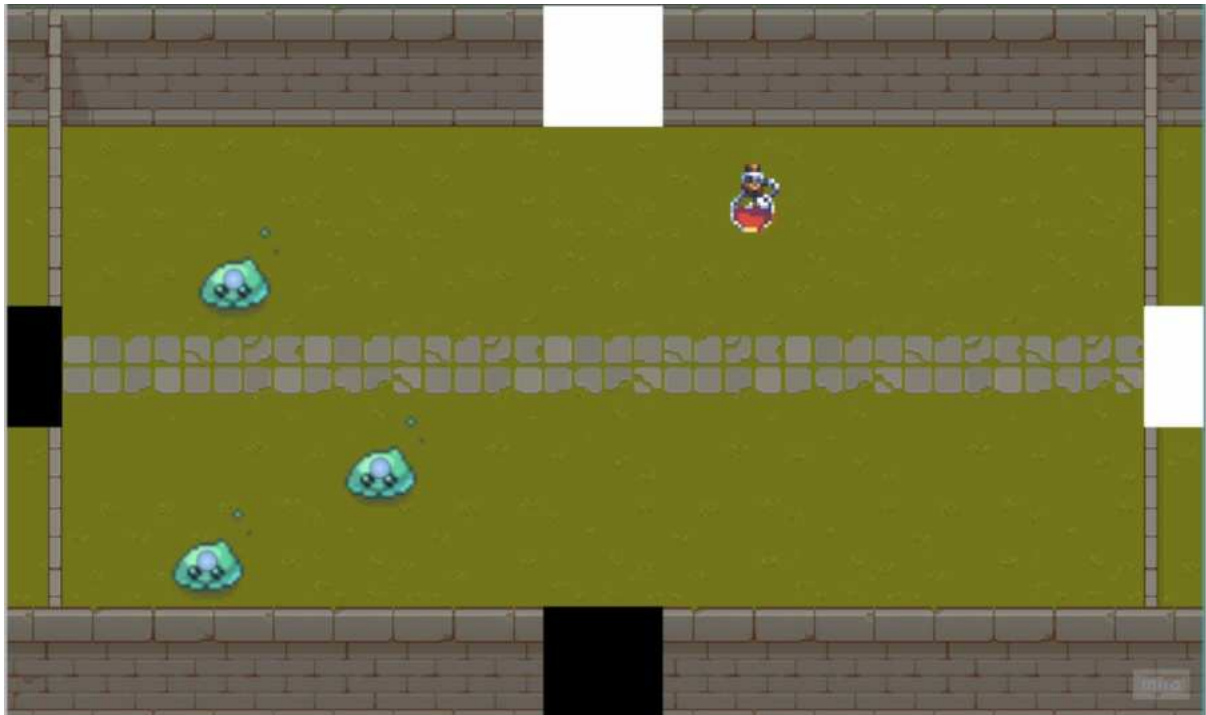


Figure 78 - Health item in the game with new enemy designs implemented.

The design of the health potion (Seen in Figure 78) was very simple to find and to implement. There is a sprite pack for potions available on the Unity Asset store which contained the red potion sprite. After importing the pack and replacing the current sprite of the health potion with the new red potion sprite. I brought the prefab into the game world and tested it through a couple of use cases and found no issues with its implementation.

```

129
130     // Spawn HealthPotion in every 3rd room and the final room
131     if (_roomCount % 3 == 0 || _roomCount == _maxRooms)
132     {
133         SpawnHealthPotion(newRoom);
134     }
135
136     return true;
137 }
138
139 // Function to spawn a HealthPotion in the room
140 private void SpawnHealthPotion(GameObject room)
141 {
142     if (_healthPotionPrefab == null) return;
143
144     // Get a random position within the room bounds
145     Vector3 spawnPosition = GetRandomPositionInRoom(room.transform.position);
146     GameObject healthPotion = Instantiate(_healthPotionPrefab, spawnPosition, Quaternion.identity, room.transform);
147
148     // Set the health potion's sprite renderer to the "Player" sorting layer
149     SpriteRenderer healthPotionRenderer = healthPotion.GetComponent<SpriteRenderer>();
150     if (healthPotionRenderer != null)
151     {
152         healthPotionRenderer.sortingLayerName = "Player";
153     }
154 }
155

```

Figure 79 - Additions to room manager script to implement health items.

After testing the functionality of the health item it was integrated into the Room Manager to create dynamic spawning throughout the dungeon (Seen in Figure 79). The implementation spawns health potions in every 3rd room and the final room, using a modulo operation for the recurring pattern. The SpawnHealthPotion() method handles instantiation with several

safeguards. First, it validates the prefab reference exists, then calculates a random position within the central area of the room with help from the enemy spawning functionality. Each potion is parented to its room for organizational clarity and automatically configured to use the "Player" sorting layer, ensuring proper visual rendering above environmental elements and proper collision logic. This creates balanced distribution where players can anticipate healing points, potions avoid edge placement near doors, and the final room guarantees a health boost before completion.

### 5.6.7 Goal 5 – Enemy design implementation



*Figure 80 - Enemy walking sprite sheet.*

The implementation of the enemy design began with the selection of a suitable sprite sheet (seen in Figure 80), which was sourced from a third-party asset library. The chosen sprite sheet was selected based on two critical factors, its comprehensive animation frames covering all necessary enemy states (idle, attack, and damage animations), and its consistent visual style that matched the game's already established pixel art aesthetic. This careful selection process ensured the sprite sheet could be integrated into the project with no modifications to the style while maintaining visual coherence across all game elements.

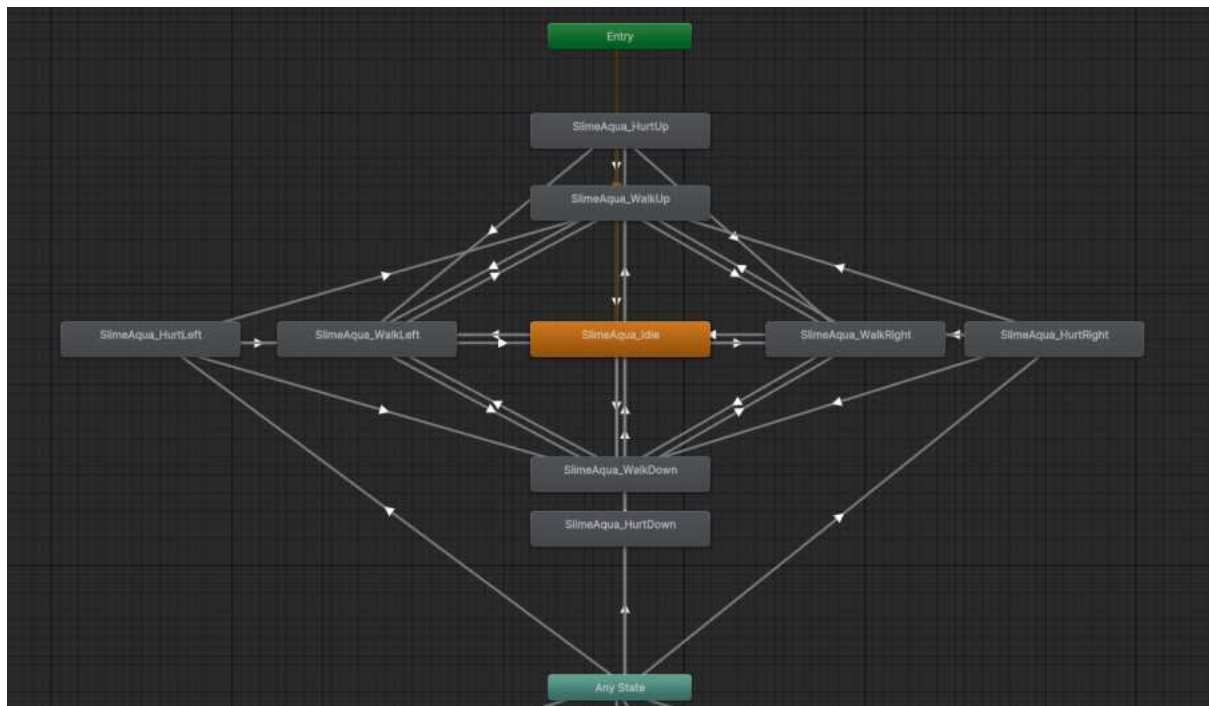


Figure 81 - Enemy animation controller.

After importing the slime sprite sheets, time was spent slicing each of the sheets into each animation sequences to then work on transitions within the animation controller (Seen in Figure 81). When building the animation controller, it was made apparent that starting small and really focusing on each animation sequence respective of each other was the only way to make sure that the transition system was built to a standard that the game deserved. This method also made it easier to make sure that every animation node had a logical transition for every use case scenario that could have happened with the use of specific booleans and triggers.



```

39     else
40     {
41         _agent.ResetPath(); // Stop moving if the player is not reachable
42     }
43
44     // Get movement delta for animator
45     float deltaY = transform.position.y - _previousPosition.y;
46     float deltaX = transform.position.x - _previousPosition.x;
47
48     float sensitivity = 0.005f; // Sensitivity threshold for animation transitions
49
50     bool isMovingUp = deltaY > sensitivity;
51     bool isMovingDown = deltaY < -sensitivity;
52     bool isMovingRight = deltaX > sensitivity;
53     bool isMovingLeft = deltaX < -sensitivity;
54
55     // Prioritize vertical movement animations if it's greater than horizontal
56     if (Mathf.Abs(deltaY) > Mathf.Abs(deltaX))
57     {
58         _animator.SetBool("WalkUp", isMovingUp);
59         _animator.SetBool("WalkDown", isMovingDown);
60         _animator.SetBool("WalkRight", false);
61         _animator.SetBool("WalkLeft", false);
62     }
63     else // Prioritize horizontal movement animations
64     {
65         _animator.SetBool("WalkRight", isMovingRight);
66         _animator.SetBool("WalkLeft", isMovingLeft);
67         _animator.SetBool("WalkUp", false);
68         _animator.SetBool("WalkDown", false);
69     }
70
71     _previousPosition = transform.position; // Update previous position
72
73

```

Figure 82 – Animation controller variables added to enemy movement.

The Enemy script was modified to implement the directional movement animations by comparing position changes between frames (Seen in Figure 82). It calculates movement deltas by tracking the enemy's current position against its previous frame position. A small sensitivity threshold prevents animation triggers during minimal movement which in turn increases the accuracy of the animation transitions. The system prioritizes vertical animations when vertical movement values exceed horizontal movement values, otherwise defaulting to horizontal animations. Four boolean parameters (Up/Down/Left/Right) are selectively activated in the Animator Controller based on these calculations, ensuring only one directional animation plays at a time. After evaluation, the current position is stored as `_previousPosition` for the next frame's comparison. This creates responsive animation transitions that match actual movement direction and prevent animation conflicts while maintaining smooth visual feedback.

```

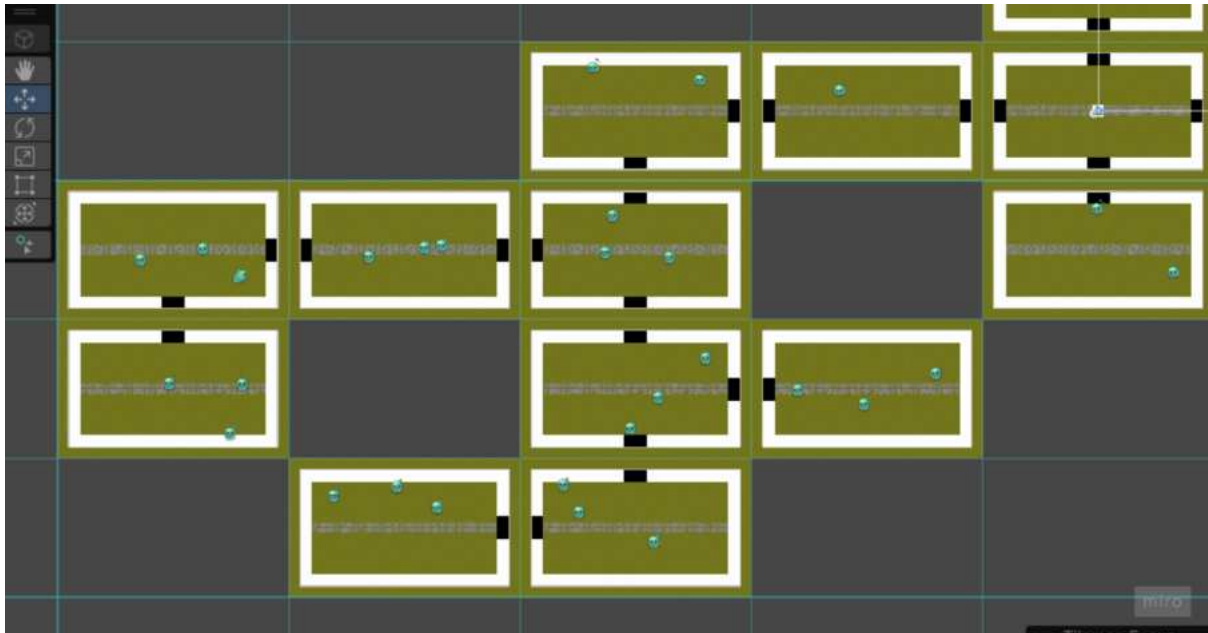
93
94 public void Knockback(Transform t)
95 {
96     Vector3 _direction = (_center.position - t.position).normalized;
97
98     _knocked = true;
99     _agent.isStopped = true;
100     _agent.velocity = _direction * _knockbackVel;
101
102     // Enable Hurt state
103     _animator.SetBool("Hurt", true);
104
105     // Trigger the correct Hurt animation based on movement direction
106     if (_animator.GetBool("WalkUp"))
107     {
108         _animator.SetTrigger("HurtUp");
109     }
110     else if (_animator.GetBool("WalkLeft"))
111     {
112         _animator.SetTrigger("HurtLeft");
113     }
114     else if (_animator.GetBool("WalkRight"))
115     {
116         _animator.SetTrigger("HurtRight");
117     }
118     else
119     {
120         _animator.SetTrigger("HurtDown");
121     }
122
123     StartCoroutine(UnKnocked());
124 }
125

```

Figure 83 - Animation controller variables added to enemy damage.

The Enemy Knockback() method was modified to integrate directional hurt animations alongside the existing physics system (Seen in Figure 83). When triggered, the method first calculates the knockback direction away from the knockback trigger and sets the hurt state to true. The system then starts three key actions. First it stops the NavMeshAgent's pathfinding temporarily to avoid pathfinding errors from occurring, it then applies the knockback physics, and finally it activates animation responses through a two-layer system. The base "Hurt" boolean enables the hurt state in the Animator Controller, while specific directional triggers are fired based on the enemy's current movement direction which was handled by the movement system. This ensures the knockback animation matches the correct direction when hit. If no directional movement booleans are active, it defaults to the "HurtDown" animation as a safeguard. The knockback state is automatically cleared after a set duration via the UnKnocked coroutine.





*Figure 84 - Enemy designs implemented into game.*

After configuring the animation controller and the code implementation for the enemy animations, it was time to test the current state of the enemy animations (Seen in Figure 84). There were some minor issues with transitions at the beginning of testing, much on the side of the animation controller but after some quick fixes, it all became much smoother and much more consistent. The only key animation sequence we were missing was the death animations for the enemies as it would require to add more logic to the enemy script to be able to control the death state.

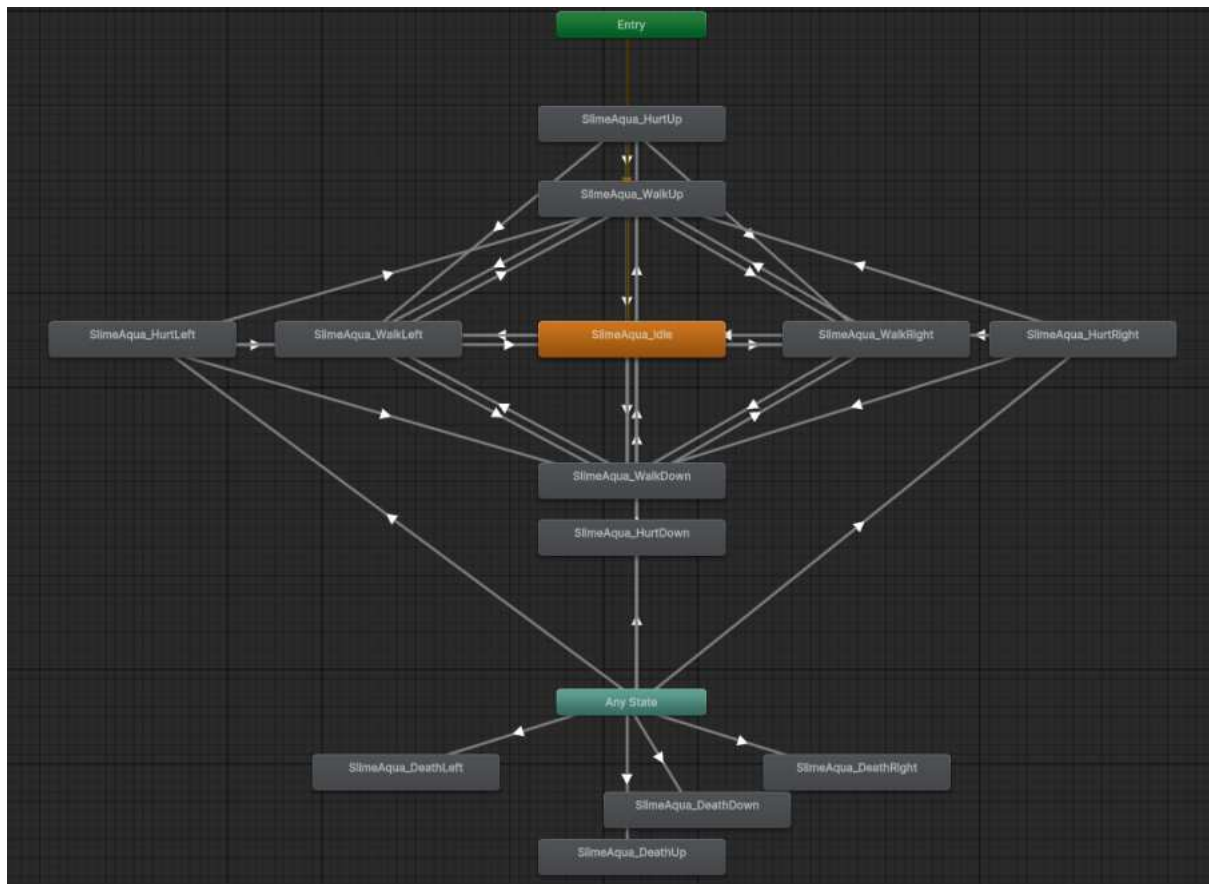


Figure 85 - Addition of death animations to enemy animation controller.

When implementing the death animations, firstly time was spent on slicing and bringing animation sequences into the existing animation controller (Seen in Figure 85), the death animations were connected to the “AnyState” node. Further transitions were not needed seeing as the enemy GameObject would be destroyed after the death animations occurred, making this implementation into the animation controller very simple.

```

116
117 public bool IsDead()
118 {
119     return _animator.GetBool("Dead");
120 }
121
122 private IEnumerator Die()
123 {
124     // Allow knockback to finish before disabling the NavMeshAgent
125     yield return new WaitForSeconds(0.5f);
126
127     // Disable the NavMeshAgent to stop pathfinding
128     _agent.enabled = false;
129
130     // Wait for the remainder of the death animation
131     yield return new WaitForSeconds(0.5f);
132
133     // Destroy the enemy object
134     Destroy(gameObject);
135 }
136

```

Figure 86 - Animation variables being implemented into enemy death logic.

The implementation of the death animations into the Enemy script (Seen in Figure 86) used a two-phase approach that makes sure the existing gameplay logic works as intended while seamlessly bringing in the animations. The `IsDead()` method creates an easy way to enable the dead state within the enemy's animation controller, which then works the same way as the knockback animations handling which worked closely with the enemy movement system. The `Die()` coroutine handles the timing to make sure that the knockback physics and the death animation are not cancelled out before the enemy is destroyed from the scene. This created a visually appealing death sequence where nothing important to the sequence gets interrupted.

### 5.6.8 Goal 6 – Fixing Enemy in Starting Room bug

When tackling this bug during a previous sprint, it was mentioned that the bug wasn't completely fixed although it was occurring less frequently. This task was dedicated to eradicating this bug from the game entirely.

During testing, each part of the room management system was inspected and tested for its intended purpose. When testing the room regeneration systems, it was noticed that not all rooms were generating but there was also no error being thrown. It was discovered that every room was being generated but due to the regeneration not recognising the first room in the queue, a room was being generated on top of the first room, which brought the enemy inside the player's spawn area. This was not seen in the game due to the sorting layers being so well organized.

```

64     private void RegenerateAllRooms()
65     {
66         // Destroy all existing rooms
67         foreach (var room in _roomObjects)
68         {
69             Destroy(room);
70         }
71
72         // Clear the room list, grid, and queue
73         _roomObjects.Clear();
74         _roomGrid = new int[_gridSizeX, _gridSizeY];
75         _roomQueue.Clear();
76         _roomCount = 0;
77         generationComplete = false;
78
79         // Generate the first room
80         Vector2Int initialRoomIndex = new Vector2Int(_gridSizeX / 2, _gridSizeY / 2);
81         StartRoomGenerationFromRoom(initialRoomIndex);
82
83         // Spawn or move the player to the first room
84         MovePlayerToFirstRoom();
85     }

```

Figure 87 - Modifications to the room manager script.

The `RegenerateAllRooms()` method was completely restructured (Seen in Figure 87). Instead of destroying all rooms, it now preserves the first room, then resets all generation variables such as `_roomGrid` and `_roomQueue`, before restarting the regeneration process. This restructuring allows the same functionality as before but without ignoring the first room. The `MovePlayerToFirstRoom()` function works as a safeguard in case of an issue with the first rooms coordinates on the grid to ensure proper positioning after regeneration.

```

87     private void StartRoomGenerationFromRoom(Vector2Int roomIndex)
88     {
89         _roomQueue.Enqueue(roomIndex);
90         int x = roomIndex.x;
91         int y = roomIndex.y;
92         _roomGrid[x, y] = 1;
93         _roomCount++;
94
95         var initialRoom = Instantiate(_roomPrefab, GetPositionFromGridIndex(roomIndex), Quaternion.identity);
96         initialRoom.name = $"Room-{_roomCount}";
97         initialRoom.GetComponent<Room>().RoomIndex = roomIndex;
98         _roomObjects.Add(initialRoom);
99
100         // First room has no enemies
101         SpawnEnemies(initialRoom, 0);
102     }

```

Figure 88 - Modifications to the room manager script.

The `StartRoomGenerationFromRoom()` method was modified to improve room tracking from the first generation (Seen in Figure 88). It now consistently places the first room in the centre of the grid and registers its position with the `_roomGrid` array and the `_roomObjects` list. The function explicitly tags the first room, ensuring that no enemies spawn inside of it during any generation attempt.

```

104     private bool TryGenerateRoom(Vector2Int roomIndex)
105     {
106         int x = roomIndex.x;
107         int y = roomIndex.y;
108
109         if (_roomCount >= _maxRooms) return false;
110         if (Random.value < 0.5f && roomIndex != Vector2Int.zero) return false;
111         if (CountAdjacentRooms(roomIndex) > 1) return false;
112         if (_roomGrid[x, y] != 0) return false;
113
114         _roomQueue.Enqueue(roomIndex);
115         _roomGrid[x, y] = 1;
116         _roomCount++;
117
118         var newRoom = Instantiate(_roomPrefab, GetPositionFromGridIndex(roomIndex), Quaternion.identity);
119         newRoom.GetComponent<Room>().RoomIndex = roomIndex;
120         newRoom.name = $"Room-{_roomCount}";
121         _roomObjects.Add(newRoom);
122
123         OpenDoors(newRoom, x, y);
124
125         // Determine enemy count based on room number
126         int enemyCount = GetEnemyCountForRoom(_roomCount);
127         SpawnEnemies(newRoom, enemyCount);
128
129         return true;
130     }
131

```

Figure 89 - Modifications to the room manager script.

The TryGenerateRoom() method was improved with additional validation checks to prevent incorrect room placement (Seen in Figure 89). Now it verifies four conditions before spawning a room, these conditions include the maximum room count, random generation chance, adjacent room count, and a grid position check. The grid position check is the most important change for this bug, making sure the position is not occupied before spawning a room inside. Only successful validation of these conditions enqueues the room to be instantiated.

```
132     private void MovePlayerToFirstRoom()
133     {
134         if (_roomObjects.Count == 0)
135         {
136             Debug.LogError("No rooms available to place the player.");
137             return;
138         }
139
140         // Get the first room
141         GameObject firstRoom = _roomObjects[0];
142         Vector3 playerSpawnPosition = firstRoom.transform.position;
143
144         // Spawn or move the player
145         if (_player == null)
146         {
147             _player = Instantiate(_playerPrefab, playerSpawnPosition, Quaternion.identity);
148         }
149         else
150         {
151             _player.transform.position = playerSpawnPosition;
152         }
153
154         Debug.Log("Player moved to the first room.");
155     }
156
```

Figure 90 - Modifications to the room manager script.

The `MovePlayerToFirstRoom()` was added (Seen in Figure 90) as a coordination function which handles the player's position in conjunction with the reworked generation system. It either spawns a player or moves an existing player to the first room, guaranteeing that the player starts in the first room.



```

192 public void OpenDoors(GameObject room, int x, int y)
193 {
194     Room newRoomScript = room.GetComponent<Room>();
195
196     // Only open doors if the current room has 0 enemies
197     if (newRoomScript._enemyCount == 0)
198     {
199         // Neighbors
200         Room leftRoomScript = GetRoomScriptAt(new Vector2Int(x - 1, y));
201         Room rightRoomScript = GetRoomScriptAt(new Vector2Int(x + 1, y));
202         Room topRoomScript = GetRoomScriptAt(new Vector2Int(x, y + 1));
203         Room bottomRoomScript = GetRoomScriptAt(new Vector2Int(x, y - 1));
204
205         // Open doors to immediate neighbors based on available neighbors
206         if (x > 0 && _roomGrid[x - 1, y] != 0 && leftRoomScript != null)
207         {
208             newRoomScript.OpenDoor(Vector2Int.left);
209             leftRoomScript.OpenDoor(Vector2Int.right);
210         }
211         if (x < _gridSizeX - 1 && _roomGrid[x + 1, y] != 0 && rightRoomScript != null)
212         {
213             newRoomScript.OpenDoor(Vector2Int.right);
214             rightRoomScript.OpenDoor(Vector2Int.left);
215         }
216         if (y > 0 && _roomGrid[x, y - 1] != 0 && bottomRoomScript != null)
217         {
218             newRoomScript.OpenDoor(Vector2Int.down);
219             bottomRoomScript.OpenDoor(Vector2Int.up);
220         }
221         if (y < _gridSizeY - 1 && _roomGrid[x, y + 1] != 0 && topRoomScript != null)
222         {
223             newRoomScript.OpenDoor(Vector2Int.up);
224             topRoomScript.OpenDoor(Vector2Int.down);
225         }
226     }

```

Figure 91 - Modifications to the room manager script.

The OpenDoors() function was improved (Seen in Figure 91) with neighbour validation which cross-references the \_roomGrid array, preventing doors from opening in invalid directions. The function now only activates when the current room has no enemies. The script also uses the GetRoomScriptAt() method to communicate with other rooms, ensuring that doors open between neighbouring rooms.

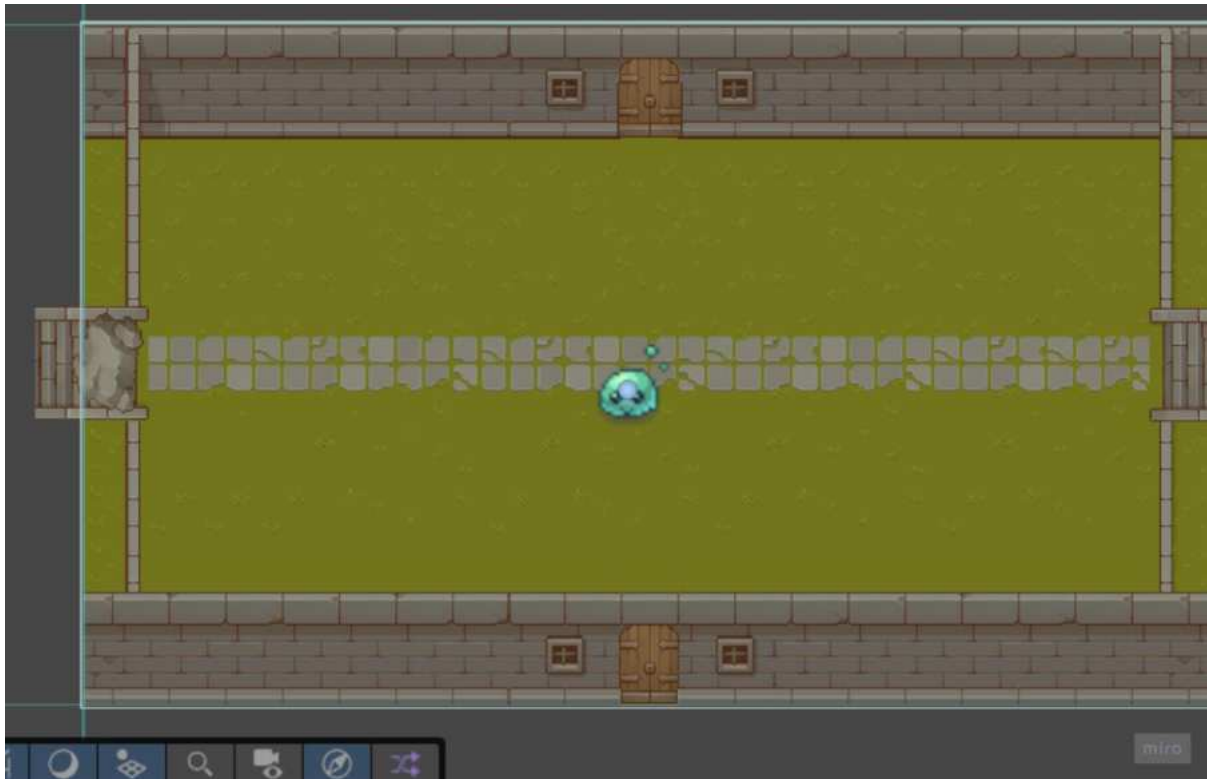
## 5.7 Sprint 6

### 5.7.1 Goals

- Add side door design.
- Add a fully designed player character.
- Update melee system.
- Fix enemy clipping bug.
- Add mini map to user interface.
- Update health bar design.

- Update menu designs.
- Add options menu.
- Add a main menu design.
- Add audio system.
- Add controller support.

### 5.7.2 Goal 1 – Add Door Design to Room



*Figure 92 - Updated room design with door prefabs brought in*

The final stage of the room design process involved creating the doors (seen in Figure 92). Each door required two distinct sprite variations to function within the smart door system, one for the closed state and one for the open state. Designing the top and bottom doors proved straightforward, despite relying on an unfamiliar prefab creation technique with no prior research.

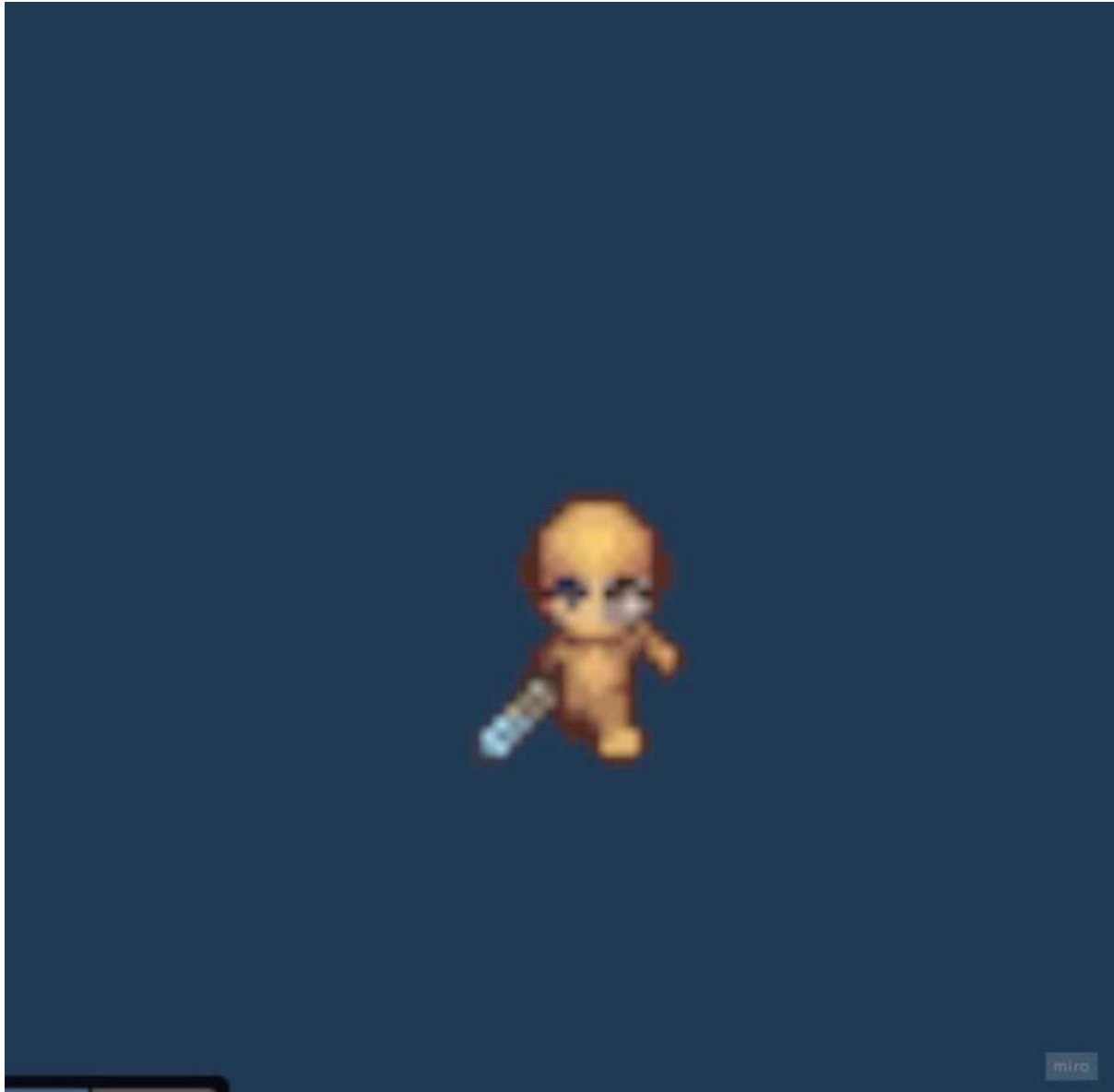
The door prefabs in both the closed and open state were assembled using four 2D game objects each, equipped with Sprite Renderer components. Suitable door and wall sprites were selected from the chosen tile palette and assigned to the corresponding game objects. A Box Collider 2D was added to the closed state prefabs to ensure it behaved the same as the previous placeholder models. Careful alignment minimized visual clipping and ensured the components fit seamlessly within the environment, followed by functionality testing to confirm behaviour consistency with the previous model.

Attention then turned to the left and right doors, which presented a challenge. The tile palette that was chosen lacked appropriate sprites for side-facing doors. Several approaches were attempted, such as rotating existing door models on the Z-axis to simulate the desired effect,



but the results were unconvincing and felt sloppy. A breakthrough came when a photography student suggested using stairs instead of doors for the room's sides. A rough prototype based on this idea was assembled and integrated. A blocked version of the stairs, featuring a rock obstruction, was also created to serve as the closed-door variation. These designs replaced the original placeholders and passed testing without issues.

### 5.7.3 Goal 2 – Add a Player Design



*Figure 93 - New character design being implemented*

The implementation of a player design (seen in Figure 93) began with finding a suitable sprite sheet that would suit the needs of the game. The sprite sheet was sourced from the same third-party website as the enemy slime sprite sheet. Both asset packs aligned well with the existing game environment, maintaining the visual aesthetic that was established early in development. The player sprite sheet included a wide range of animations which made it an easy choice, although not all animations were implemented, such as all four direction based idle animations, to ensure that the essential animations were functional and ready in time for the major user testing phase as at this stage of development it was rapidly approaching.

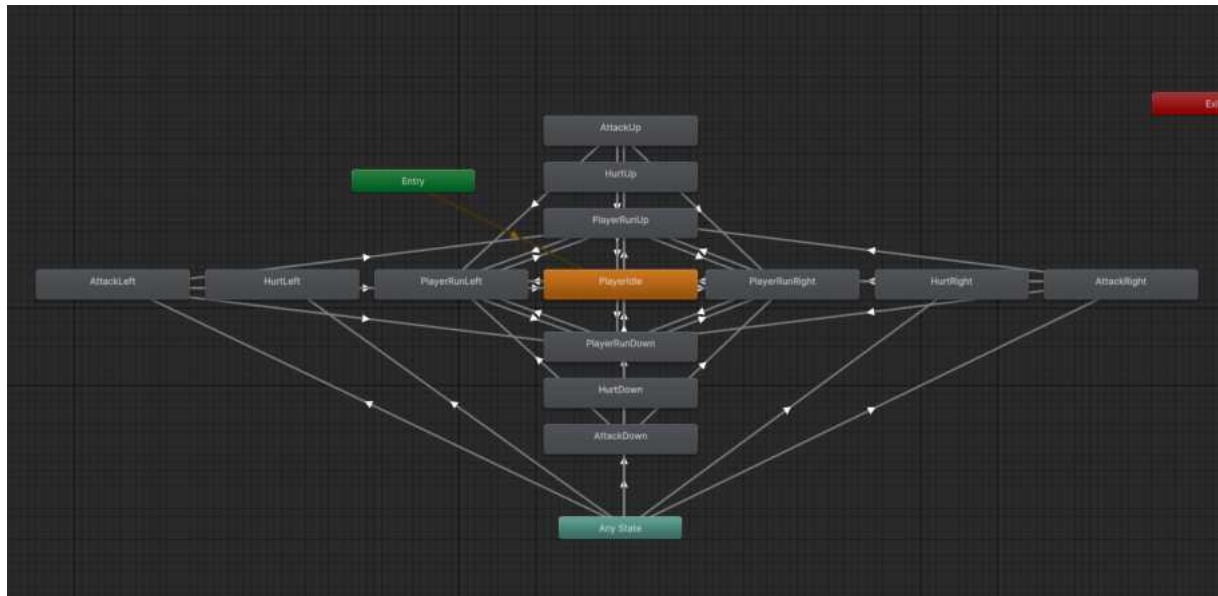


Figure 94 - Player animation controller

Each sprite sheet was sliced into its respective animations before being configured in the player animation controller (Seen in Figure 94). Transitions for each animation state were configured in the controller, using the experience gained with the enemy animation controller to create a more refined controller straight away, without needing modifications later. The process began with the player's movement animations, focusing on transitions between directional states.

A short timer was then added to reset the animation back to an idle state, preventing any unintended crossover between transitions. Once the movement animations were in place, the hurt animations were implemented using a similar approach, followed by the addition of the attack animations. As the animation controller expanded, a lot of care was taken to ensure each transition felt logical.

```

private void HandleMovement()
{
    _movement.Set(InputManager.Movement.x, InputManager.Movement.y);
    _rb.linearVelocity = _movement * _moveSpeed;

    if (_movement.sqrMagnitude <= 0f)
    {
        _idleTimer -= Time.deltaTime;

        if(_idleTimer <= 0f && !_playerMelee._attackTriggered)
        {
            SetIdleState();
        }
    }
    else
    {
        // Ensure Idle is false when moving
        _idleTimer = _idleDelay;
        _animator.SetBool("Idle", false);

        _animator.SetBool("WalkDown", _movement.y < 0);
        _animator.SetBool("WalkUp", _movement.y > 0);
        _animator.SetBool("WalkLeft", _movement.x < 0);
        _animator.SetBool("WalkRight", _movement.x > 0);

        if (_movement.y < 0) _previousDirection = PlayerDirection.Down;
        else if (_movement.y > 0) _previousDirection = PlayerDirection.Up;
        else if (_movement.x < 0) _previousDirection = PlayerDirection.Left;
        else if (_movement.x > 0) _previousDirection = PlayerDirection.Right;
    }
}

```

Figure 95 - Player movement code updated with animator integrations.

The HandleMovement() function within the player movement script (Seen in Figure 95) was updated to interact with the player's animation controller, this was largely based on the same system that was developed for the enemy animations. Directional booleans were set based on movement along the X or Y axis, triggering the corresponding movement animations. While the player is in motion, the idle timer resets, once movement stops, the timer begins counting down and finally transitioning to the idle state after a short time. This straightforward system functions effectively, and thanks to careful setup of the animation controller transitions, no directional overlap occurs during horizontal or vertical movement. The code evaluates the dominant movement direction and activates the appropriate animation accordingly.

```
private void SetIdleState()
{
    // Set Idle to true when not moving
    _animator.SetBool("Idle", true);
    _animator.SetBool("WalkDown", false);
    _animator.SetBool("WalkLeft", false);
    _animator.SetBool("WalkRight", false);
    _animator.SetBool("WalkUp", false);
    _animator.SetBool("Hurt", false);
}
```

Figure 96 - Player movement code updated with animator integrations.

The SetIdleState() function (Seen in Figure 96) is triggered when the idle timer has reached zero. It is a very simple function that sets every non-idle state to false and sets the idle state to true. When the idle state is set to true the idle animation plays and begins looping for as long as the player is standing still. There is no exit time set on the idle animation so it transitions quickly and smoothly to whatever movement or attack animation gets triggered.

```

public void Knockback(Transform t)
{
    Vector2 direction = _center.position - t.position;

    _knocked = true;

    // Enable Hurt state
    _animator.SetBool("Hurt", true);

    // Trigger the appropriate Hurt animation based on the last direction
    switch (_previousDirection)
    {
        case PlayerDirection.Down:
            _animator.SetTrigger("HurtDown");
            break;
        case PlayerDirection.Up:
            _animator.SetTrigger("HurtUp");
            break;
        case PlayerDirection.Left:
            _animator.SetTrigger("HurtLeft");
            break;
        case PlayerDirection.Right:
            _animator.SetTrigger("HurtRight");
            break;
    }

    _rb.linearVelocity = direction.normalized * _knockbackVel;

    StartCoroutine(Unknocked());
}

```

Figure 97 - Player movement code updated with animator integrations.

The hurt animations for the player were integrated within the Knockback() function in the Player Movement script (Seen in Figure 97), as this placement made the most sense, allowing the hurt state to be activated during knockback and reset within the UnKnocked() coroutine.

Within the HandleMovement() function, an if statement tracks and stores the player's previous movement direction using an Enum variable. This value is then referenced to ensure the correct hurt animation is triggered. When the knockback function is called, the hurt boolean is set to true, and the corresponding directional trigger is activated based on the stored movement direction. With both the boolean and trigger conditions met, the animation controller's "Any State" logic identifies the appropriate transition and plays the correct hurt animation.

```
private IEnumerator Unknocked()
{
    // Get the current animation clip length
    float animationLength = _animator.GetCurrentAnimatorStateInfo(0).length;

    // Use the longer time between knockback and animation duration
    float waitTime = Mathf.Max(_knockedTime, animationLength);

    yield return new WaitForSeconds(waitTime);

    _knocked = false;
    _animator.SetBool("Hurt", false);
}
```

Figure 98 - Player movement code updated with animator integrations.

The UnKnocked() coroutine (Seen in Figure 98) is responsible for resetting both the hurt animation state and the knocked player state after a specified duration of time. Initially, there were timing discrepancies between the knockback effect and the length of the hurt animation. To resolve this, the coroutine was adjusted to wait for the longer of the two durations, either the knockback time or the current hurt animation length. This change addressed the issue without introducing any noticeable delay to the knockback effect. Once the wait time concludes, both the knocked and hurt states are set to false, allowing the HandleMovement() function to resume normal operation and enabling movement and attack animations to play as needed.



## 5.7.4 Goal 3 – Update Combat System

```
public Vector2 HandleAim()
{
    if (_isAttacking)
        return _lockedAimDirection; // Use locked direction during attack

    Vector3 mouseWorldPosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
    Vector2 aimDirection = (mouseWorldPosition - _center.position).normalized;

    if (Mathf.Abs(aimDirection.x) > Mathf.Abs(aimDirection.y))
    {
        aimDirection = new Vector2(Mathf.Sign(aimDirection.x), 0);
    }
    else
    {
        aimDirection = new Vector2(0, Mathf.Sign(aimDirection.y));
    }

    // Lock in aim direction if an attack is starting
    if (_playerMelee._attackTriggered)
    {
        _lockedAimDirection = aimDirection;
        _isAttacking = true;
    }

    float angle = Mathf.Atan2(aimDirection.y, aimDirection.x) * Mathf.Rad2Deg;

    return aimDirection;
}
```

Figure 99 - Updated handle aim function for easier animation controlling

The aiming system (Seen in Figure 99) was updated to determine and return an attack direction based on the most dominant cardinal direction of the mouse position when the player is attacking. This directional information is passed to the Weapon script, which manages the remainder of the attack logic. This approach allows the attack animations to be selected in a manner similar to the player's hurt animations, ensuring consistency across animation states.

```

public class PlayerMelee : MonoBehaviour
{
    public GameObject _meleeTop;
    public GameObject _meleeBottom;
    public GameObject _meleeLeft;
    public GameObject _meleeRight;

    private InterfaceManager _interfaceManager;
    private Animator _animator;

    public float _attackDuration = 0.5f;
    public float _attackCooldown = 0.5f;
    public float _cooldownTimer = 0f;

    public bool _attackTriggered = false;

    private PlayerMovement _playerMovement;

    void Awake()
    {
        _meleeTop.SetActive(false);
        _meleeBottom.SetActive(false);
        _meleeLeft.SetActive(false);
        _meleeRight.SetActive(false);

        _interfaceManager = FindObjectOfType<InterfaceManager>();
        _animator = GetComponent<Animator>();
        _playerMovement = GetComponent<PlayerMovement>();
    }
}

```

Figure 100 - Updated player melee script with animation controller integration

At the top of the updated Player Melee script (seen in Figure 100), both the animator and Player Movement script are referenced to integrate the new aim mechanic into the combat system. This setup allows the aim direction, calculated by the Handle Aim function, to be accessed and passed to the Animator. The Melee script serves as a mediator between these two components, ensuring the correct directional animation is triggered during attacks.



```

void Attack()
{
    _attackTriggered = true;
    _cooldownTimer = _attackCooldown;
    _animator.SetBool("Attacking", true);
    Vector2 aimDirection = _playerMovement.HandleAim(); // Get the aim direction from PlayerMovement

    DeactivateAllMeleeObjects();

    if (aimDirection == Vector2.up) // Aim is upwards
    {
        _animator.SetTrigger("AttackUp");
        _meleeTop.SetActive(true);
    }
    else if (aimDirection == Vector2.down) // Aim is downwards
    {
        _animator.SetTrigger("AttackDown");
        _meleeBottom.SetActive(true);
    }
    else if (aimDirection == Vector2.left) // Aim is left
    {
        _animator.SetTrigger("AttackLeft");
        _meleeLeft.SetActive(true);
    }
    else if (aimDirection == Vector2.right) // Aim is right
    {
        _animator.SetTrigger("AttackRight");
        _meleeRight.SetActive(true);
    }

    StartCoroutine(EndAttack());
}

```

Figure 101 - Updated player melee script with animation controller integration.

The Attack() function (Seen in Figure 101) was updated to add an animation system like the one used in the player's Knockback() function. It begins by triggering the attack and resetting the cooldown timer to prevent attacks from being triggered too quickly. The aim direction is then retrieved from the Player Movement script, and all melee hitboxes are initially deactivated to prepare for the attack. A series of conditional statements follow, each corresponding to a specific attack direction, enabling the player to perform directional attacks as needed. Finally, the EndAttack() coroutine is called to disable the attack state and reset the necessary variables.

```
private IEnumerator EndAttack()  
{  
    yield return new WaitForSeconds(_attackDuration);  
    _animator.SetBool("Attacking", false);  
    _attackTriggered = false;  
    DeactivateAllMeleeObjects();  
  
    _playerMovement._isAttacking = false;  
}  
  
private void DeactivateAllMeleeObjects()  
{  
    _meleeTop.SetActive(false);  
    _meleeBottom.SetActive(false);  
    _meleeRight.SetActive(false);  
    _meleeLeft.SetActive(false);  
}
```

Figure 102 - Updated player melee script with animation controller integration.

The EndAttack() coroutine (seen in Figure 102) is fully executed only after the attack duration timer has completed. Once triggered, it resets the attack state and the \_attackTriggered boolean. All melee objects are then deactivated to prevent conflicts during the next attack sequence. This system has performed reliably during testing, with no issues observed.

#### 5.7.5 Goal 4 – Fix Player Clipping Bug

During combat testing, an issue was found where enemies could push the player inside the wall boundaries due to improper interaction with the player knockback system. When the player received damage, knockback would be applied as intended, however if the player was cornered, they could be pushed into and through walls, becoming stuck within the wall. After extensive testing, the cause was traced to the enemy's Rigidbody configuration. The enemy's Rigidbody had been set to Kinematic to fix a previous bug involving the AI pathfinding system during player knockback, but this inadvertently allowed enemies to ignore level boundaries and push the player into inaccessible areas.

```

public class KnockbackTrigger : MonoBehaviour
{
    private Rigidbody2D _rb;
    private Enemy _enemy;
    private PlayerMovement _player;

    void Awake()
    {
        _rb = GetComponent<Rigidbody2D>();
        _enemy = GetComponent<Enemy>();
    }

    void OnCollisionEnter2D(Collision2D other)
    {
        _player = other.collider.GetComponent<PlayerMovement>();

        if (_player != null && !_enemy.IsDead())
        {
            // Temporarily set Rigidbody to kinematic
            _rb.isKinematic = true;

            // Apply knockback to player
            _player.Knockback(transform);

            // Revert Rigidbody to dynamic after a short delay
            Invoke(nameof(ResetRigidbody), 0.5f);
        }

        private void ResetRigidbody()
        {
            _rb.isKinematic = false;
            _rb.linearVelocity = Vector2.zero; // Reset velocity to prevent unwanted movement
        }
    }
}

```

Figure 103 - Updated enemy script to fix clipping bug.

To fix this issue, the Rigidbody was changed back to a Dynamic state, though this initially reintroduced pathfinding issues and then dynamically switching the enemy's Rigidbody to Kinematic for a brief period, specifically half a second, when player triggers the enemy's knockback system. This allowed the enemy AI to remain unaffected during knockback interactions, while still respecting level boundaries. Updates can be seen in Figure 103.

### 5.7.6 Goal 5 – Update User Interface

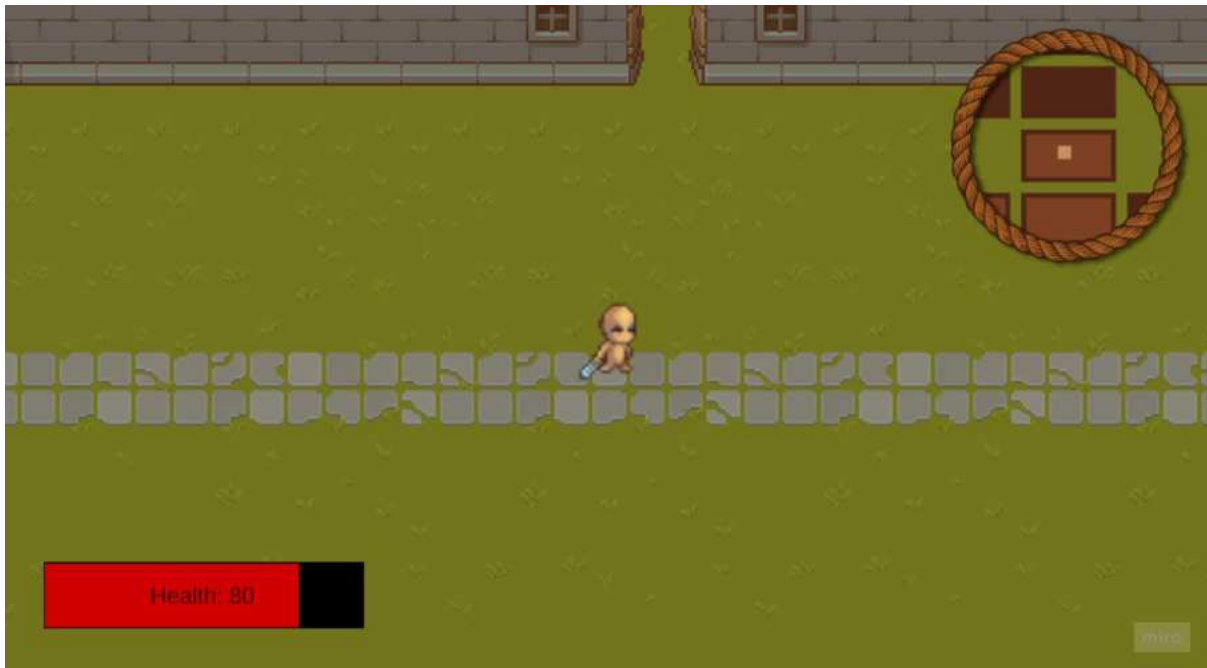


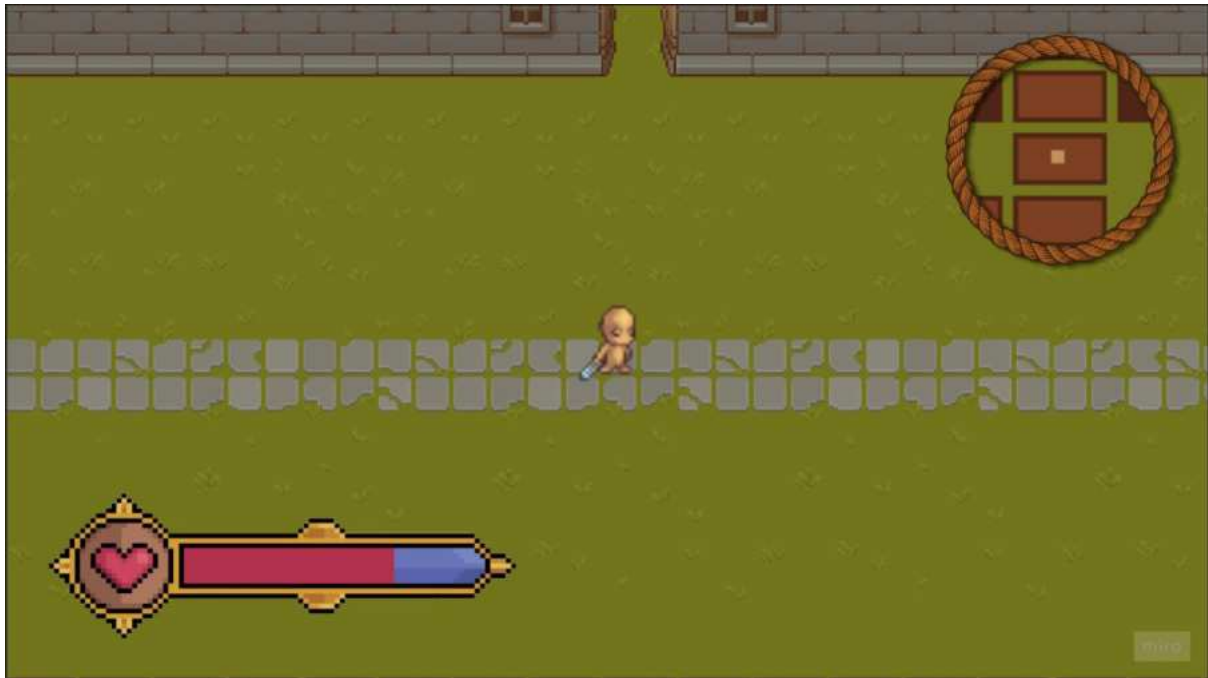
Figure 104 - Mini map added to user interface.

Based on the research that was done on user interfaces in chapter 2, it felt appropriate to implement a mini-map feature to the game's UI (Seen in Figure 104) Some basic research into tutorials and forum posts was conducted to find the best way to implement this idea. The method of using a secondary CineMachine paired with a new render texture and a new render layer was the decided to be the best way this implementation would work with the rest of the game's systems. A minimap icon was added to each item that would have been required to be seen on the map that was only able to be seen by the secondary CineMachine. This implementation worked perfectly.

```
void CheckRoomCleared()
{
    if (_enemyCount == 0 && _clearedIcon != null)
    {
        _clearedIcon.SetActive(true);
    }
}
```

Figure 105 - Small code addition for room clearing icon.

There was a small code update to the Room Manager (Seen in Figure 105) for the map to have better visual communication. When a room was cleared, a second icon would appear over the room to identify the room as being cleared to let players know that they have already been down that path, this was an attempt to improve the sense of direction which felt lacking in the smaller testing groups.



*Figure 106 - Updated health bar design.*

The next task to tackle was updating the health bar design to make it fit the visual aesthetic that the game had developed. After searching, a template was found that had two different sprite version available for a health bar and a mana bar. A prototype design was put together using photoshop before being imported into unity. The pre-existing health bar was then fitted into the new design template, the number of health was then removed as it was deemed as unnecessary screen clutter. The background of the health bar was taken from the mana bar template and worked very well visually. This can be seen in Figure 106.

### 5.7.8 Goal 6 – Updating Pause Menu

Updating the pause menu design was the next logical step after updating the user interface. Thanks to the wireframing that was mentioned previously, the layout and functionality for the pause menu was already made. All that had to be done was to update the font and the background to fit the design language that was being built within the game. The background for the pause menu was found using a third-party website and was slightly edited with the help of photoshop to make space for the pause menu text and buttons.



Figure 107 - Updated pause menu design.

The font was found on a website called DaFont which is known for its wide array of text fonts in many different styles. After finding the font and the background, they were imported into Unity and brought straight into the game scene, simply swapping out and game objects that stood there originally. The pause menu was tested after the game object designs were swapped and there were no issues found.

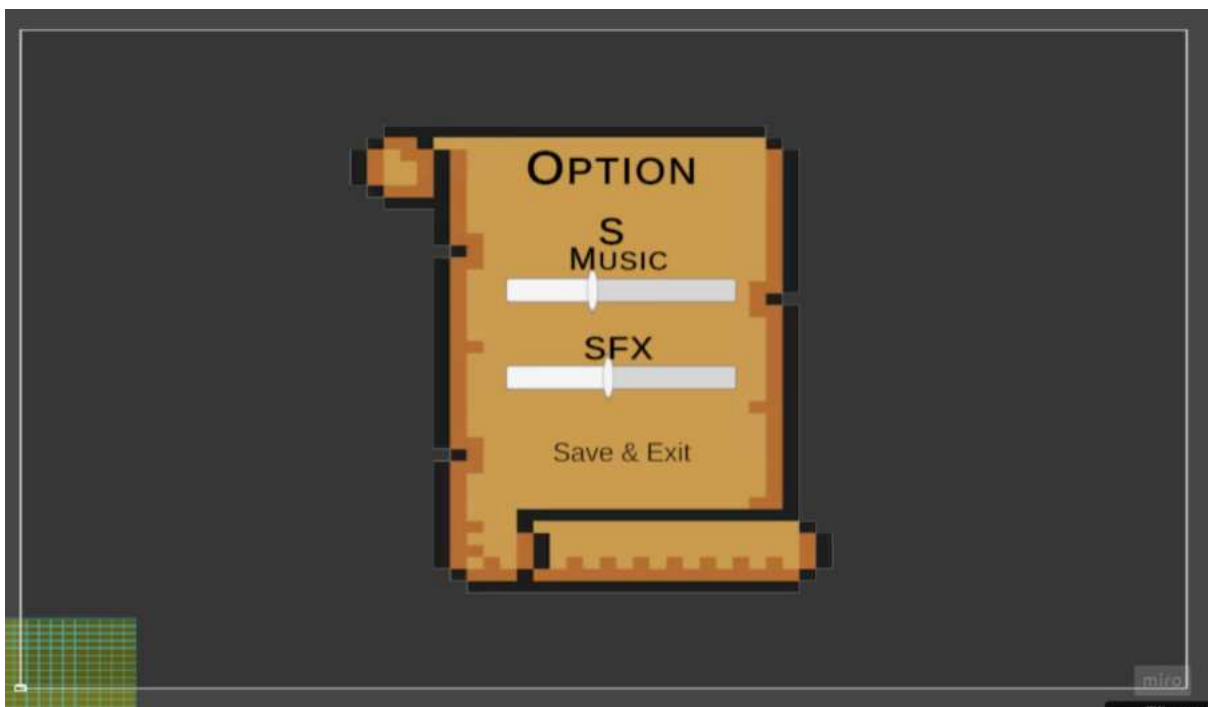


Figure 108 - Options menu addition.

A new panel was added to the user interface canvas to fit the options menu. This also required a new button to be added to the pause menu to be able to swap canvas items to access the options menu. The design philosophy was set when updating the pause menu so the options



menu followed the same philosophy. Sliders were added to the options menu to be able to control the volume of the game which is a feature that would be added later during this sprint. The save and exit button was added with functionality to close the options panel and re-open the pause menu panel, giving the player proper menu navigation. The updated designs can be seen in Figure 107 and Figure 108.

```

1  using UnityEngine;
2  using Unity.Cinemachine;
3  using UnityEngine.SceneManagement;
4  using UnityEngine.InputSystem;
5  using UnityEngine.EventSystems;
6
7  public class InterfaceManager : MonoBehaviour
8  {
9      [Header("Canvas Panels")]
10     [SerializeField] private CinemachineCamera _cineCam;
11     [SerializeField] private GameObject _pauseMenu;
12     [SerializeField] private GameObject _optionsMenu;
13     [SerializeField] private GameObject _loadingScreen;
14     [SerializeField] private GameObject _userInterface;
15
16     private GameObject _player;
17     public bool _isPaused = false;
18     private RoomManager _roomManager;
19     [SerializeField] private InputAction _pauseButton;
20     [SerializeField] private GameObject _musicSlider;
21     [SerializeField] private GameObject _resumeButton;
22
23     [Header("Interface Audio")]
24     [SerializeField] private AudioSource _interfaceAudioSource;
25     [SerializeField] private AudioClip _openMenuClip;
26     [SerializeField] private AudioClip _closeMenuClip;
27
28     void Awake()
29     {
30         Time.timeScale = 1f;
31     }
32

```

Figure 109 - Updated Interface manager.

The updated Interface Manager script (Seen in Figure 109) has a reformatted organizational structure to make better use of the header functionality within the Unity VsCode library. New serialized fields have been added to set the required game objects into their respective sections to make sure each object is getting the functionality it needs. There is also addition to some audio system game objects which will be explained in a later goal. Additional serialized fields were added, including a CinemachineCamera for the minimap camera handling and an InputAction controller support.



```

33     void Start()
34     {
35         _pauseMenu.SetActive(_isPaused);
36         _optionsMenu.SetActive(false);
37
38         _roomManager = FindObjectOfType<RoomManager>();
39         if (_cinCam == null)
40         {
41             _cinCam = GameObject.FindObjectOfType<CinemachineCamera>();
42         }
43
44         _player = GameObject.FindWithTag("Player");
45
46         if (_roomManager == null)
47         {
48             Debug.LogError("RoomManager not found in the scene!");
49         }
50
51         if (_player != null && _cinCam != null)
52         {
53             _cinCam.Follow = _player.transform;
54             _cinCam.LookAt = _player.transform;
55         }
56         else
57         {
58             Debug.LogWarning("Player or Cinemachine Camera not found!");
59         }
60     }
61
62     void OnEnable()
63     {
64         _pauseButton.Enable();
65     }
66

```

Figure 110 - Updated Interface manager.

The updated Start() method (Seen in Figure 110) has a variable set to the room manager which will be used in a function to work with the minimap functionality. It also locates the secondary Cinemachine camera which is used to handle the minimap. It then gets the player and uses the Cinemachine brain to follow the player, creating an accurate minimap that follows the player. The OnEnable() function is added to activate the \_pauseButton InputAction, making sure that the controller support is added to the script.

```

67     void OnDisable()
68     {
69         _pauseButton.Disable();
70     }
71
72     void Update()
73     {
74         if (_roomManager != null)
75         {
76             _loadingScreen.SetActive(!_roomManager.generationComplete);
77         }
78
79         _userInterface.SetActive(!_isPaused && !_loadingScreen.activeSelf);
80
81         if (Input.GetKeyDown(KeyCode.Escape) || _pauseButton.triggered)
82         {
83             TogglePauseMenu();
84         }
85     }
86
87     private void TogglePauseMenu()
88     {
89         _isPaused = !_isPaused;
90         _pauseMenu.SetActive(_isPaused);
91         _userInterface.SetActive(false);
92
93         PlayInterfaceSound(_isPaused);
94
95         _userInterface.SetActive(!_isPaused && !_loadingScreen.activeSelf);
96
97         // Pause the game when the menu is open
98         Time.timeScale = _isPaused ? 0f : 1f;
99     }
100

```

Figure 111 - Updated Interface manager.

The OnDisable() function (Seen in Figure 111) makes sure that if the Interface Manager script becomes inactive, the \_pauseButton will be disabled, this is more of safeguard to have in case of an issue with compiling. The update loop has been updated to look for the InputAction button as well as the escape key to toggle the pause menu, as well as continuously checking the state of the loading screen and pause menu to set the User Interface state accordingly. The TogglePauseMenu() function has been updated to integrate some of the audio functionality along with an additional setter for the user interface state to lessen the possibility of the system being confused.

```

101     public void ToggleOptionsMenu(bool open)
102     {
103         _pauseMenu.SetActive(!open);
104         _optionsMenu.SetActive(open);
105
106         if (open)
107         {
108             EventSystem.current.SetSelectedGameObject(_musicSlider);
109         }
110         else
111         {
112             EventSystem.current.SetSelectedGameObject(_resumeButton);
113         }
114     }
115
116     private void PlayInterfaceSound(bool isPaused)
117     {
118         if (_interfaceAudioSource != null)
119         {
120             AudioClip clipToPlay = isPaused ? _openMenuClip : _closeMenuClip;
121             _interfaceAudioSource.PlayOneShot(clipToPlay);
122         }
123     }
124
125     public void ResumeGame()
126     {
127         _isPaused = false;
128         _pauseMenu.SetActive(false);
129
130         PlayInterfaceSound(_isPaused);
131         _userInterface.SetActive(!_loadingScreen.activeSelf);
132         Time.timeScale = 1f;
133     }
134
135     public void QuitGame()
136     {
137         SceneManager.LoadScene("MainMenu");

```

Figure 112 - Updated Interface manager.

The newly added `ToggleOptionsMenu()` function (Seen in Figure 112) handles the opening and closing of the options menu, much like the `TogglePauseMenu()` function. This function also makes sure that the event system is configured to enable controller support for menu navigation, it does this by making sure the Resume button or the Music Slider are selected when the menu is opened, using that as a start point for navigation. The `ToggleOptionsMenu()` function makes sure to close the pause menu before opening the options menu with a clever use of booleans, setting one to the opposite of the other.

### 5.7.9 Goal 7 – Main Menu Implementation

After implementing the new pause menu design and option menu functionality, it was time to move onto the Main Menu implementation. The game needed a way for the room generation scene to be reset after the player had died, so creating a main menu to switch into before loading back into the room generation scene made the most logical sense. Thanks to the

wireframing that was done previously along with the design philosophy that is being realised, creating the design for the menu was far easier.



*Figure 113 - Main menu added.*

The background art for the main menu was sourced from a third-party website which specialised in copyright free, ai generated artworks, this specific piece of art was prompted by another user and available on the website. This specific piece of art was taken as it fit very well with the visual aesthetic of the game, along with the colours that had been used, it was simply too perfect not to use.

The scroll asset for the updated pause menu design was reused as it fit well as a background for the title of the game. The buttons for the menu were made without a template in mind, but instead were made just adhering to the visual style of the menu, making sure the colours chosen would reflect the style appropriately while still having enough contrast to be legible.

When collecting screenshots of the pause menu and main menu, a visual glitch with the unity engine occurred which reverted the chosen font (seen in Figure 107) back to the default font provided by the engine (seen in Figure 113). This glitch was not visible in the final build of the game and only occurred after the major testing phase.





*Figure 114 - Options menu added.*

When adding the options menu (Seen in Figure 114), the same background game object was used for the options menu, along with the same slider components that we created for the Music and SFX sliders in the options menu available in the pause menu, and the button components that were designed for the main menu, this created a consistent design between the options menu and the main menu.

```

1  using UnityEngine;
2  using UnityEngine.UI;
3  using UnityEngine.SceneManagement;
4  using UnityEngine.EventSystems;
5  using TMPro;
6
7  public class MainMenu : MonoBehaviour
8  {
9      [Header("Canvas Panels")]
10     [SerializeField] private GameObject _mainPanel;
11     [SerializeField] private GameObject _optionsPanel;
12
13     [Header("Buttons")]
14     [SerializeField] private GameObject _newGameButton;
15     [SerializeField] private GameObject _musicSlider;
16
17     [Header("Button Text")]
18     [SerializeField] public TMP_Text _newText;
19     [SerializeField] public TMP_Text _optionText;
20     [SerializeField] public TMP_Text _exitText;
21     [SerializeField] public TMP_Text _saveText;
22
23     [Header("Audio Settings")]
24     [SerializeField] private AudioSource _audioSource;
25     [SerializeField] private AudioClip _backgroundMusic;
26
27     private EventSystem _eventSystem;
28
29     void Awake()
30     {
31         _eventSystem = EventSystem.current;
32         _mainPanel.SetActive(true);
33         _optionsPanel.SetActive(false);
34         SetTextColors();
35         PlayMenuMusic();
36         SetSelectedButton(_newGameButton);
37     }

```

Figure 115 - Main menu script.

The newly created Main Menu script (Seen in Figure 115) handles all the functionality for the main menu, being heavily influenced by the pre-existing Interface Manager script which is used heavily within the Room Generation scene, which can be seen in the way the code is structured in Figure 115.

```
39     public void LoadRoomGeneration()  
40     {  
41         SceneManager.LoadScene("RoomGeneration");  
42     }  
43  
44     public void OpenOptions()  
45     {  
46         _mainPanel.SetActive(false);  
47         _optionsPanel.SetActive(true);  
48         SetSelectedButton(_musicSlider);  
49     }  
50  
51     public void CloseOptions()  
52     {  
53         _optionsPanel.SetActive(false);  
54         _mainPanel.SetActive(true);  
55         SetSelectedButton(_newGameButton);  
56     }  
57  
58     public void ExitGame()  
59     {  
60         Application.Quit();  
61     }  
62  
63     private void SetTextColors()  
64     {  
65         ChangeTextColor(_newText);  
66         ChangeTextColor(_optionText);  
67         ChangeTextColor(_exitText);  
68         ChangeTextColor(_saveText);  
69     }
```

Figure 116 - Main menu script.

The LoadRoomGeneration() function (Seen in Figure 116) handles the scene transition from the main menu to the Room Generation scene using Unity's scene management library, which is used as the gameplay centre for the game. This method works very well and did not need any modifications during testing.

The OpenOptions() function handles the transition between the main menu panel and the options menu panel. When activated the function deactivates the main menu panel while simultaneously activating the options menu panel, also setting the selected button to the music slider to make sure controller menu navigation was supported.



The CloseOptions() function works in parallel to the OpenOptions() function, reactivating the main menu panel and setting the selected button as the new game button to ensure that the controller navigation wasn't nullified when transitioning through the panels.

The ExitGame() function is quite simple and self-explanatory, shutting down the application. This function doesn't work within editing mode so this was not tested until the major testing phase.

The SetTextColors() function was written to fix a bug that was occurring with the TextMeshPro components where either every text element in the game would change or they wouldn't change at all, but this was resolved by brute forcing the colour change with the implementation of this function.

```

71     private void ChangeTextColor(TMP_Text text)
72     {
73         if (text != null)
74         {
75             text.enableVertexGradient = false;
76             text.color = Color.white;
77
78             Material newMaterial = new Material(text.fontMaterial);
79             newMaterial.SetColor("_FaceColor", Color.white);
80             text.fontMaterial = newMaterial;
81         }
82     }
83
84     private void PlayMenuMusic()
85     {
86         if (_audioSource != null && _backgroundMusic != null)
87         {
88             _audioSource.clip = _backgroundMusic;
89             _audioSource.loop = true;
90             _audioSource.Play();
91         }
92     }
93
94     private void SetSelectedButton(GameObject button)
95     {
96         if (_eventSystem != null && button != null)
97         {
98             _eventSystem.SetSelectedGameObject(button);
99         }
100    }
101 }

```

Figure 117 - Main menu script.

The ChangeTextColor() function (Seen in Figure 117) is written to work in conjunction with the SetTextColors() function. This function configures the TextMeshPro components by creating a

new material and applying it to the chosen text components to negate any material conflicts and successfully changing the text colours.

The PlayMenuMusic() functions work with the audio system, handling the background music for the main menu while making sure it loops in case the player is at the menu for a longer period of time, while also providing null checks as a safeguard.

The SetSelectedButton() function is the most crucial function in making sure that controller support is available for menu navigation, making sure that there is a starting point for the controller navigation system so that the user can navigate at their pleasure. Without this function, the controller would not have a starting point and therefore would not be able to navigate the menus at all.

### 5.7.10 Goal 8 – Audio system Implementation

The implementation was the next step in the development process. With the major testing phase rapidly approaching, it was necessary to get the sound system implemented at this stage. All the sound effects and background music were selected at this stage, all that had to be done is set up the mixer and the correlating scripts.

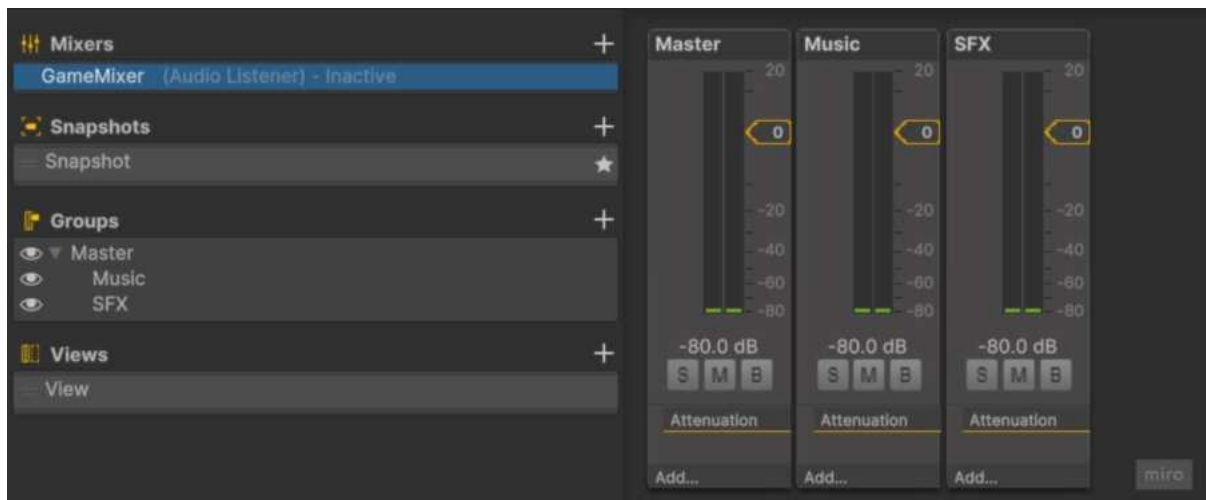


Figure 118 - Game audio mixer.

Setting up the audio mixer was very simple. The only modification to the audio mixer that was needed was to split the Master volume group into two sections that would be able to handle the sound effects and the background music dynamically. This can be seen in Figure 118.

```

1  using UnityEngine;
2  using UnityEngine.Audio;
3  using UnityEngine.UI;
4
5  public class VolumeSettings : MonoBehaviour
6  {
7      [SerializeField] private AudioMixer _myMixer;
8      [SerializeField] private Slider _musicSlider;
9      [SerializeField] private Slider _sfxSlider;
10
11     private void Start()
12     {
13         if (PlayerPrefs.HasKey("musicVolume"))
14         {
15             LoadVolume();
16         }
17         else
18         {
19             SetMusicVolume();
20             SetSFXVolume();
21         }
22     }
23
24     public void SetMusicVolume()
25     {
26         float volume = _musicSlider.value;
27         _myMixer.SetFloat("MusicParam", Mathf.Log10(volume) * 20);
28         PlayerPrefs.SetFloat("musicVolume", volume);
29     }

```

Figure 119 - Volume settings script.

The creation of the Volume Settings script (Seen in Figure 119) was quite simple as it followed the same steps that were taken for a previous project. Starting with gaining access to the audio mixer that was created by creating a serialized variable for it to be placed in, along with the sliders for each version of the option menus. Firstly, the script checks the PlayerPrefs for saved volume settings, loading these values in if they exist, and setting these values to default if they do not.

The SetMusicVolume() converts the music sliders from a linear value to a decibel-based mixer attenuation logarithmically, allowing the slider to interface with the audio mixer's decibel system all while PlayerPrefs saves the raw slider value. This same system is introduced in the SetSFXVolume() function just being saved under different PlayerPrefs to make sure that the systems are consistent yet different.

```
30
31     public void SetSFXVolume()
32     {
33         float volume = _sfxSlider.value;
34         _myMixer.SetFloat("SfxParam", Mathf.Log10(volume) * 20);
35         PlayerPrefs.SetFloat("sfxVolume", volume);
36     }
37
38     private void LoadVolume()
39     {
40         _musicSlider.value = PlayerPrefs.GetFloat("musicVolume");
41         _sfxSlider.value = PlayerPrefs.GetFloat("sfxVolume");
42
43         SetMusicVolume();
44         SetSFXVolume();
45     }
46 }
```

Figure 120 - Volume settings script.

The Load Volume() function (Seen in Figure 120) retrieves the saved volume settings for both the music and the sound effects and applies them to the sliders in the options menu. It then calls the set music and set sound effects functions to set them to the saved values, creating a cohesive sound settings system.

```

1  using UnityEngine;
2  using UnityEngine.SceneManagement;
3
4  public class AudioManager : MonoBehaviour
5  {
6      [Header("Audio Source")]
7      [SerializeField] AudioSource _musicSource;
8      [SerializeField] AudioSource _sfxSource;
9
10     [Header("Audio Clip")]
11     public AudioClip _background;
12     public AudioClip _menuBackground;
13
14     private void Start()
15     {
16         string sceneName = SceneManager.GetActiveScene().name;
17
18         if (sceneName == "RoomGeneration")
19         {
20             _musicSource.clip = _background;
21         }
22         else if (sceneName == "MainMenu")
23         {
24             _musicSource.clip = _menuBackground;
25         }
26
27         if (_musicSource.clip != null)
28         {
29             _musicSource.loop = true;
30             _musicSource.Play();
31         }
32     }
33

```

Figure 121 - Audio manager script.

The Audio Manager script (Seen in Figure 121) is primarily in charge of the background music for the gameplay and main menu scenes. As is evident by the scene-specific handling for the background music. The music clip that is chosen for either of the two available scenes is chosen based on the scene name that is currently chosen.

```

34     public void PlayLoopingSFX(AudioClip clip)
35     {
36         if (_sfxSource.clip != clip)
37         {
38             _sfxSource.clip = clip;
39             _sfxSource.loop = true;
40             _sfxSource.Play();
41         }
42     }
43
44     public void StopLoopingSFX()
45     {
46         if (_sfxSource.loop)
47         {
48             _sfxSource.Stop();
49             _sfxSource.clip = null;
50             _sfxSource.loop = false;
51         }
52     }
53 }

```

Figure 122 - Audio manager script.

The PlayLoopingSFX() function (Seen in Figure 122) provides a controlled playback of chosen sound effects. It is a very simple function that verifies if the clip exists and isn't already looping before making sure that it is set to loop before playing the audio clip. The StopLoopingSFX() exists to stop the loop of anything that is currently looping by nullifying the clip and setting the loop to false.

```

[Header("Player Audio")]
[SerializeField] private AudioSource _playerWalkSource;
[SerializeField] private AudioSource _playerDamageSource;
[SerializeField] private AudioClip _walkClip;
[SerializeField] private AudioClip _hurtClip;

```

Figure 123 - Additions to player movement script for player audio.

The Player Movement was updated (Seen in Figures 123 until 128) to integrate the newly made audio system, adding new audio sources and clips that the player movement system needs.



```
if (!_playerWalkSource.isPlaying)
{
    _playerWalkSource.clip = _walkClip;
    _playerWalkSource.loop = true;
    _playerWalkSource.Play();
}
```

*Figure 124 - Additions to player movement script for player audio.*

The player walk source and walk clip are controlled inside the HandleMovement() function, playing the walking sound effect from the walk source only when the walking state is set to true. This proximity to the animation state handling creates an accurate system, making sure that when the player is visually moving across the screen, the correct sound effect is playing as well. Each important sound effect getting its own source makes sure that different sounds can play at the same time without cancelling each other out which creates a more immersive gameplay experience.



```
private void SetIdleState()
{
    // Set Idle to true when not moving
    _animator.SetBool("Idle", true);
    _animator.SetBool("WalkDown", false);
    _animator.SetBool("WalkLeft", false);
    _animator.SetBool("WalkRight", false);
    _animator.SetBool("WalkUp", false);
    _animator.SetBool("Hurt", false);

    if (_playerWalkSource.isPlaying)
    {
        _playerWalkSource.Stop();
    }
}
```

Figure 125 - Additions to player movement script for player audio.

The SetIdleState() function has also been updated to add stop the walking sound effects when the player has stopped moving. This script works the same way as the HandleMovement(), handling the animation state and the sound at the same time, creating a consistent visual and audio experience for the player.

```
private void PlayHurtSound()
{
    if (_playerDamageSource != null && _hurtClip != null)
    {
        _playerDamageSource.PlayOneShot(_hurtClip);
    }
}
```

Figure 126 - Additions to player movement script for player audio.

The PlayHurtSound() function is placed inside the player's knockback function, playing the hurt sound briefly. The placement of this function follows the same idea as the SetIdleState() and HandleMovement() functions, keeping the visual and audio responses of the game consistent.

```
[Header("Player Audio")]
[SerializeField] private AudioSource _playerAttackSource;
[SerializeField] private AudioClip _attackClip;
```

Figure 127 - Additions to player melee script for player audio.

After implementing and testing the walk and damage noises, it was time to move onto implementing the attack noises for the player inside the Player Melee script. The setup for this audio integration had the same approach as previous audio integrations where the necessary audio source and clip were added to the script with the use of serialized fields.

```
private void PlayAttackSound()
{
    if (_playerAttackSource != null && _attackClip != null)
    {
        _playerAttackSource.PlayOneShot(_attackClip); // Play the attack sound
    }
}
```

Figure 128 - Additions to player melee script for player audio.

The PlayAttackSound() function was constructed and implemented the same way as the PlayHurtSound() function, where the attack sound is played for a brief period and the function is placed in the same area of the script which handles the animation states to continue to create a consistent visual and audio experience.

```
[Header("Enemy Audio")]
[SerializeField] private AudioSource _enemyWalkSource;
[SerializeField] private AudioSource _enemyDamageSource;
[SerializeField] private AudioClip _walkClip;
[SerializeField] private AudioClip _hurtClip;
[SerializeField] private AudioClip _deathClip;
```

Figure 129 - Additions to enemy script for enemy audio.

The audio integration for the enemy (Seen in Figures 129 until 133) was the last game object that needed to be added. This implementation process took a little bit longer as the enemy script had all its functionality inside of one script, so more time and care had to be taken to make sure every sound effect was being placed in the right section.

The process started by creating serialized fields for every necessary audio source and audio clip that the enemy would need and then beginning to move these clips into their required sections.

```

private void SetIdleState()
{
    _currentState = EnemyState.Idle;

    if (_enemyWalkSource.isPlaying)
    {
        _enemyWalkSource.Stop();
    }

    // Disable all movement animations
    _animator.SetBool("WalkUp", false);
    _animator.SetBool("WalkDown", false);
    _animator.SetBool("WalkLeft", false);
    _animator.SetBool("WalkRight", false);

    // Trigger the universal idle animation
    _animator.SetBool("Idle", true);
}

private void SetWalkingState(Vector3 movementDelta)
{
    _currentState = EnemyState.Walking;

    if (!_enemyWalkSource.isPlaying)
    {
        _enemyWalkSource.clip = _walkClip;
        _enemyWalkSource.loop = true;
        _enemyWalkSource.Play();
    }

    _animator.SetBool("Idle", false);
}

```

Figure 130 - Additions to enemy script for enemy audio.

The walking animation sound effect was set up the same way as the player's walking sound effect where the enemy walking sound effect would play when the enemy animations began to play and then would stop when the idle animation state was activated. This method was used again due to how well it worked when implementing the player's sound effects.

```
public void TakeDamage(float _damage)
{
    _health -= _damage;

    if (_health <= 0 && _currentState != EnemyState.Dead)
    {
        _currentState = EnemyState.Dead;
        _animator.SetBool("Hurt", false);
        _animator.SetBool("Dead", true);
        PlayDeathSound();

        // Stop all movement animations
        _animator.SetBool("WalkUp", false);
        _animator.SetBool("WalkDown", false);
        _animator.SetBool("WalkLeft", false);
        _animator.SetBool("WalkRight", false);
        _animator.SetBool("Idle", false);
    }
}
```

Figure 131 - Additions to enemy script for enemy audio.

The death sound was placed in the damage handling section of the script as this is also where the enemy death logic is held, so this section was the most appropriate, and during testing proved to work as intended.



```

public void Knockback(Transform t)
{
    if (!_agent.enabled) return;

    Vector3 _direction = (_center.position - t.

    _knocked = true;
    _agent.isStopped = true;
    _agent.velocity = _direction * _knockbackVe

    // Enable Hurt state
    _animator.SetBool("Hurt", true);
    PlayHurtSound();
}

```

Figure 132 - Additions to enemy script for enemy audio.

The hurt sound was set to be played when the enemy is knocked back which also triggers the hurt animation, creating a consistent response from the enemy when they are damaged.

```

private void PlayHurtSound()
{
    if (_enemyDamageSource != null && _hurtClip != null)
    {
        _enemyDamageSource.PlayOneShot(_hurtClip);
    }
}

private void PlayDeathSound()
{
    if (_enemyDamageSource != null && _deathClip != null)
    {
        _enemyDamageSource.PlayOneShot(_deathClip);
    }
}
}

```

Figure 133 - Additions to enemy script for enemy audio.

The hurt and death sound functions were built the same way as the sound functions from the player's sound implementation due to how well this system worked the first time, simply checking if the sound sources and clips are not null and then briefly playing the chosen sound clip.

### 5.7.11 Goal 9 – Controller Support

The final step before getting ready for the major user testing phase was to implement controller support for the game. This was done after some advice was given by a lecturer about making it easier for testers to be able to just pick up and play the game and to have less of a learning curve with the controls.

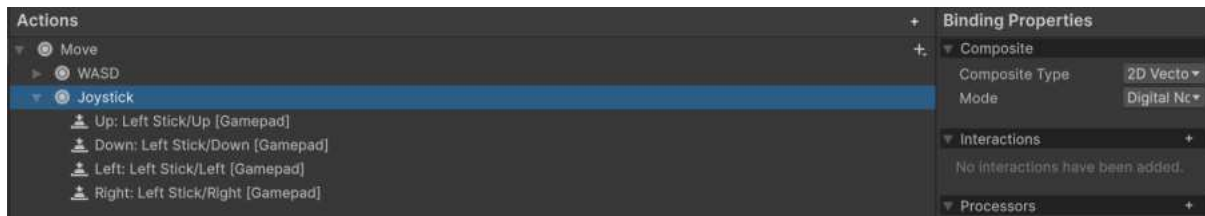


Figure 134 - New input system additions for controller support.

The player movement was very simple to integrate thanks to the New Input System in the unity engine (Seen in Figure 134). A secondary input was created for the move action with the cardinal directions of the left joystick and thanks to the input manager script and the fact that the values for both keyboard and controller were of the same type, this was the only addition that had to be made for the player movement to have controller support. There were some slight modifications to the animation transition sensitivity threshold to make sure that the correct animations were playing when the player was moving horizontally but it was a very minor update.

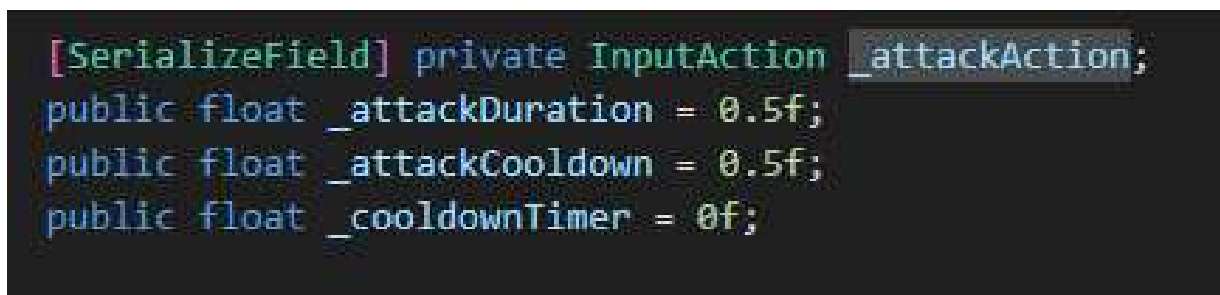


Figure 135 - Additions to player melee script for controller support.

The implementation of controller support for the player combat was nearly just as simple as the player movement (Seen in Figures 135 and 136). A new input action variable was added to the script which can be set to any button value thanks to the listen feature that this variable has. This variable is how controller support was added for the player melee script.

```

void Update()
{
    if (_cooldownTimer > 0f)
    {
        _cooldownTimer -= Time.deltaTime;
    }

    if ((_attackAction.triggered || Input.GetKeyDown(KeyCode.Mouse0)) && _cooldownTimer <= 0f && _interfaceManager._isPaused == false)
    {
        Attack();
        PlayAttackSound();
    }
}

```

Figure 136 - Additions to player melee script for controller support.

The input condition for the player attack was also updated to adhere to the new change which was quite simple thanks to the implementation of the unity input action variable. Now the script is checking for both keyboard and controller inputs, allowing players to choose the controller method they feel most comfortable with and being able to change the controller scheme at a moment's notice.

```

private GameObject _player;
public bool _isPaused = false;
private RoomManager _roomManager;
[SerializeField] private InputAction _pauseButton;
[SerializeField] private GameObject _musicSlider;
[SerializeField] private GameObject _resumeButton;

```

Figure 137 - Additions to interface manager script for controller support.

The interface manager was updated (Seen in Figures 137 and 138) the same way with the implementation of the input action component for controller support. This time being set to a different button towards the centre of the controller to make sure that there was no input confusion.



```

void Update()
{
    if (_roomManager != null)
    {
        _loadingScreen.SetActive(!_roomManager.generationComplete);
    }

    _userInterface.SetActive(!_isPaused && !_loadingScreen.activeSelf);

    if (Input.GetKeyDown(KeyCode.Escape) || _pauseButton.triggered)
    {
        TogglePauseMenu();
    }
}

```

Figure 138 - Additions to interface manager script for controller support.

The Update() loop was updated in a similar way to the player melee script where the input condition to open the pause menu is now looking for the original keyboard input method or the new pause button that would have been set in the Unity engine inspector. Keeping both keyboard and controller input options available to the player at all times.

## 5.8 Sprint 7

### 5.8.1 Goals

- Complete a first draft thesis report before Easter break.
- Receive and apply feedback given by thesis supervisor.

### 5.8.2 Goal 1 – First Draft of Report

The first draft of this report was not written in order from chapter 1 to 8. Instead, chapters were written out of sequence to help figure out how they would connect and flow together. Writing started with a chapter that had a lot of content and didn't rely heavily on the others, which ended up being chapter 6, which focused on user testing. This chapter was written first because the testing had just been completed, so the information was fresh and easy to write about.

After chapter 6, work moved to chapter 1, the introduction. One section in the introduction needed a short overview of user testing, so content from chapter 6 was used and adjusted to fit. This same approach continued throughout the report, with some sections left unfinished until feedback was received on structure and layout.

Chapter 5 was the only chapter postponed on purpose because it was longer and more time-consuming. Finishing the shorter chapters first meant getting quicker feedback, which helped improve the rest of the report. Even though chapter 5 didn't get much feedback directly, the feedback from other chapters helped shape its final version. Overall, this writing method turned out to be the most effective way to develop the first draft.

### 5.8.3 Goal 2 – Receive Feedback

Valuable feedback was received from the thesis supervisor regarding the first draft of the report. While much of the content was well-received, several recommendations were provided to improve the overall flow and reduce the repetitive text within the report. Suggestions included reformatting sections of the design and research chapters and referencing figures from other chapters where relevant to the current discussion. Additional advice focused on smaller improvements to enhance the reader's experience, such as adopting a different style for figure titles, including a figure table beneath the table of contents, and slightly adjusting the APA referencing style to better accommodate sources such as YouTube videos. Guidance was also given on restructuring parts of chapter 5 to improve readability and reduce content congestion. Each piece of feedback was highly relevant and directly contributed to strengthening the clarity and impact of the report.

## 6 Testing

### 6.1 Introduction

This chapter presents the full testing process carried out during the project's development, beginning with functional testing where each game mechanic and feature was manually tested to ensure it behaved as intended. It then moves into the user testing phase, which combines feedback and observations from both small focus group sessions and a larger round of testing. Player input was collected through surveys, gameplay feedback, and direct observation, offering valuable insight into how users interacted with the game's systems, controls, and overall experience. The chapter concludes with a breakdown of this data and the key suggestions received, highlighting what was learned from the testing process and how it helped shape the final outcome of the game.

### 6.2 Functional Testing

#### 6.2.1 Menu Navigation

Test No	Description of test case	Expected Output	Actual Output	Comment
1	From the Main Menu. Start a new game.	When the new game button is pressed, the game scene is loaded.	The game scene is loaded.	This mechanic works correctly.
2	From the Main Menu. Exit the game.	When the exit button is pressed, the application would quit.	The game did not quit.	There is a bug in the main menu script, but the game can still be exited with the Alt+F4 command.
3	From the Main Menu. Enter the Options Menu.	When the Options button is pressed, the options panel is activated.	The options panel is activated.	This mechanic works correctly.
4	From the Options Menu. Change the Music volume.	For the music slider to adjust the music volume.	The music is adjusted when the slider moves.	This mechanic works correctly.
5	From the Options Menu, change the SFX volume.	For the SFX slider to adjust the volume.	The SFX is adjusted when the slider moves.	This mechanic works correctly.
6	From the Options Menu. Exit to the Main Menu.	For the Main Menu panel to be activated.	The Main Menu Panel is activated.	This mechanic works correctly.
7	Enter the Pause Menu.	The Pause Menu panel is activated	The Pause Menu panel is activated.	This mechanic works correctly.

		when pause button is pushed.		
8	From the Pause Menu. Resume the game.	When the Resume button is pressed, time continues.	The game time continues.	This mechanic works correctly.
9	From the Pause Menu. Exit the game.	The scene changes to the Main Menu when the Exit game button is pressed.	The scene changes to the main menu.	This mechanic works correctly.
10	From the Pause Menu. Enter the Options Menu.	Switch to the options panel when the options button is pressed.	The panel switches when the button is pressed.	This mechanic works correctly.
11	From the Options Menu. Change the Music Volume.	When the slider moves the music volume changes.	When the slider moves the music volume changes.	This mechanic works correctly.
12	From the Options Menu. Change the SFX Volume.	When the slider moves the SFX volume changes.	When the slider moves the SFX volume changes.	This mechanic works correctly.
13	From the Options Menu. Exit to the Pause Menu.	The panel changes to when the Save & Exit button is pressed.	The panel changes to the pause menu when the Exit button is pressed.	This mechanic works correctly.

### 6.2.2 Player Controls

Test No	Description of test case	Expected Output	Actual Output	Comment
1	Player Movement.	For the player to move based on the directional button.	The player moved the character around the screen.	This mechanic works correctly.
2	Player Combat.	For the player to attack based on their direction.	The player attacked in direction of the last movement.	This mechanic works correctly.
3	Attack an Enemy.	For the enemy to be attacked and then die.	The enemy died after multiple hits.	This mechanic works correctly.

### 6.2.3 Enemy Interaction

Test No	Description of test case	Expected Output	Actual Output	Comment
1	Enemy pathfinding test.	When player enters the room, the enemy follows the player, calculating the shortest path.	The enemy successfully follows the player.	The enemy has obstacle detection implemented but the rooms do not have obstacles, making testing harder.
2	The enemy attacks the player.	When the enemy touches the player collider, the player loses health.	The player loses health when the enemy touches them.	This mechanic works perfectly.
3	Enemy death.	When the enemy's health hits zero, the enemy is destroyed from the scene.	The enemy was removed from the scene when their health hit zero.	This mechanic works perfectly.

### 6.2.4 Map Navigation

Test No	Description of test case	Expected Output	Actual Output	Comment
1	Wall Collider test	When the player or enemy run into a wall, they will be stopped.	The player and enemy cannot get past the wall.	This mechanic works perfectly.
2	Room Neighbour Test	When the room generates, doors will only open when there are rooms on the other side.	The doors only open when there is a room on the other side of them.	This mechanic works perfectly.
3	Door Open Test	When the room enemy count has hit 0, the doors to neighbour rooms will open.	When the room enemy count has hit 0, the doors to neighbour rooms will open.	This mechanic works perfectly.
4	Regeneration Test	When the total enemy count has hit 0, the world will regenerate, and the player will be back in the start room.	The world regenerates when the enemy count hits 0.	This mechanic works perfectly.

## 6.3 User Testing

### 6.3.1 Player Controls

#### Movement

When testing with mouse and keyboard controls, results varied depending on the user's experience with PC gaming. Those with prior experience had a much easier time understanding the control scheme, with some needing no instructions at all. Users without PC gaming experience faced a higher learning curve but were still able to understand the controls after minimal guidance.

When testing with a gamepad, the results were more consistent. Most players quickly understood that the joystick controlled movement. A small number of users with no gaming experience at all needed brief instructions, but once given, they were able to continue without further help.

How would you rate the player movement? (1 Poor - 5 Excellent)

7 responses

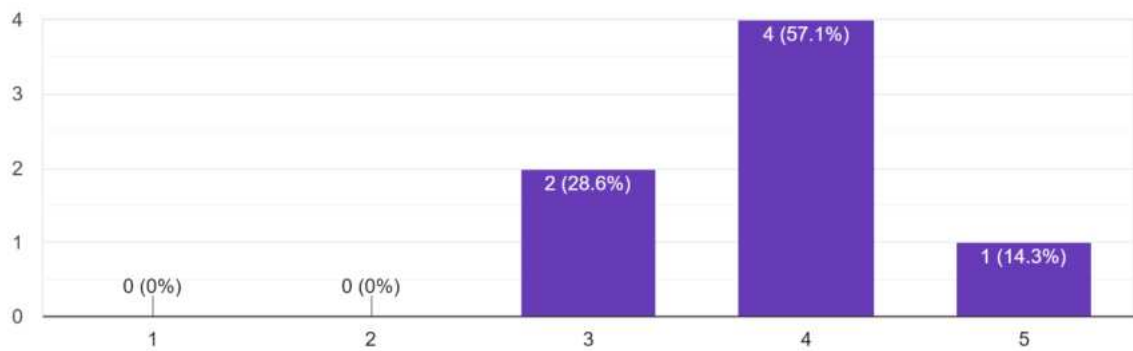


Figure 139 - Survey Rating on Player Movement

Any suggestions on improving player movement?

4 responses

- There was one minor bug with invisible walls on the north door - fix that and its solid!
- Dodge? Player knockback reduction
- A roll feature would be amazing
- It could feel a bit more fluid but it was nice

Figure 140 - Survey Suggestions on Player Movement

Combat

Did you think the player combat felt adequate? (1 Poor - 5 Excellent)  
7 responses

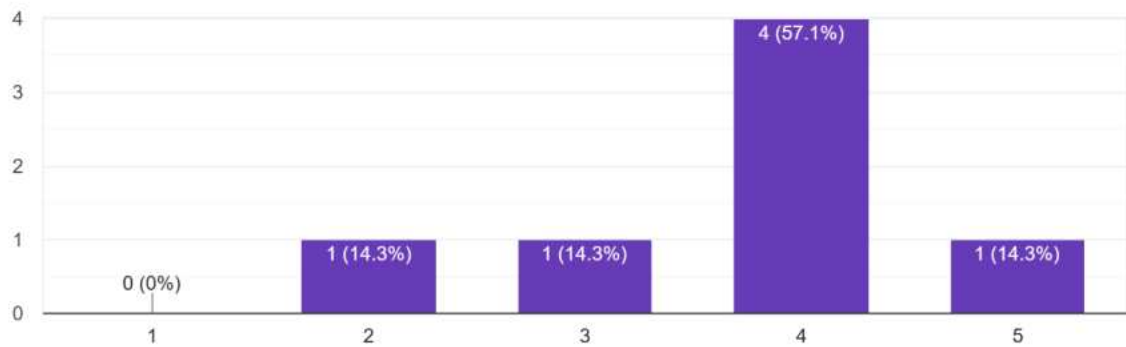


Figure 141 - Survey Rating on Player Combat.

When testing with keyboard and mouse, many users were able to figure out the combat system before receiving any instructions, regardless of their previous PC gaming experience. The simplicity of clicking on the screen to attack helped remove any potential learning barriers.

During gamepad testing, most users also identified the combat button without instruction. Some needed a bit of guidance after experimenting with different buttons, but there was a clear understanding that a combat system was present in the game.

Any suggestions on player combat improvements?  
3 responses

- For now it was excellent, bits of variance and room obstructions will really sell it
- Dodge perhaps, one or two more abilities
- The hit box was confusing, but overall good combat

Figure 142 - Survey Suggestions on Player Combat.

There was some confusion around where the player’s hitbox would appear during attacks. Most players picked it up quickly, but a few struggled to understand the exact damage area.

Confusion around attack direction also came up due to the lack of directional idle animations. This issue was noticeable during both keyboard and mouse testing, as well as with gamepad use. When the player was moving and attacking, there was clear confidence in the direction of the hit. However, when attacking from an idle position, players were less sure—even though the game saved the last movement direction to determine the attack. The idle animation facing a different direction often caused misunderstanding. Once the system was explained, players understood it better, but the initial confusion highlights that the system lacks intuitive clarity.



## 6.3.2 Enemy Interaction

### Pathfinding

How would you rate the enemy movement? (1 Poor - 5 Excellent)

7 responses

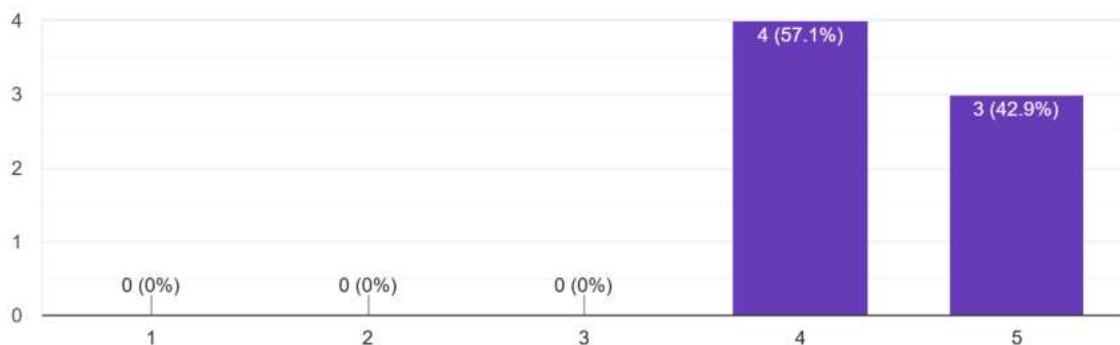


Figure 143 - Survey Rating on Enemy Movement.

User testing results revealed both strengths and weaknesses in the enemy pathfinding system. In one-on-one encounters, the system performed well. However, as more enemies were added to the room, issues became more noticeable.

It was common for the enemies to follow the same path and block each other instead of finding ways around. Another issue occurred when the player became cornered, there was no way to escape, causing a stun lock effect until the player's health ran out. Additionally, a bug was found where an enemy would spawn too close to the door, preventing the player from entering the room. This will need to be adjusted to avoid the issue in future versions.

Any suggestions on improving enemy movement?

4 responses

The only thing that would make it even cooler is adding variance - have enemies move slower and quicker by room and it'll add challenge!

Nothing unless different enemies are added

At one point, with four slimes, I had gotten stun locked in a corner

It was move fluid than my player but it was quick

Figure 144 - Survey Improvements on Enemy Movement.

There was some great feedback on the enemy pathfinding system, including the idea of adding speed variance between enemies. This would help reduce enemies grouping together and would also add more variety to each room. A programmer from Desk Rage also suggested using dedicated spawn points instead of random ones, a very helpful recommendation.

## Combat

User testing results for enemy combat were generally positive. Both the enemy and player knockback systems functioned as intended, and the player's health decreased correctly when colliding with the enemy's hitbox. Feedback on enemy combat was minimal, with other developers noting that the enemy's combat style made sense and felt natural. The most common suggestion involved the length of time the player was stunned after hitting an enemy. Some felt it was too long and made the survival more frustrating than fun. As a result, the knockback duration may be slightly reduced to see if it improves gameplay. This suggestion came from one of fifty players.

### 6.3.3 Game World

#### Map Navigation

Some useful feedback was gathered during testing for map navigation, and several ideas came up by watching how players interacted with the game world. The main suggestions included adding more content to rooms, introducing unique objects to help tell rooms apart, designing new room layouts, and creating a key-and-door system that lets players choose their path instead of following a set one.

It was also observed that some players assumed all doors were interactive and often ran into closed ones while ignoring the open ones. To address this, the design of the open and closed doors will be adjusted to make them easier to tell apart.

How did you feel about the game world? (1 Poor - 5 Excellent)

7 responses

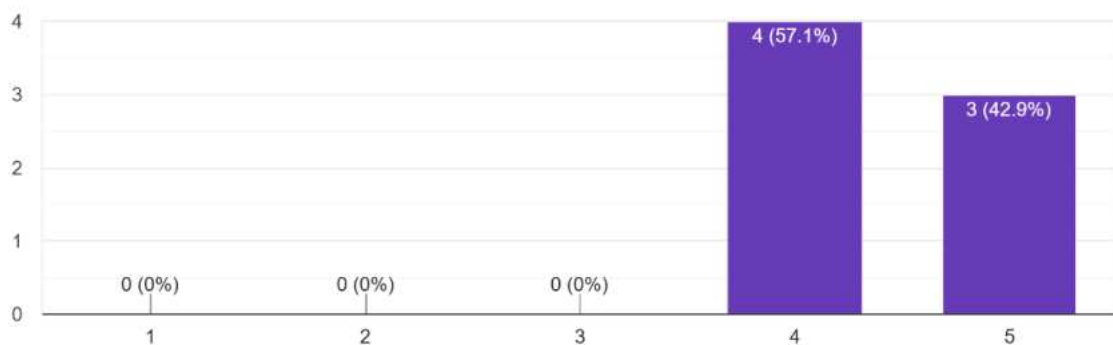


Figure 145 - Survey Rating of Game World.

Any suggestions on game world improvements?

4 responses

Perhaps some visual variance in the rooms, or some advancement through it being implied!

More stuff

A key/door feature would be awesome

Maybe have the life of the character to be a bit more durable

*Figure 146 - Survey Suggestions of Game World.*

### Minimap

The minimap feature received very little feedback during testing. However, most players seemed to understand what the minimap was showing without any explanation. When asked during gameplay, users were able to correctly identify what the icons and symbols represented on the map, showing that the feature was intuitive and had reached the standard that it needed to. A helpful suggestion from a developer at Larian involved expanding the minimap into a full-screen map, similar to a pause menu, allowing players to see more of the explored area and understand their current position and needed direction. This idea has strong potential and could be developed further by adding a map legend to reduce possible confusion for players.

### Health Potions

Very minimal feedback was received about the health potions. Every user understood what they were and how they worked right away. Some early feedback suggested increasing the amount of health restored, which was adjusted before the second round of testing and received positive results. No users mentioned the spawn rate or the health value of the potions after the change. Overall, this can be considered a successful implementation.

### 6.3.4 Menu Interaction

A lot was learned about the menu system during the major user testing phase. The main menu worked well overall, with players understanding the button functions easily. However, the navigation was not as intuitive. Many users had trouble noticing which button they were currently hovering over until they moved through more of the menu options. While they eventually figured it out, the process took longer than expected. This issue appeared consistently across different age groups and gaming experience levels, showing a clear design flaw that needs improvement. Similar issues were also found in the main options menu and the pause options menu.

Interestingly, this problem did not occur in the Pause Menu. The strong hover effect used there likely helped the player easily see what option they were selecting, making navigation smoother.

## 6.4 Conclusion

### 6.4.1 Player Controls

User testing for movement and combat mechanics revealed clear strengths and weaknesses in the current system. The learning curve for player movement and combat mechanics was found to be quite low for both keyboard and gamepad users.

However, some confusion around the player's attack direction highlighted the need for some improvements. Adding more idle animations would help show what the player's last movement direction, making it clearer where their next attack will land. Refining the attack hitbox could also help players better understand the reach and width of their attacks.

A dash or dodge mechanic was another suggestion from testers. This feature like this could add more variety and excitement to the gameplay and would be a strong addition in future updates.

### 6.4.2 Enemy Interaction

User testing showed that some of the enemy interaction systems need improvement. The pathfinding system should be updated so enemies can detect and move around each other. Giving enemies different movement speeds would add variety and make each room feel more challenging as the game progresses. Adding more enemy types with different behaviours was a common suggestion from testers. There are many ways the enemy systems could be improved to make the gameplay even more interesting and dynamic.

### 6.4.3 Game World

User testing also highlighted how players interacted with the game world. While the game's mechanics were easy to understand, the visual similarity between rooms caused some confusion about where to go next. The minimap helped reduce confusion, but didn't fully solve the issue. A common suggestion was to make a full-sized map screen and to include more content in each room, such as obstacles and landmarks, to make navigation easier and make the game world more engaging.

### 6.4.4 Menu Interaction

Based on user testing with the main menu and the pause menu systems, it is evident that the hover effect plays a critical role in guiding player understanding and navigation. The tests consistently showed that players struggled with identifying which button or slider they were interacting with when the hover effect was subtle or insufficiently noticeable. This was particularly evident in the main menu and options menu, where players often scrolled past buttons or sliders before realizing their selection. However, in the pause menu, where the hover effect was more exaggerated, players demonstrated quicker and more accurate navigation, confirming that a stronger visual indicator significantly improves usability.

## 7 Conclusion

### 7.1 Introduction

This chapter provides a comprehensive summary of the project's development journey, offering a reflection on the progress made across each phase of the project, from initial research to implementation and testing. It revisits the original objectives, evaluates how effectively they were met, and outlines key achievements and challenges encountered throughout the process. In addition to reflecting on the overall design, this chapter also includes a summary of the technologies used and how they contributed to the project's goals. The final section presents personal reflections and insights gained, offering a clear perspective on the project's success and opportunities for future improvement.

### 7.2 Technologies

#### 7.2.1 Figma

Figma was used as the primary design application for most of this project where the original wireframes, iterations of menus, and user interfaces were designed. Figma also held references to the design research that had been done to better inform the wireframes that were being made for the project. Overall, Figma was a large part of the project with its design capabilities.

#### 7.2.2 Photoshop

Photoshop CS6 was used very little over the course of this project, only being called into action when designing assets for the user interface, specifically when designing the player's health bar, the updated pause menu design, and the mini map.

Photoshop was also used when designing stickers for the project which was requested by the college as part of the student ambassador opportunity at Dublin Comic-Con 2025.

Overall, Photoshop did play an integral role in the development of the user interface, without it the game would have been lacking a lot of smaller details that added a lot to visual aspect and the enjoyment of the game.

#### 7.2.3 Unity

The choice to use the Unity engine was one of the most important decisions for this project. This decision faced a lot of careful consideration and research (research in Chapter 2). Ultimately, the decision to work with the Unity engine came down to the simple fact that it had more learning resources and assets available to it which would directly impact the scope of the project. This decision was absolutely the correct one, staying inside the Unity engine to work on new game development techniques provided a much more straightforward and stress-free environment thanks to having previous experience with the engine. The help from the community was also a huge benefit to this project, having access to resources and people that can teach you different techniques was a large reason the project went as smoothly as it did.

Overall, Unity provided everything that was needed to be able to create this project, whether it was an intuitive input system or the ability to import third party or homemade assets through the package management system the engine has.

### 7.2.3 Visual Studio Code

Visual Studio Code was used as the IDE (Integrated Development Environment) for this project, being used to access, modify, and create game scripts. Although the version of Visual Studio Code was not the same version that it used regularly. Due to the fact that the IDE is booting through a different application, any plugins or saved user preferences do not carry over unless you manually log back into the IDE every time you wish to write a piece of code. This did make the experience a bit more challenging with the lack of code snippets and auto correction tools.

Overall, Visual Studio Code provided as a solid IDE to work with thanks to previous experience, it's integration with the Unity engine felt almost seamless at certain stages of the project and pushed the principles of double-checking work and syntax spelling through the lack of any help from plugins, which has improved my skills as a programmer.

### 7.2.4 Mirro

Miro was used as the project management and documentation application for this project, using a template provided by the college to be able to break down the work in sprint sections. Using Miro, each stage and goal of the project was properly planned and documented, this helped when needing to look back on previous version of the project when writing out the project report. Whenever there was a successful change made to the project, screenshots were taken to document these changes and then placed in the current section of the template that the project was in. When goals were completed, they were moved from the In Progress section to the Done section in the Kanban table that was used. Being able to look back on the Kanban and Sprint system to see previous goals and how they were implemented made huge impacts on the writing process of the report. Miro was a great tool to use for this project and shall be used again in future projects for the same level of project documentation and management.

### 7.2.5 Unity Version Control

The Unity Version Control system was something that I did not think I would be using when starting this project due to having no previous experience with it, and it was good that there wasn't too much trust placed in this system, since the engine often lost connection with the version control API.

When the system was functioning properly, it worked nearly exactly the same way as GitHub in the way that you could commit your code changes, push them up to a cloud based service and then pull or rollback the code if needed, all while being able to comment on your commits to see what each commit was for.

Although when the system was not working due to connectivity issues from the engine, there was no access to any of these options, effectively making the system useless for long periods of time. Thankfully the Unity Version Control system was not the only version control system that was implemented during this project so the risk associated with this system was minimized.

Overall, when the system was working, it was great, but the constant issues in connection with the API made it abundantly clear that this system was not ready to be deployed in its current state.

### 7.2.6 OneDrive

OneDrive was used as the primary version control system for the duration of this project due to previous experiences of losing assets when using GitHub to store Unity files. OneDrive was

used the same way as any version control system where an updated version of the game would be uploaded to the OneDrive file with a comment within the title of the file to note which version of the project it was and what was done most recently.

Although due to the size of the files that and the time it took for these files to be uploaded, the OneDrive pushes were done less frequently, only really being pushed when a major implementation was made to the game, such as the first version of the Procedural Generation system being created or when the Health item was brought into the game.

Overall, OneDrive's contribution to the project was a great help and a very good stress reliever. Knowing that there was a safeguard put in place in case of a major problem with the Unity Version Control system was a very large comfort, especially towards the end of the project's development when most of the work consisted of non-functional design-based updates.

## 7.3 Project Phases

### 7.3.1 Research

The research phase proved to be one of the most critical components of this project, particularly in how the applied research was conducted. Without the amount of research that was gathered, the project would have seen immense increases in bugs and technical problems when being developed.

Research into back-end system design was significantly more extensive than the front-end research due to a lack of knowledge and experience working with these kinds of programming ideas and techniques, but thanks to this extensive research section, better decisions were made on what game engine would be chosen for development of this project, the pathfinding algorithms that would be implemented, as well as the implementation process of procedural generation into a 2D game world.

Without this research phase, there would not have been such a well maintained codebase or stress free development environment, so this phase would be considered a success.

Overall, the research phase can be considered a major success, as it directly influenced key development decisions and laid a strong foundation for both design and implementation. The depth and relevance of the applied research ensured a smoother workflow, fewer technical issues and a more focused development process, ultimately contributing to the stability and coherence of the final product.

### 7.3.2 Requirements

The front-end and back-end design research conducted in Chapter 2 played a vital role in the development of the project's functional and non-functional requirements. By analysing design patterns, gameplay structures, and interface design philosophies from comparable games, this research provided a clear direction for defining the technical and experimental goals for the project. It laid the groundwork for both the design philosophy and the software architecture, ensuring that development could proceed with clarity and purpose. As a result, the requirements phase was well-informed and efficiently executed, directly benefiting from the depth of research established earlier in the process.

Overall, the requirements phase successfully translated research findings into actionable development goals, ensuring alignment between the intended user experience and technical



execution. It provided a clear roadmap for implementation, helped avoid over scoping the project, and supported a modular, maintainable development structure. This early clarity proved essential in maintaining focus throughout the project's development and contributed significantly to the project's overall stability.

### 7.3.3 Design

The design phase was largely influenced by the research and requirements work done in Chapter 2 and Chapter 3. With a strong foundation already established, both back-end architecture and front-end wireframes were easier to translate into practical and achievable design solutions. The early analysis of system structure and user interface principles allowed for a more focused and informed approach when outlining the project's functionality and user experience.

Overall, the design phase provided a clear visual and structural roadmap for development, bridging the gap between planning and implementation. The modular design approach and organized file structures helped streamline the development and simplify future adjustments. By aligning the design decisions with previously defined requirements, the project maintained consistency and coherence across both the technical and visual elements.

### 7.3.4 Implementation

The implementation phase was broken down into two week sprint segments, creating small and manageable goals to fulfil during the weeks. This system was followed as it seemed to adhere to the same style of development that was already being followed when planning out this project, to create small and effective implementations, keeping systems detached from each other as much as possible to create the best environment for debugging and code maintenance. The sprint goals were not radically adhered to, if the goals were completed ahead of schedule then I just kept working into the next development phase, not letting any time go to waste.

Thanks to a strong research phase, specifically in applied research, the beginning of the implementation phase started very strong as well, being able to create a test area full of foundational systems such as player combat, health, movement and collisions with very little issue. Even being able to create the first test for the procedural generation system.

This momentum continued to be carried through the rest of the implementation phases, taking these foundational systems and reworking them as the game needed, sometimes completely rewriting them but never changing the idea behind the system or how it should function. Thankfully, there were very few bugs found through the duration of the project, the ones that were found were traced and fixed before they became larger problems later in the project.

Overall, with the results of the user testing chapter in mind, this phase of development seemed to be a success, nearly every goal for the game was completed while consistently keeping a largely maintainable code base with very little to no bugs being found during the user testing phases.

### 7.3.5 User Testing

The user testing (as explained in Chapter 1) was broken up into three phases. The manual testing phase, the focus group testing phase and the major testing phase. Each phase was extremely beneficial for the development of the project. The manual testing phase went on for

the entire duration of the project, when adding a new feature to the game or modifying an already existing system, manual testing on the change or the addition would occur by myself.

The goal of manual testing would be to find bugs or exploits inside the game before letting other testers find them, so that other testing phases would provide better user data than simply listing issues that could have been found through manual testing. This goal was achieved with great success, being able to gather better user data from the focus group sessions that revolved around what the game can improve on rather than what the game needs to fix before it is playable.

The focus group sessions went very successfully thanks to the manual testing which was conducted before, the game received great user feedback about how the game felt, what could be changed, and what new features the game would need to become a better experience for the users. The focus group really shined a light on the direction the game needed to go.

The final phase of testing was the major testing phase. A handful of students along with myself were given the opportunity to attend Dublin Comic-Con in March of 2025 as ambassadors for the Game Design course which would be launching its first year in the coming September. During this event, the major testing phase would be conducted. The goal was to get as many people to play the game and to give feedback on the current stage of the game's development and this goal was a success. There weren't many people who wanted to do the surveys made for the game but there were a lot of people who enjoyed giving their feedback through conversation which was just as useful.

Contact with game studios such as Larian and Black Shamrock along with game design organizations such as Desk Rage and IMIRT was made during this phase of testing and they gave some incredible feedback on not only the game but also in terms of furthering my career in game design.

Overall, each phase of user testing was a success, each phase complimenting the previous and getting great results through all of them.

## 7.6 Reflection

### 7.6.1 Project Management

Project management was handled effectively throughout the duration of the project. The use of a Miro board ensured that all documentation, goals, and materials were organized in a central, accessible location. Breaking the documentation into individual goals and agile sections made it easier to reference changes and progress when compiling materials for the final report. While the sprint goals were more aligned with a Kanban methodology than traditional sprints, this approach contributed to a consistent and manageable workflow. These project management strategies played a significant role in the project's overall success and helped maintain a low-stress development environment.

Overall, the chosen project management approach helped keep everything organized and on track. Breaking the work into smaller goals made it easier to stay focused and adjust when needed. This method kept development moving smoothly and made it easier to record progress for reports and documentation.

### 7.6.2 Views on The Project

After the initial brainstorming phase, there was a strong sense of confidence in the project's feasibility and direction. While the visual style of the project did evolve during development, the core concept of the game remained consistent throughout development. The current state of the project at this time reflects a solid execution of that idea. Positive feedback from both industry professionals and the general public has been especially rewarding, reinforcing the project's creative and technical efforts.

Overall, the project has been a valuable learning experience in both design and development. It demonstrated how a clear vision, guided by research and iterative development, can result in a well-received final product. The experience has strengthened long-term goals within the game development field and built confidence in pursuing further opportunities in the industry.

### 7.6.3 Working with a Supervisor

Working with Timm on this project was an absolute treat. Throughout the development process, he was always supportive and encouraged the vision that was shared to him from the start. He gave helpful feedback and asked fair questions that showed he was genuinely interested in the project. His advice was clear and easy to understand, which made it easier to stay confident and focused.

Timm was easy to talk to and always seemed to understand where the project was going. Meetings felt productive, and his support and guidance made a big difference in the success of the project and created a really positive working environment.

### 7.6.4 Further Competencies & Skills

Throughout this project, skills in C# and the Unity engine improved significantly thanks to hands-on problem solving and research into back-end systems. Moving to a modular scripting approach also made it easier to explore and test different features without affecting other parts of the project. This structure allowed more flexibility to experiment, learn from mistakes, and find working solutions more confidently.

This was also the first time a procedural generation system was developed, which involved both theory and applied research to be conducted. That process helped deepen the understanding of Unity's capabilities, especially in how to generate dynamic game worlds. Another new area was pathfinding. Learning how to utilise a system that detects and follows the player through dynamic levels was very difficult.

Overall, the experience built a stronger foundation in programming, game systems, and Unity workflows. These skills will be valuable in future work, for both solo and collaborative works.

### 7.6.5 How the Project Could be Developed Further

This project has a lot of potential for future improvements, with most of them focusing on adding more content to the project. One main issue during testing was the lack of idle animations in each cardinal direction being implemented, which made it harder for players to tell which way their aim was facing, sometimes leading players to miss an attack.

Another common suggestion was to introduce more enemy types to keep combat interesting and varied. Other possible additions include increasing room variety, creating a full map view for better navigation, and adding systems such as keys to open doors, an inventory system for

managing items, and a boss fight to give the project more of a goal. These ideas all support the same development direction, expanding the project's content to make the experience deeper and more rewarding.

Overall, the project has a solid foundation to build on, and adding these features would help bring the game closer to a more complete and polished version.

## 7.7 Conclusion

In conclusion, this project brought together all key stages of game development (Research, Planning, Design, and Implementation) into one well-rounded and rewarding process. Early research played a major role in guiding decisions about the game's structure, helping to avoid common problems and supporting important choices around tools, systems, and programming techniques. This included deep research into procedural generation and pathfinding algorithms, which were major technical milestones in the project. These systems helped shape a dynamic and replayable game world and taught valuable lessons about how complex features work inside the Unity engine.

The requirements and design phases, which were heavily influenced by the research phase, gave the project a solid foundation to build on, making it easier to move into development with clear goals and a structured plan. During implementation features such as animation, user interface, audio, and gameplay mechanics were built and tested in smaller pieces, allowing for better control, easier debugging, and a smooth development experience. Project management tools like Miro helped keep the work on track, while feedback from testers provided useful insights for future improvements.

Overall, the project achieved its goals and demonstrated what's possible with a strong creative vision and a steady, organized workflow. While there is still room to grow, especially with more content and expanded features, the project as it stands is functional, stable, and a solid foundation for future development.

## 8 References

- Adobe Systems Incorporated. (2012). *Adobe Photoshop CS6* [Computer software]. <https://www.adobe.com/products/photoshop.html>
- Belwariar, R. (2018, September 7). A search algorithm - GeeksforGeeks. GeeksforGeeks. <https://www.geeksforgeeks.org/a-search-algorithm>
- Chris's Tutorials. (2021, May 3). *How to import a 2D character sprite sheet and use in a GameObject in Unity (2021)* [Video]. YouTube. <https://www.youtube.com/watch?v=FXXc0hTWIMs>
- Code Monkey. (2018, September 16). *How to make a simple minimap (Unity tutorial for beginners)* [Video]. YouTube. <https://www.youtube.com/watch?v=kWhOMJMihC0>
- Epic Games. (n.d.). *Unreal Engine* [Game engine]. <https://www.unrealengine.com/>
- Figma, Inc. (2024). *Figma* (Version X.X) [Computer software]. <https://figma.com/>
- Free Game Assets. (n.d.). *Free base 4 direction male character pixel art* [Game asset]. itch.io. <https://free-game-assets.itch.io/free-base-4-direction-male-character-pixel-art>
- Free Game Assets. (n.d.). *Free slime mobs - pixel art top-down sprite pack* [Game asset]. itch.io. <https://free-game-assets.itch.io/free-slime-mobs-pixel-art-top-down-sprite-pack>
- Game Code Library. (2023, August 10). *Melee & ranged top down combat - Unity 2D* [Video]. YouTube. <https://www.youtube.com/watch?v=-4bsGg7dVFo>
- Game Code Library. (2024, June 25). *PERFECT tilemap sorting layers - Top Down Unity 2D #3* [Video]. YouTube. <https://www.youtube.com/watch?v=Uld0mwanBZg>
- Game Code Library. (2024, August 9). *Player tracking and camera bounds - Top Down Unity 2D #4* [Video]. YouTube. <https://www.youtube.com/watch?v=kV9rVinFyAk>
- Game Code Library. (2024, August 22). *Map transitions by waypoints - Top Down Unity 2D #5* [Video]. YouTube. <https://www.youtube.com/watch?v=9r9YbHsjSKs>
- Godot Engine. (2023). *Godot 4* [Game engine]. <https://godotengine.org/>
- h8man. (2022, November 26). *NavMeshPlus*. GitHub. <https://github.com/h8man/NavMeshPlus>
- Iqbal, M. A., Panwar, H., & Singh, S. P. (2022). Design and implementation of pathfinding algorithms in Unity 3D. *International Journal for Research in Applied Science and Engineering Technology*, 10(4), 71–79. <https://doi.org/10.22214/ijraset.2022.41136>
- Maurya, A., Yadav, A., & Baiswar, A. (2022). Pathfinding visualizer. *I-Manager's Journal on Software Engineering*, 16(4), 24–24. <https://doi.org/10.26634/jse.16.4.18801>
- McMillen, E., & Himsl, F. (2011). *The Binding of Isaac* [Video game]. Edmund McMillen. [https://store.steampowered.com/app/113200/The\\_Binding\\_of\\_Isaac/](https://store.steampowered.com/app/113200/The_Binding_of_Isaac/)
- Microsoft. (2024). *OneDrive* (Version X.X) [Computer software]. <https://onedrive.live.com/>

- Microsoft. (2024). *Visual Studio Code* (Version X.X) [Computer software]. <https://code.visualstudio.com/>
- Nuke Nine. (2018). *Vagante* [Video game]. Nuke Nine. <https://store.steampowered.com/app/323220/Vagante/>
- PitIT. (2022, March 30). *Unity tutorial: Knockback anyone in ANY game* [Video]. YouTube. <https://www.youtube.com/watch?v=ZyCixhKdslo>
- Quaternius. (n.d.). *Pixel art top down basic* [Asset pack]. Unity Asset Store. <https://assetstore.unity.com/packages/2d/environments/pixel-art-top-down-basic-187605>
- RealtimeBoard, Inc. (2024). *Miro* (Version X.X) [Computer software]. <https://miro.com/>
- Rehope Games. (2023, February 25). *How to add music and sound effects to a game in Unity | Unity 2D platformer tutorial #16* [Video]. YouTube. <https://www.youtube.com/watch?v=N8whM1GjH4w>
- Rehope Games. (2023, March 2). *Unity audio mixer tutorial | Unity 2D platformer tutorial #17* [Video]. YouTube. <https://www.youtube.com/watch?v=IxHPzrEq1Tc>
- Risi, S., & Togelius, J. (2020). Increasing generality in machine learning through procedural content generation. *Nature Machine Intelligence*, 2(8), 428–436. <https://doi.org/10.1038/s42256-020-0208-z>
- Rootbin. (2023, July 21). *2024 AI pathfinding: Unity 2D pathfinding with NavMesh tutorial in 5 minutes* [Video]. YouTube. <https://www.youtube.com/watch?v=HRX0pUSucW4>
- Rootbin. (2023, October 29). *Unity tutorial: Roguelike room / dungeon generation (like The Binding of Isaac)* [Video]. YouTube. <https://www.youtube.com/watch?v=eK2SlZxNjiU>
- Sasquatch B Studios. (2024, February 15). *Top down movement - Unity tutorial* [Video]. YouTube. <https://www.youtube.com/watch?v=RN3yuCvazL4>
- Shen, Z. (2022). Procedural generation in games: Focusing on dungeons. *SHS Web of Conferences*, 144, 02005. <https://doi.org/10.1051/shsconf/202214402005>
- Supergiant Games. (2020). *Hades* [Video game]. Supergiant Games. <https://www.supergiantgames.com/games/hades/>
- Team Cherry. (2017). *Hollow Knight* [Video game]. Team Cherry. <https://www.hollowknight.com/>
- Tokegameart. (n.d.). *Pixel potions with animation* [Asset pack]. Unity Asset Store. <https://assetstore.unity.com/packages/2d/environments/pixel-potions-with-animation-118801>
- Unity Technologies. (2024). *Unity Asset Store* [Digital marketplace]. <https://assetstore.unity.com/>
- Unity Technologies. (2024). *Unity Version Control* (Version X.X) [Computer software]. <https://unity.com/products/version-control>

Unity Technologies. (2025). *Unity* (Version 6000.0.32f1) [Computer software].

<https://unity.com/>

User Testing Form. (2025). *User testing form*. Google Docs.

[https://docs.google.com/forms/d/e/1FAIpQLSc7JTgWyRHWGSS2lXjOtmr795Z6RcUFieAnb\\_AJ19uUnOgZ3Q/viewform?usp=dialog](https://docs.google.com/forms/d/e/1FAIpQLSc7JTgWyRHWGSS2lXjOtmr795Z6RcUFieAnb_AJ19uUnOgZ3Q/viewform?usp=dialog)

Yannakakis, G. N., & Togelius, J. (2011). Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2(3), 147–161.

<https://doi.org/10.1109/T-AFFC.2011.6>