

# Final Year Project Report

## Full Unit – Final Report

---

# Building a Game

Prithvi Sathyamoorthy Veeran

---

A report submitted in part fulfilment of the degree of

**BSc (Hons) in Computer Science**

**Supervisor:** Julien Lange



Department of Computer Science  
Royal Holloway, University of London

April 11, 2025

## Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 14845

Student Name: Prithvi Sathyamoorthy Veeran

Date of Submission: 11/04/2025

Signature: Prithvi Sathyamoorthy Veeran

# Table of Contents

Abstract .....	5
Chapter 1: Introduction.....	6
1.1 Background.....	6
1.2 Aims & Objectives .....	6
1.3 Project Motivation.....	7
1.4 Technological Choices .....	8
1.5 Timeline (Milestones) .....	9
Chapter 2: Literature & Background Theory .....	13
2.1 Background.....	13
2.2 Game Design Principles in Roguelike Dungeon Crawlers .....	14
2.3 What Makes a Successful Roguelike/Dungeon Crawler? .....	15
2.4 Historical and Industry Context .....	16
Chapter 3: Current Implementation .....	18
3.1 Procedural Dungeon Generation .....	20
3.2 The Player .....	24
3.3 The Enemy .....	27
3.4 Environmental Interaction.....	29
3.5 User Interface .....	30
3.6 Visual & Audio Design.....	34
3.7 Settings Management .....	35
Chapter 4: Software Engineering .....	38
4.1 Software Methodologies .....	38
4.2 Testing.....	39
4.3 Code Quality .....	41
4.4 Version Control .....	41
4.5 Requirements Analysis.....	42
4.6 Godots Architecture and Technical Design .....	43
Chapter 5: Professional Issues.....	46

Chapter 6: Reflections.....	48
6.1 Reflection on the Project .....	48
6.2 Personal Self-Assessment .....	48
6.3 Analysis of Code and Challenges Encountered .....	48
6.4 Milestones Achieved .....	49
6.5 Future Plans.....	50
Chapter 7: Bibliography.....	51
Chapter 8: Appendix .....	54
8.1 User Manual.....	54
8.2 Gantt Chart – Project Timeline.....	55
8.3 User Play-Testing Form.....	55
8.4 Project Diary .....	55
8.5 Link to Demo Video.....	64

# Abstract

This project involved the development of a 3D top-down dungeon crawler game built using the Godot 4 engine. The game puts the player in a procedurally generated dungeon environment where they must traverse through a vast dungeon with many rooms, fight enemies, and find items such as potions and chests to use during the gameplay. Core features within this game include real-time melee-based combat, AI-driven enemies, an animated UI, score tracking which saves the highest score, and a modular scene system for spawning rooms, interactable items, and enemies.

Procedural generation was implemented through modular grid-based logic, where the rooms are predefined scenes, allowing them to be generated in random areas and connected through defined doorway points. Each room spawns enemies that respond dynamically to the player's distance, adopting behaviour states such as idle, chase, and attack. Some enemies, such as ranged mages, use spellcasting through coroutine-based logic, further enriching combat variety. The interaction mechanics allow players to collect healing potions around the dungeon or open chests which grant the player a sizeable score reward, enhancing player engagement and reward systems.

The game was developed using GDScript and structured around Godot's node-based architecture to improve scalability and maintainability. Alongside that an agile development methodology was used to manage iterative implementation and testing. Audio-visual feedback, modular scripting, and consistent coding standards were applied throughout development to ensure a polished and responsive gameplay experience. The game also incorporates a user interface, which has a health and score display, pause and option menus, and persistent high score tracking.

# Chapter 1: Introduction

In recent years, the game development industry has seen a rise in interest toward procedurally generated environments and dynamic gameplay systems. This project explores the development of a game that focuses its core principles on roguelikes and procedural generation. By integrating key game design principles with modern development tools and methodologies, this project not only displays technical implementation but also reflects on the creative and iterative process of building an interactive system from the ground up. This report presents the full development lifecycle, from initial research and design to testing, refinement, and final evaluation.

## 1.1 Background

The origins of the dungeon crawler genre come from tabletop role-playing games such as Dungeons & Dragons, where players are tasked with traversing complex environments filled with enemies, treasures, and traps. The classification of a game as a roguelike is dependent upon various significant and minor factors [22]. Video games like Rogue (1980) and Diablo (1996) [4] significantly contributed to the genre's popularity by incorporating hack-and-slash combat mechanics and procedural level generation, which subsequently became hallmarks of the genre. Diablo 2 (2000) [5] further enhanced these elements, creating a paradigm for modern dungeon crawlers through its diverse character classes, skill trees, and procedurally generated environments. Games like Hades (2020) and Torchlight (2009) have reinvented the genre in recent years with their dynamic combat systems, stylised graphics, and intricate narratives. These advancements have demonstrated the genre's versatility and everlasting appeal, prompting creators to continually generate creative ideas.

## 1.2 Aims & Objectives

### 1.2.1 Aims

This project aims to create a roguelike, top-down dungeon crawler game that integrates modern gaming technology with the nostalgia of traditional dungeon crawlers. The game contains procedurally generated dungeons, tough real-time combat, and an immersive visual and audio experience. Engineered for replayability, it seeks to impart the exhilaration of fighting, the allure of exploration, and the gratification of discovering treasures and vanquishing foes. The project aims to maintain superior technical quality while ensuring the game is engaging for a broad audience through the use of advanced game development techniques and the integration of accessibility features.

### 1.2.2 Objectives

- **Procedural Level Generation:** Leverage robust procedural generating methods to provide diverse and unpredictable dungeon configurations. This entails creating diverse landscapes featuring distinct themes, configurations, and levels of difficulty to sustain player engagement. Including chests and collectibles that will appear in the dungeon for added immersion and player engagement. This will be achieved through:
  - Developing algorithms, such as cellular automata, which utilise randomisation and weighted probability to produce unique dungeon layouts or using modular grid-based generation, which is where areas are dynamically instantiated.
- **Engaging Combat Mechanics:** Develop a combat systems that enables the players to strategically think about approaching certain rooms in the dungeon as they are a melee

only player. Develop an adaptive artificial intelligence that responds to player actions and delivers a tough gaming experience. Steps to implement can include:

- Establishing a modular combat system featuring interchangeable attack types,
- Iteratively testing and balancing combat mechanics to guarantee gameplay that is fair and enjoyable.
- **Loot:** Create and implement an intuitive item system enabling players to obtain and use items discovered within the dungeon.
  - Implementing item categorisation into categories such as consumables, equipment, and quest items.
  - Ensuring seamless integration of the loot system into the gameplay loop.
- **User Interface Design:** Develop an aesthetically pleasing and user-centric interface that promotes intuitive navigation and improves the overall gameplay experience. This objective encompasses health bars, inventory management systems, and a potential mini map.
- **Visual Design:** Develop an environment that's appealing visually and consistent, which should complement gameplay mechanics and enhance immersion.
  - Use/Create cohesive assets and sprites, with animations and visual effects that reflect character actions, enemy behaviours, and certain environmental interactions.
  - Ensuring visual clarity and readability of gameplay elements, facilitating play interaction and engagement.
- **Audio Design:** Create an immersive audial experience that enhances the gameplay for the player by providing background music and atmospheric ambience, along with sound effects for interactions, combat and environment, and clear auditory feedback for user actions and game events.
  - Using thematic tracks that align with the mood and genre of the project.
  - Implement sound effects for player actions, enemies, environment and ambience.

## 1.3 Project Motivation

The dungeon crawler genre has consistently inspired and fascinated me, driven by a lifelong enthusiasm and passion for gaming. This project represents a personal endeavour to integrate my passion for gaming with the technological expertise I have developed through my academic and professional experiences. Since childhood, I have been captivated by the complex mechanics, strategic dilemmas, and immersive environments presented by games. This project provides an opportunity to develop something that is uniquely my own, merging creativity and technology into a game of which I can take pride.

The choice to create a dungeon crawler arises from its significant potential for creativity and player involvement. Titles such as Diablo 2 and Hades exemplify the integration of exploration, combat, and narrative into a captivating experience. This project draws from these ideas, seeking to expand the limits through the integration of procedural generation, dynamic combat mechanisms, and captivating level designs featuring puzzles and traps. This project leverages the genre's strengths

while addressing shortages in current games. This project uses procedural generation, distinguishing it from many games that utilise static levels, thus guaranteeing that each playthrough is unique and enhancing replayability.

This project, aside from fulfilling a university requirement, has provided me an entry point into my lifelong passion for game development. Even though I have always found enjoyment in games, developing one from scratch provided me with an entirely new perspective and has enhanced my love for the craft. This project has pushed me both technically and creatively, giving me the chance to explore the foundations of game design, user experience, and iterative development. It has supported my long-term goal of advancing my interests in game development, either professionally or as a personal goal. In many ways, this marks the beginning of that journey – my first real step into the world of creating interactive experiences and a solid foundation that I hope to build on in future projects and potentially a career.

This project additionally serves as an opportunity to demonstrate my progression as a programmer and my capacity to manage a complex, multifaceted project. Utilising modern technologies such as the Godot 4 engine, procedural algorithms, and agile development methodologies, I intend to make a game that embodies both my enthusiasm and my technical expertise, as well as my creativity. This project aims to actualise a personal vision and share it among fellow game enthusiasts.

## 1.4 Technological Choices

While developing a game, the technologies used are an important factor to consider as a developer, as this is what allows the game to have a strong foundation. Each choice is backed by the ability to achieve the project's objectives. The combination of these technologies ensures that the project can meet its ambitious goals of delivering a game that is dynamic and polished.

### 1.4.1 Game Engine: Godot 4

Godot 4 was selected as the primary development engine because of its versatility, lightweight structure, and comprehensive support for both 2D and 3D game creation. The integrated script editor and robust features facilitate rapid iteration and testing, rendering it suitable for developing complex systems like procedural level creation and dynamic combat mechanics. It also allows for easy deployment on multiple platforms, allowing publication of the game to be a lot easier. Furthermore, Godot's open-source framework allows for customisation, while its strong developer community provides dependable support and resources. These elements render Godot 4 an excellent and cost-effective option for this project.

### 1.4.2 Programming Language: GDScript

GDScript, the native scripting language of Godot, is essential to this project. Its lightweight syntax has been precisely designed for game development, guaranteeing seamless integration with Godot's extensive API. This close integration helps the development of gaming mechanisms, including character movement, combat interactions, and AI behaviours. The simplicity of GDScript enables swift prototyping, while its performance optimisation guarantees smooth gameplay experiences. The advanced capabilities of GDScript, including class extension and reusable component creation, provide it as an essential tool for developing modular and scalable games.

### 1.4.3 Project Management Tools: Trello

Trello functions as a core project management tool, allowing for easy task monitoring, milestone scheduling, and workflow optimisation. It endorses the Agile technique implemented for this project, guaranteeing adaptability and ongoing assessment of progress. By categorising tasks and tracking their completion, Trello helps maintain clear focus throughout the project lifecycle.



#### 1.4.4 Version Control: GitLab

GitLab was chosen as the version control system to support iterative development and safe experimentation. Its robust branching and merging abilities allowed efficient parallel development of features. The review of code changes can be done due to the merge functionality, enhancing the quality of it and reducing the number of errors in the codebase. GitLab's integration of Continuous Integration (CI) also supported automated testing processes, which ensured that each integration maintained the project stability and readability.

#### 1.4.5 Technologies That Were Considered

Several technologies were carefully explored and analysed early in the project to determine the best tools for attaining the project's objectives. While the Godot engine and GDScript were eventually chosen, other engines and programming languages were examined but rejected for various reasons.

Unity and Unreal Engine were top contenders due to their robust feature sets and active development communities. Unity's versatility, C# scripting and robust asset store offered tremendous flexibility, which makes it a market favourite, although Unreal Engine's graphics powers and Blueprint visual scripting system are unrivalled. However, both engines have a higher learning curve than Godot, especially for a single developer working on a time-sensitive project. In contrast, Godot's accessible interface, streamlined scripting via GDScript, and a lightweight framework with an emphasis on 2D/3D development provide a much more suitable solution for the scope and technical needs for this project.

Regarding programming languages, C# (with Unity) and C++ (with Unreal Engine) were both considered. Both these languages offer extensive capabilities and performance advantages; the simplicity, readability, and native integration of GDScript with Godot's engine outweighed the alternatives, facilitating much quicker integration and debugging – which are key factors in the project's agile methodologies.

Lastly, for visual asset creation, alternatives such as Adobe Photoshop or Aseprite were considered. Despite the freedom they both provide to create whatever assets for the project, due to time constraints, it was better suited to use pre-made assets such as the UI and ones for the actual game world.

### 1.5 Timeline (Milestones)

The timeline for the development of this game spans over the academic year, split into two terms. The project will be split into several phases to ensure that both the design and the implementation of the game are progressing well. Each phase will include specific milestones, ensuring that development stays on track and allows for flexibility for any unforeseen changes. As seen below:

#### 1. Pre-Production Phase (week 1-3)

- **Objective:** Establish a clear foundation for the project, such as conceptualising the game mechanics, setting up the development environment, and preparing the initial assets. The conduction of research will still be going on, as well as research into the game engine.
  - **Week 1-2:**
    - Initial project planning and setup.
    - Study the Godot engine and mechanics along with papers relevant to the mechanics of the game.
    - Create the initial project design document.

- Meet with the supervisor and finalise ideas.
- Start the draft of the project plan.
- **Week 3:**
  - Carry out learning on the Godot engine.
  - Complete tutorials of basic core functions.
  - Complete the project plan.

## 2. Core system development (week 4-10)

- **Objective:** Focus on developing the game's core gameplay mechanics, including the menu screens for the player to navigate, player controls, enemies (and their AI), dungeon map creation, and basic combat mechanics. The goal here is to have a playable prototype by the end of this phase. Alongside all of this, it is important to make sure that the report is being written up while the development of the game is in progress.
  - **Week 4-5:**
    - Create the menus desired for the game, along with fluid connectivity between them.
    - Implement player movement and a basic combat system.
    - Test to see if these mechanics are suitably implemented.
  - **Week 6-7:**
    - Set up the player HUD.
    - Begin development of enemies and their factories.
  - **Week 8-9:**
    - Set up collision with player, enemies, and the placeholder map.
    - Test simple player, enemy, and map interaction.
  - **Week 9-10:**
    - Develop the procedural generation of the dungeon.
    - Integrate player and enemy mechanics into procedural levels.
  - **Week 11-12:**
    - Add any other features if there is time, such as power-ups.
    - Refine some of the mechanics and refactor.
    - Consider all test cases for everything developed so far.

- Prepare for the interim review, the 5000-word report and the demo code.

### 3. Milestone: first playable prototype (end of term 1)

- By the end of term one, there will be a working prototype, featuring the basic functions of the game, such as exploration, combat, and rudimentary dungeon generation. In-house playtesting will begin at this stage, and the feedback from this will be collected to guide the development into the next term.

### 4. Advanced features and refinement (Week 11-17)

- **Objective:** In this phase I will refine core systems and add more complex features, such as advanced enemy AI, further intricate level design, and a more fleshed-out combat design. This phase will also contain debugging and playtesting.
  - **Week 11-13:**
    - Consider the feedback from the playtesting and adjust the timeline to fit in these comments.
    - Enhance the enemy AI (pathfinding, group behaviours).
    - Add more enemy types with potential unique abilities (bosses (optional)).
  - **Week 14-15:**
    - Implement the power-up and pickups system (inventory system).
    - Refine the procedural generation.
  - **Week 16-17:**
    - Test all the mechanics in the game so far.
    - Bug fixing and early optimisation is to be conducted.

### 5. Milestone: Alpha version (mid-term 2)

- At this stage, the game should be feature complete in terms of the core elements, with all the major mechanics functional. The focus of the project will now shift to finalisation, fixing bugs, polishing the game, and if there is extra time, any additional optional features.

### 6. Polish and final testing (weeks 17-23)

- **Objective:** Focus on the refinement of the game experience, including the visuals, user interface, and any other extra features. Extensive playtesting will be done to make sure the game is polished.
  - **Week 17-19:**

- Finalise all the assets required
- Polish procedural map generation.
- **Week 20-22:**
  - Conduct final playtesting.
  - Resolve any remaining bugs and make final optimisations.
- **Week 23:**
  - Prepare the final submission.
  - Prepare for the final report and demo.
  - Publish game.

#### **7. Milestone: Final submission (End of term 2)**

- The game should now be complete, along with the final report and the demo for it. Any remaining work will be concluded.

A detailed Gantt chart representing the projects development timeline is included In **appendix 8.2.**

## Chapter 2: Literature & Background Theory

### 2.1 Background

Roguelikes have long been highly regarded for their focus on replayability, challenging gameplay, and emergent narratives. At the core of their appeal is procedural generation, a technique that dynamically creates content such as levels, enemy configurations, and loot for each game algorithmically, rather than manually. This synergy between the roguelike genre and procedural generation fosters endless variability, ensuring that no two playthroughs are identical. Players are encouraged to adapt to unpredictable environments and situations, which adds depth and excitement to the gameplay experience. Procedural generation not only reduces the manual labour required for level design [30] but also aligns seamlessly with roguelike principles of exploration, adaptability, and tactical decision-making. By creating a unique and dynamic world, these games maintain player engagement while opening opportunities for creative design and technological innovation.

The variety of game genres emphasises the unique qualities meant to satisfy different player experiences. For example, although action games include fast-paced fighting and reflex-driven difficulties, role-playing games concentrate on narrative richness and character advancement. Particularly noteworthy are that roguelikes' focus on strategic gameplay, repeatability, and uncertainty. Procedural generation—which guarantees that every playthrough stays different and interesting—is a trademark of the roguelike genre. Though *Rogue* gave its name to the roguelike genre, it was not the first game to use procedural generation and permadeath [35]. Still, its impact confirmed these concepts as a mainstay of the genre and highlighted their continuing appeal in games.

Not limited to roguelikes, procedural generation has been effectively applied in many genres. Procedural approaches are shown by games such as *Minecraft* and *Diablo* [4]. Procedural generation in *Minecraft* controls not only the terrain but also the location of buildings and treasures by use of artificial intelligence [28]. These illustrations show how effectively procedural generation could improve replayability and envelop players in dynamic and large environments.

Modern gaming also depends on artificial intelligence (AI), which shapes the behaviour of non-player characters (NPCs) and improves the whole player experience. From simple pathfinding and opponent movement to complex decision-making systems modelled on human conduct, artificial intelligence finds expression in games. In dungeon crawlers and roguelikes, AI is essential for creating engaging challenges, as enemies must adapt dynamically to player actions while adhering to the game's design principles. Millington [26] highlights the importance of balancing intelligence with predictability, ensuring enemies are formidable yet not frustratingly overpowered. Techniques like finite state machines and A\* pathfinding algorithms are commonly employed to manage enemy behaviour, facilitating adaptive responses to player movement, combat, and exploration. In procedural environments, AI must also adjust to the unpredictability of the game world, ensuring cohesive and responsive gameplay. By integrating these AI principles, games create an immersive experience where players feel as though they are competing against intelligent adversaries rather than pre-programmed scripts.

Good and poor games vary mostly in their capacity to enthrall players and offer a coherent, pleasurable experience. Well-designed games captivate players with their simple mechanics, meaningful challenges, and balance of accessibility and complexity. Effective level design depends, as Adams [1] underlines, on well-defined goals, logical development, and a feeling of accomplishment. On the other hand, badly crafted games irritate players with imbalanced difficulty, unclear objectives, or repetitive surroundings that fail to motivate involvement.

A critical component of good game design is pacing—offering moments of intensity balanced with opportunities for exploration and reflection. Adams [1] further notes that well-crafted levels subtly guide players, keeping them orientated while encouraging exploration. Conversely, games that rely on repetitive tasks or lack variety risk losing the player's interest. Visual and audio feedback also play a vital role in reinforcing player actions and immersing them in the game world. Games with inconsistent design elements or insufficient polish can break immersion, ultimately alienating players. By adhering to these principles, games can deliver an engaging and cohesive experience that resonates with their audience.

With a unique mix of procedural generation, emergent gameplay, and strategic complexity, roguelikes have carved themselves a clear spot in the gaming industry apart from other genres. Their emphasis on unpredictability and repeatability never fails to excite players with experiences that seem novel and interesting with every run. While other genres depend on handmade levels or linear advancement, roguelikes flourish on the challenge and discovery that accompany dynamic, always-shifting environments. This flexibility not only fits the changing expectations of players but also guarantees the relevance of the genre in a market that celebrates creativity. As games like *Hades* and *The Binding of Isaac* demonstrate, roguelikes are not just a niche—they are a testament to the enduring appeal of games that challenge and reward in equal measure.

## 2.2 Game Design Principles in Roguelike Dungeon Crawlers

### 2.2.1 Core Gameplay Loop:

In game design, the gameplay loop refers to the iterative cycle of actions that a player engages in continuously [6]. Roguelike dungeon crawlers generally exhibit a tight core loop: the player navigates procedurally generated dungeon levels, engages in combat with enemies, collects loot or upgrades, and tries to survive as long as possible. Upon death (which is permanent to true roguelikes), the game resets, beginning the game loop afresh. This loop is inherently compelling because each run pushes the player to advance further or refine their strategy to tackle the level. In my project, the core loop was designed around short dungeon “runs” – players explore a new dungeon layout, fight enemies, collect loot and eventually either perish or achieve the win threshold, then repeat. This design embodies the “one more game” mentality, which is common in roguelikes, where a quick restart keeps the player coming back for another try [10]. By guaranteeing that each cycle of play is fun and rewarding on its own, the player stays engaged despite the repetition of the loop [14].

### 2.2.2 Player Feedback and Responsiveness:

Providing clear and immediate player feedback is essential for a satisfying gameplay experience [14]. In a fast-paced dungeon crawler, players must immediately see the outcomes of their actions; for example, an attack should deliver a distinct sound and visual effect, enemy health bars should reflect this change, and damage to the player should be obvious. Audiovisual signals assist players in comprehending the repercussions of their choices and make moment-to-moment combat feel “juicy” and responsive [14]. My game emphasises the feedback with the implementation of screen shakes and health bar depletion upon damage received and unique sound cues for different events. This not only improves the player’s immersion but also serves a gameplay purpose: prompt feedback enables the player to learn and modify their strategies.

### 2.2.3 Progression systems:

Well-designed progression instils a sense of advancement in players and sustains their motivation [14]. Roguelike dungeon crawlers often have two tiers of progression: within-run progression and meta-progression. During a run, the player’s character grows stronger by levelling up, acquiring better gear, or learning new skills and abilities – a quintessential RPG progression that provides a rewarding feeling. Traditional roguelikes, however, use permadeath, such as my own, resulting in the complete reset of progress upon death, so starting each new run from the beginning. To soften

this blow and encourage long-term engagement, many modern roguelike-inspired games (often referred to as roguelites) add a feature called meta-progression: persistent upgrades or unlockables that carry over between the runs. This way, even an unsuccessful attempt might contribute to something permanent, addressing the concern that repeated failures may feel futile [10]. The literature suggests that providing players with a sense of persistent progress greatly increases long-term retention in roguelikes [10]. Indeed, games like *The Binding of Isaac* famously have unlockable items and characters to keep the player invested across several hours. Another example would be *Slay the Spire*, where each run allows the player to unlock more cards to add to their deck, in turn allowing the player to build a more powerful deck when completing a run of the spire.

#### 2.2.4 Procedural Generation:

Procedural content generation is a cornerstone of roguelike design, creating fresh level layouts, enemy placement and item drops on each playthrough [21][15]. Randomisation guarantees that no two runs are identical, directly enhancing variety and replayability. Early roguelikes like *Rogue* (1980) pioneered this approach by using pseudorandom algorithms to generate dungeon maps and surprises for the player [21]. Each dungeon level is algorithmically constructed, populating rooms with different configurations of enemies and loot each time. This design principle prevents the game loop from becoming predictable or stale - players are required to react to new circumstances spontaneously, which is a key part of the fun. This project draws on established practises, but even random generation follows rules (sometimes called “procedural design” rather than pure randomness) - for example, making sure that each level has at least one viable path and appropriate resources for the player. The outcome is a balance between surprise and playability. According to Josh Bycer, the integration of randomisation with a structured progression of difficulty and diversity is pivotal for enhancing the replayability of roguelike games [15].

## 2.3 What Makes a Successful Roguelike/Dungeon Crawler?

Designers and researchers have identified certain qualities that an effective roguelike or dungeon crawler should strive for. These include a satisfying level of challenge, substantial content variation, strong replayability, and an immersive player experience.

- **Challenge and Fairness:** Roguelikes are characterised by their high difficulty and unforgiving nature – but part of their appeal is the thrill of surviving against the odds. However, successful design must guarantee that the challenge is perceived as fair and not frustratingly arbitrary. Players should lose because of their mistakes or decisions they've decided to take, not because of unavoidable bad luck [10]. Traditional designs for roguelikes explicitly warn against “insta-death” scenarios or unforeseen traps that kill the player without recourse. One example would be “The No Beheading Rule”, which is where players should not be eliminated by a single attack out of the blue, without some chance to react or prevent it [17]. This project harnesses this principle by avoiding cheap one-hit kills. The player should have subtle clues before triggering traps, and every enemy attack should be countered or mitigated with the right strategy, and any fatal situation generally originates from the player's own choices. By having the game this way, the challenge is always maintained, but players can feel that their deaths are “lesson filled” rather than frustrating. This learning curve is essential in roguelikes; as one author puts it, the genre involves “overcoming adversity, crashing up against insurmountable challenges and chipping away until you become good enough to conquer it” [8].
- **Replayability and “One More Run” Appeal:** Replayability is undoubtedly one of the defining trait of roguelikes – the anticipation that players will end up fail many times and still eagerly try again for another run. Achieving strong replayability involves more than just random levels; it pertains to the mechanics by which the game keeps the player motivated after each loss. Successful roguelikes tend to have a tight cycle of attempt, fail, learn, and retry, with minimal friction between each attempt [10]. This project uses this

ideology and is built on it. Having implemented instant restart mechanics, there are no lengthy cutscenes or loading times gating the next attempt. This design is deliberate, because, as Daniel Doan notes, quick restarts contribute greatly to the “just one more” addictiveness roguelikes have [10].

## 2.4 Historical and Industry Context

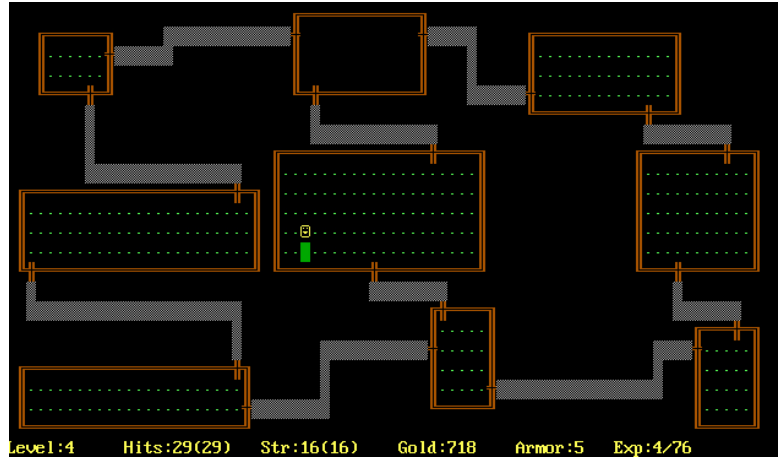


Figure 1. Shows a procedurally generated dungeon layout in the original Rogue (1980) [2]

The roguelike genre originated in the early 1980s with the game Rogue, which was strongly influenced by high-fantasy dungeon crawls such as Dungeons & Dragons and earlier text adventures [21]. Rogue introduced key ideas that would define the genre, as stated before. This philosophy was a stark departure from the more forgiving games that were present at the time and created a deeply challenging experience for players. Despite (or potentially because of) its difficulty, Rogue became extremely popular on UNIX systems in the 1980s, even leading it to be distributed with operating system releases [21]. This success led to many imitators and inspired the creation of a whole family of dungeon crawling games, leading to the term “Roguelike” to be coined and given to the genre [7].

During the 1980s and 1990s, roguelikes such as NetHack (1987), Angband (1990), and ADOM (1994) expanded the original concept by including complexity and new aspects. Significantly, the genre predominantly belonged to amateur developers and dedicated players during this period; these games frequently employed ASCII or rudimentary sprite graphics and were given at no cost. They prioritised intricate gameplay and “emergent” complexity over the aesthetics; for instance, NetHack gained notoriety for its extensive array of interactive object mechanisms and hidden secrets, cultivating a devoted community of players who exchanged insights on its mysteries. A pivotal point for the community occurred in 2008 with the Berlin Interpretation, an initiative by developers and enthusiasts to explicitly define the characteristics of a roguelike. The Berlin Interpretation outlined key characteristics of the genre, including random environment generation, turn-based gameplay, tile-based movement, resource management, and permadeath [21][7]. It also acknowledges that roguelikes are focused mainly on gameplay over the narrative conveyed to the player [21]. This definition helped set a benchmark, distinguishing true roguelikes from the more loosely related games that would soon follow.

In the 2000s, especially the 2010s, roguelike design ideas became prominent in popular gaming, leading to the emergence of “roguelites”—games that include fundamental aspects like procedural generation and permadeath but with alterations to attract a wider audiences [21]. Initial instances were Diablo (1996), which pioneered a widely embraced permadeath “hardcore mode”, and Dungeon Hack (1993), which integrated generative dungeons with a first-person viewpoint and save capability [21].



In the early 2010s, independent games such as *The Binding of Isaac* (2011) and *FTL: Faster Than Light* (2012) attained both critical acclaim and financial success by utilising roguelike gameplay in conjunction with approachable progression mechanisms. *The Binding of Isaac* captivated gamers with its vast array of unlockable material, whilst *FTL* integrated roguelike features with strategic spaceship management. This trend persisted with titles like *Spelunky* (2012), *Rogue Legacy* (2013), *Dead Cells* (2018), and *Hades* (2020), each creatively integrating roguelike elements into other genres. By 2020, the genre had gained significant notoriety to be referred to as "the year of the roguelike", highlighted by *Hades* receiving several Game of the Year awards for its captivating gameplay and narrative depth [8].

This evolution has ignited debates over definitions, especially in differentiating "roguelikes" from "roguelites". Traditionalists frequently use the Berlin Interpretation [8], whereas some developers currently use the term with greater flexibility. Irrespective of terminology, roguelike concepts have significantly influenced modern game development, equipping independent developers with resources to create expansive gameplay with limited resources [21].

## Chapter 3: Current Implementation

This chapter provides a detailed description of the key features and systems developed throughout the project, demonstrating how each of the objectives outlined earlier was implemented. It covers various aspects of the game, including procedural generation, combat mechanics, AI behaviours, user interface design, visual and audio elements, and overall game structure. Figures 2 and 3 here give a good overview of the project as a whole.

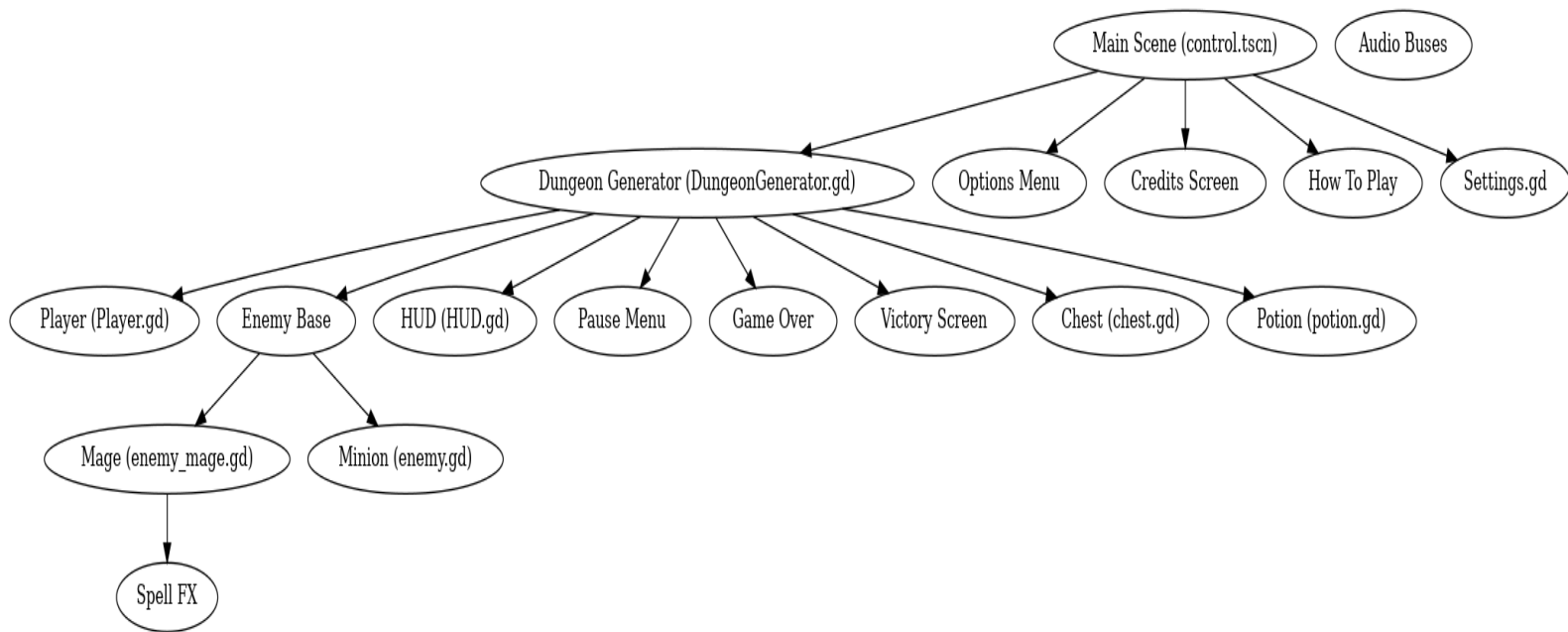


Figure 2. A node and script dependency diagram of the game, showcasing structure and script relationships

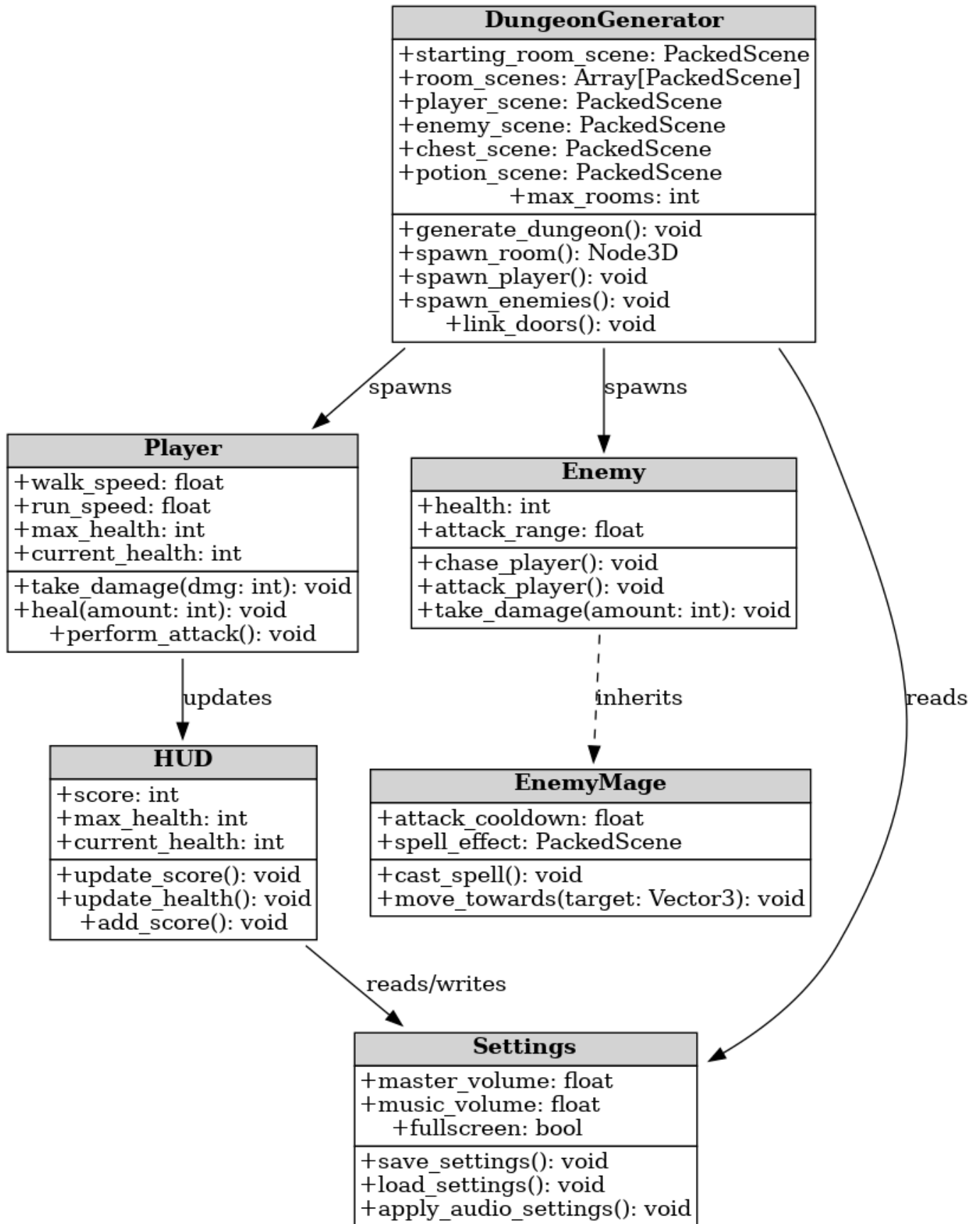


Figure 3. A UML class diagram illustrating the structure and interactions among the key game components

### 3.1 Procedural Dungeon Generation

Procedural generation serves as the backbone of this project, facilitating the creation of unpredictable and varied dungeon layouts, crucial for sustaining player engagement and replayability. The procedural generation method constructs dungeon levels and their contents dynamically (as seen in figure 4), greatly decreasing manual workload and providing a unique gameplay experience with each run.

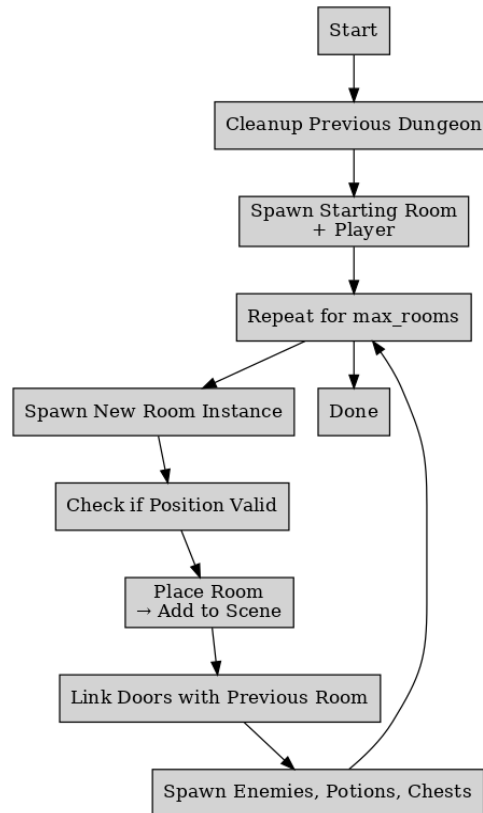


Figure 4. Flowchart of the dungeon generation algorithm, outlining the steps taken by *DungeonGenerator* to construct the layout

#### 3.1.1 Core Dungeon Generation

```

func generate_dungeon():
    if room_scenes.is_empty():
        push_error("ERROR: No rooms assigned to 'room_scenes' in
        Inspector!")
        return

    # Spawn the first room
    var start_room = starting_room_scene.instantiate()
    add_child(start_room)
    await get_tree().process_frame
    start_room.global_position = Vector3.ZERO
    spawned_rooms.append(start_room)
    used_positions[Vector3.ZERO] = true

    # Spawn Player
    if player_scene:
        player_instance = player_scene.instantiate()

        # Place player at the spawn point of the first room
        var spawn_point = start_room.find_child("PlayerSpawn",
        true, false)
  
```

```

        if spawn_point:
            player_instance.global_position = spawn_point.global_position
        else:
            player_instance.global_position = start_room.global_position + Vector3(0,
1, 0)

            add_child(player_instance)
    else:
        push_error("ERROR: No player scene assigned in Inspector!")

    # Spawn Additional Rooms
    for i in range(max_rooms - 1):
        var new_room = await spawn_room()
        if new_room:
            spawned_rooms.append(new_room)

```

Figure 5. Function handling the dungeon generation

The core of the procedural generation system utilises a modular grid-based approach combined with randomised room placement. During runtime, an initial room is instantiated at a fixed origin, establishing it as a foundation for further rooms to expand around it, as seen in figure 5. Subsequent rooms are sequentially generated and positioned adjacent to previously placed rooms. The selection of rooms is randomised from a curated list of preconfigured scenes, including a standard room and a trap room, ensuring variety in both visual and gameplay scenarios. The player is also spawned on a set marker node in the starting room.

### 3.1.2 Dungeon Door Connectivity

```

# --- Door Connection ---
func link_doors(prev_room, new_room, offset):
    var prev_door = get_doorway(prev_room, offset)
    var new_door = get_doorway(new_room, -offset)

    if prev_door and new_door:
        prev_door.set_meta("linked_room", new_door)
        new_door.set_meta("linked_room", prev_door)

```

Figure 6. Functions showing how the logic behind the dungeon doors work

In order to guarantee seamless and logical connectivity between rooms, a systematic approach was implemented. This was accomplished by establishing doorways in the cardinal directions of east, west, north, and south, and ensuring that there were valid connections between adjacent rooms. The function `link_doors()`, shown in figure 6, guarantees coherent navigation, preserving the integrity of the gameplay by utilising structured yet randomised pathways.

### 3.1.3 Dungeon Room Spawner

```

# --- Room Spawning ---
func spawn_room() -> Node3D:
    if room_scenes.is_empty():
        push_error("ERROR: No rooms assigned to 'room_scenes' in Inspector!")
        return null

    var selected_room = room_scenes.pick_random()
    var new_room_instance = selected_room.instantiate()
    if not new_room_instance:
        push_error("ERROR: Failed to instantiate room!")
        return null

    var room_size = get_room_size(new_room_instance)

```

```

var offset = choose_valid_room_offset(room_size)
var new_pos = spawned_rooms[-1].global_position + offset

# Snap to grid
new_pos.x = snapped(new_pos.x, room_size.x)
new_pos.z = snapped(new_pos.z, room_size.z)

if not used_positions.has(new_pos) and detect_open_door(spawned_rooms[-1]):
    new_room_instance.global_position = new_pos
    used_positions[new_pos] = true
    add_child(new_room_instance)
    await get_tree().process_frame

    # Spawn enemies inside this room
    spawn_enemies_in_room(new_room_instance)
    spawn_potion_in_room(new_room_instance)

if chest_scene and chest_room_count < max_chest_rooms and
spawned_rooms.size() >= 1:
    if spawn_chest_in_room(new_room_instance):
        chest_room_count += 1

    link_doors(spawned_rooms[-1], new_room_instance, offset)

    return new_room_instance

return null

```

Figure 7. Function overseeing the room generation for the dungeon

Each room generation involves:

- **Position validation** - Positions are selected from adjacent spaces around existing rooms, checking occupancy and available connections. This is done with the help of two more functions to help identify the size of the room, which uses a room boundary node in the scene, and to get a random room direction offset, to decide which position is valid for a room to spawn in.
- **Room placement**: Selected positions snap to a predetermined grid to prevent the overlapping of rooms and makes sure that there is consistent alignment.
- **Dynamic content spawning**: Following the room placement, enemies, chests and potions are dynamically instantiated using functions shown in figure 8, which are based on predefined spawn markers set in the scene, further enhancing replayability through unpredictability. This relies on calling the respective functions that `spawn_room()` calls:

```

# --- Enemy Spawning ---
func spawn_enemies_in_room(room: Node3D):
    await get_tree().process_frame

    var spawn_container = room.find_child("EnemySpawnPoints", true,
false)
    if spawn_container:
        var spawn_points = spawn_container.get_children()
        spawn_points = spawn_points.filter(func(p): return p is Marker3D
or p is Node3D)
        spawn_points.shuffle()

        var enemies_to_spawn = min(2, spawn_points.size())

```

```

    for i in range(enemies_to_spawn):
        if enemy_scene:
            var enemy_instance = enemy_scene.instantiate()
            var spawn_point = spawn_points[i]

            room.add_child(enemy_instance)
            enemy_instance.global_transform.origin =
spawn_point.global_transform.origin

            enemy_instance.connect("died", Callable(self,
"_on_enemy_died"))
            total_enemies += 1

# --- Potion Spawning ---
func spawn_potion_in_room(room: Node3D):
    if potions_spawned >= max_potions or potion_scene == null:
        return

    await get_tree().process_frame

    var spawn_container = room.find_child("PotionSpawnPoints", true,
false)
    if spawn_container:
        var spawn_points = spawn_container.get_children()
        spawn_points = spawn_points.filter(func(p): return p is Marker3D
or p is Node3D)
        spawn_points.shuffle()

        if spawn_points.size() > 0:
            var spawn_point = spawn_points[0]
            var potion = potion_scene.instantiate()
            room.add_child(potion)
            potion.global_transform.origin =
spawn_point.global_transform.origin
            potions_spawned += 1

```

Figure 8. Functions that spawn the items and enemies into the dungeon

### 3.1.4 Rational Behind Technical Decisions

The modular grid-based approach was selected after a thorough evaluation of alternative procedural generation methods, including pure random-walk algorithms and cellular automata. While pure random layouts were initially enticing due to their simplicity, they posed the risk of generating disconnected or unplayable dungeons, which disrupted player immersion and gameplay flow. Although cellular automata were effective in specific contexts, they were found to be excessively complex for structured 3D layouts that necessitated controlled spatial logic.

The modular approach that was chosen offered substantial benefits:

- Structure predictability: Guaranteed spatial coherence and navigability.
- Ease of debugging: Grid-based positions simplify the process of identifying and resolving overlaps or disconnections.
- Scalability: The ability to expand the project in the long term is facilitated by the addition of new modular room templates and configurations.

## 3.2 The Player

The player controller is the fundamental component of the game's interactivity and gameplay feedback along with it being the players means of interacting with the game. It was developed with a strong emphasis on immersive environmental feedback, combat precision, state-based animations, fluid movement, and responsive input management. The structure is scalable and simple to debug or extend due to its modular design and adherence to object-oriented principles.

### 3.2.1 Movement & Physics

```
# --- Movement + Physics ---
func handle_movement(delta: float) -> void:
    if is_dead:
        return

    var input_dir := Vector3.ZERO

    # Movement input
    if Input.is_action_pressed("move_forward"): input_dir.z += 1
    if Input.is_action_pressed("move_back"):    input_dir.z -= 1
    if Input.is_action_pressed("move_left"):    input_dir.x += 1
    if Input.is_action_pressed("move_right"):   input_dir.x -= 1

    # Normalize to prevent faster diagonal movement
    if input_dir.length() > 0:
        input_dir = input_dir.normalized()
        var current_speed := walk_speed
        var target_pitch := 1.0

        if Input.is_action_pressed("sprint"):
            current_speed = run_speed
            target_pitch = 1.5

    # Footstep SFX
    if not footstep_player.playing:
        footstep_player.pitch_scale = target_pitch
        footstep_player.play()
    else:
        footstep_player.pitch_scale = target_pitch

    # Rotation
    var target_rotation_y = atan2(input_dir.x, input_dir.z)
    rotation.y = lerp_angle(rotation.y, target_rotation_y, 10.0 *
delta)

    # Velocity
    velocity.x = input_dir.x * current_speed
    velocity.z = input_dir.z * current_speed
    else:
        velocity = Vector3.ZERO
        if footstep_player.playing:
            footstep_player.stop()

    move_and_slide()
    update_animation(input_dir)
```

Figure 9. Functions used to handle the players movement

The player navigates a 3D environment via normalised directional controls, accommodating both walking and running movements. Gravity is utilised to replicate grounding, and a terminal velocity



is imposed to avoid excessive vertical fall speeds, though it wasn't entirely necessary to add this as there aren't many situations where the player will be falling.

The sprint mechanism is managed by adjusting pace and the tone of footstep sounds in real time, offering both functional and sensory feedback. Movement input is normalised to avert diagonal speed changes, a frequent error in several games.

**Design Decision:** Initially, navmesh or pathfinding was contemplated; however, a direct control method was selected for its accuracy and reactivity in a fast-paced dungeon crawler, and navmesh is better suited to for other things such as enemy patrol routes. Rotation is seamlessly interpolated according to the input vector direction via `lerp_angle()`, which preserves visual fidelity during abrupt turns and gives a refined feel to the game.

### 3.2.2 Animation State Handling

```
# --- Animation ---
func update_animation(input_dir: Vector3) -> void:
    if is_dead or is_attacking or is_interacting:
        return

    if input_dir.length() > 0:
        var is_sprinting = Input.is_action_pressed("sprint")
        var anim = "Running_A" if is_sprinting else "Walking_A"

        if animation_player.current_animation != anim:
            animation_player.play(anim)
    else:
        if animation_player.current_animation != "Idle":
            animation_player.play("Idle")
```

Figure 10. Function that manages the players animation

Animations dynamically represent movement and interaction states. The `update_animation()` function, as seen in figure 10, chooses appropriate animations based on the context: idle, walking, sprinting, or attacking.

By distinguishing between movement and combat animations, the design mitigates animation conflicts and guarantees that attacks consistently take priority over movement when required. Transitions are seamless and align with player expectations.

### 3.2.3 Combat System

```
# --- Combat ---
func perform_attack() -> void:
    can_attack = false
    is_attacking = true

    await get_tree().create_timer(0.35).timeout
    $SlashSound.play()
    animation_player.stop()
    animation_player.play("1H_Melee_Attack_Slice_Diagonal", -1, 1.0)
    animation_player.seek(0, true)

    var enemies = get_enemies_in_range()
    for enemy in enemies:
        enemy.take_damage(attack_damage)

    await animation_player.animation_finished
    await get_tree().create_timer(0.2).timeout

    is_attacking = false
```

```

    animation_player.play("Idle")
    await get_tree().create_timer(attack_cooldown).timeout
    can_attack = true

func get_enemies_in_range() -> Array:
    var enemies := []
    var space_state = get_world_3d().direct_space_state

    var query := PhysicsShapeQueryParameters3D.new()
    var shape := SphereShape3D.new()
    shape.radius = attack_range
    query.shape = shape
    query.transform = Transform3D(Basis(), global_transform.origin +
    transform.basis.z * 1.5)

    var result = space_state.intersect_shape(query)
    for hit in result:
        if hit.collider != self and
hit.collider.has_method("take_damage"):
            enemies.append(hit.collider)

    return enemies

```

Figure 11. The function used to handle the combat systems for the player

The player uses a melee-focused combat system. The `perform_attack()` function utilises `PhysicsShapeQueryParameters3D` to play the animation, activate sound effects, and check for adversaries within a spherical range upon receiving an attack input.

Only enemies with a `take_damage()` method are the ones affected, promoting a loosely coupled system that easily supports various enemy types. A cooldown mechanism is implemented to prevent spamming, enforcing combat pacing.

**Design Decision:** Melee-based spherical overlap tests offer a scalable and clear alternative to collision triggers or ray casting. In top-down games, where the player may struggle with depth perception, this feature is essential for assuring responsive and satisfying combat by preventing precision-based misses.

### 3.2.4 Health, Healing, and Death Logic

A dynamic HUD display that uses graphical hearts to visually represent the player's condition is used to manage health. Damage generates feedback, including screen shake and sound effects, which adds significance to each strike.

The game provides a seamless transition from the gameplay to the failure state by triggering a death animation, disabling the HUD, and revealing the game over the screen upon death.

**Design Decision:** Instead of employing a numeric health indicator, segment-based hearts were selected to express the classic aesthetics of dungeon crawlers and to enhance visual readability. This decision promotes the clarity of gameplay and immersion.

### 3.2.5 Environmental Interactions

```

func get_interactable_chest():
    var space_state = get_world_3d().direct_space_state
    var query = PhysicsShapeQueryParameters3D.new()
    query.shape = SphereShape3D.new()
    query.shape.radius = 2.5
    query.transform = global_transform
    query.collision_mask = 1

```

```

var results = space_state.intersect_shape(query)
for result in results:
    if result.collider.has_method("interact"):
        return result.collider

return null

```

Figure 12. Function that handles the players interaction with items within the dungeon

The player interacts with items in the dungeon by calling upon this interact function, displayed in figure 12, which is used for both the chests and the potions. Animation timing is synced with interaction, ensuring visual and gameplay feedback are aligned.

**Design Decision:** This approach enhances the world immersion. It prevents abrupt or mechanical feeling interactions, resulting in a more polished experience.

### 3.3 The Enemy

The enemy system in this project has been built upon a modular and scalable AI framework, employing state-driven behaviour, animation synchronisation, and damage-based interactions with the player. Two separate enemy kinds were created: a Skeleton Warrior (melee) and a Skeleton Mage (ranged), both utilising the same underlying logic but contrasting in battle style and abilities. This dual-enemy concept enhances gameplay through diverse strategies, movement, and levels of difficulty.

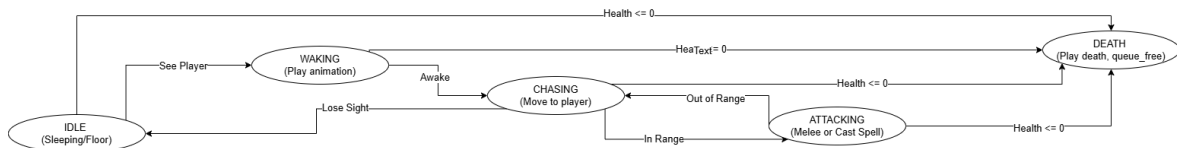


Figure 13. Finite State Machine diagram for enemy behaviour, depicting the transitions between states

#### 3.3.1 Structure & Inheritance

The enemies utilise a component-driven design approach, wherein both enemy types exhibit structural similarities yet are executed through distinct scripts for thorough behavioural management. Instead of rigidly tying them through the method of inheritance, which may lead to complexity, shared functionality such as navigation, damage, and detection was implemented individually for each class while maintaining consistency to provide clarity and flexibility.

This decision was intentional: preserving complete separation enabled each enemy to own a distinct animation controller, sound library, and reaction system without the danger of overriding parent functions. Nevertheless, on a broader scale, this pattern would be optimal for consolidation under a parent class or for employing a composition-based architecture.

#### 3.3.2 Skeleton Warrior (Melee AI)

```

# --- AI Processing ---
func _physics_process(delta: float) -> void:
    if current_state == State.DEAD or not is_instance_valid(player):
        return

    var player_visible = can_see_player()

    if not is_aware and player_visible:

```

```

        await wake_up()

    if is_aware:
        if player_visible:
            if current_state != State.CHASING:
                anim_player.play("Walking_D_Skeletons")
                current_state = State.CHASING
            else:
                if current_state != State.IDLE:
                    anim_player.play("Idle_Combat")
                    current_state = State.IDLE

        if current_state == State.CHASING:
            chase_player(delta)

        if is_aware and is_player_in_range():
            await attack_player()

# --- Wake-Up Sequence ---
func wake_up() -> void:
    if is_aware or is_waking_up:
        return

    is_waking_up = true
    if wake_sfx:
        wake_sfx.play()

    print("Skeleton waking up")
    anim_player.play("Skeletons_Awaken_Floor_Long")
    await anim_player.animation_finished

    anim_player.play("Walking_D_Skeletons")
    is_aware = true
    is_waking_up = false
    current_state = State.CHASING

```

Figure 14. The Script handling behaviour behind the enemy, using a state machine

The Skeleton Warrior employs a finite state machine with three fundamental states: IDLE, CHASING, and DEAD, as seen in figure 13 and 14. Its behaviour is activated when the player approaches its vision\_range. Upon the player being detected, the opponent transitions from a stationary position to an awakening animation accompanied by a sound effect, enhancing its presence and intensifying tension.

Once the skeletons have been activated, they begin chasing the player, with the use of directional movement and pathing logic. The rotation is handled manually to ensure that the enemy always faces the player during pursuit.

Once the enemy has moved close enough to the player and is within attack\_range, the skeleton plays a synchronised melee animation and checks if the player is still in range before the damage is applied to the player.

### 3.3.3 Skeleton Mage (Ranged AI)

```

await anim_player.animation_finished

var fx = spell_effect.instantiate()
fx.global_position = cast_origin.global_position
get_tree().current_scene.add_child(fx)

```

Figure 15. Code showing the mage enemy calling upon a coroutine to commence its attack

The mage utilises the same fundamental architecture while incorporating ranged attacks and coroutine-based spellcasting. Upon awakening, the mage advances towards the player but halts inside range to unleash a spell attack. The spell attack uses a coroutine to synchronise animation, visual effects, and audio. If the player is still within range after the casting animation has finished, damage is applied. The visual spell effect (MageSpellEffect.tscn) self-destructs after 0.5 seconds to decrease scene clutter and keep the immersion.

**Design Decision:** Coroutines (await) guarantee that damage and spell visuals appear only when casting is completed, removing desync between VFX and gameplay. This helps spellcasting feel more immediate and refined without requiring a rigid timeline system.

### 3.3.4 Audio, Feedback, and Immersion

Each enemy has its own SFX cues for walking, attacking, waking up, and dying integrated using nodes, as seen in figure 21. This layered audio design guarantees that enemies may be identified only via sound, increasing suspense and player awareness.

```
hud.add_score(200) #for melee
hud.add_score(350) #for mage
```

The HUD integration further ties combat into the overall game loop. On death, each enemy awards score points. This reinforces progression and gives the player consistent feedback for their actions.

## 3.4 Environmental Interaction

In addition to enemy engagements, the player interacts with the environment via dynamic items like healing potions and reward chests, which are intended to give diversity, strategy, and reward structure to exploration. These interactions are carried out by using proximity-based detection and customised interact() methods triggered by player input.

### 3.4.1 Potion Interaction

```
func interact() -> void:
    if used:
        return

    var player = get_tree().get_first_node_in_group("player")
    if player and player.has_method("heal"):
        player.heal(heal_amount)
        print("Potion used! Healed for", heal_amount, "HP.")

        # Play drink sound if available
        if has_node("PotionDrinkSound"):
            $PotionDrinkSound.play()

    used = true
    await get_tree().create_timer(1.35).timeout
    queue_free() # Remove the potion after sound plays
```

Figure 16. Interaction function for the potion to be used by the player

Potions are a restorative mechanism that allows players to regain their health while traversing dungeons. The potion object is put at random spawn spots across the rooms and activates when the player hits the "interact" key ('E') near it. When the player interacts with it, a specific quantity (20 HP) of healing is applied, a sound effect is played, and the potion vanishes.

This system offers tactical depth; players must manage their position and health awareness, deciding when to engage and when to retreat and recuperate.

**Design Decision:** A simple used flag prohibits repeated activations, ensuring potions only provide one benefit. The short delay before removal, along with the auditory cue, provides clear feedback while maintaining immersion.

### 3.4.2 Chest Interaction

```
func interact():
    if is_open:
        return # Already opened, skip

    is_open = true

    # Play chest open animation
    if $AnimationPlayer:
        $AnimationPlayer.play("open")

    # Play open sound
    if has_node("ChestOpenSound"):
        $ChestOpenSound.play()

    # Update the HUD score
    var hud = get_tree().get_first_node_in_group("hud")
    if hud and hud.has_method("add_score"):
        hud.add_score(chest_value)
```

*Figure 17. Chest interaction script*

Chests serve as reward objects. When the player interacts with a chest, it produces an animation and a sound effect before awarding points through the HUD system, displayed in figure 17. Each chest provides a considerable score boost (750 points), strengthening the reward loop and encourages exploration.

Chests are implemented using StaticBody3D nodes and an interact() function. The is\_open Boolean flag prohibits multiple activations, guaranteeing that the chest cannot be farmed or re-triggered. The animation and sound player nodes make the interaction more haptic and satisfying.

## 3.5 User Interface

The user interface was thoroughly developed to strike a balance between clarity, utility, and visual coherence in both the main menu system and the in-game HUD. These UI components function as both a feedback mechanism and a driver of immersion, guiding the player through the game's different states via subtle transitions, auditory feedback, and consistent visual structure.

### 3.5.1 Main Menu System

```
extends Control

## --- Main Menu Script ---
## Handles button interactions, transition animations, and scene changes.

func _ready():
    # Play initial background fade-in
    menu_anim.play("BG fade_in")
    await menu_anim.animation_finished

    if menu_anim.has_animation("fade_in"):
        menu_anim.play("fade_in")

    # Connect all button signals
    start_btn.pressed.connect(_on_start_button_pressed)
```

```

options_btn.pressed.connect(_on_options_button_pressed)
exit_btn.pressed.connect(_on_exit_button_pressed)
htp_btn.pressed.connect(_on_HTP_button_pressed)
credits_btn.pressed.connect(_on_credits_button_pressed)

func _on_start_button_pressed():
    _play_click_and_transition("res://Scenes/DungeonGenerator.tscn")

func _on_HTP_button_pressed():
    _play_click_and_transition("res://Scenes/HowToPlay.tscn")

func _on_credits_button_pressed():
    _play_click_and_transition("res://Scenes/credits.tscn")

func _on_options_button_pressed():
    click_player.play()
    await click_player.finished

    menu_anim.play("BG_fade_out")
    await menu_anim.animation_finished

    var options_scene = preload("res://Scenes/OptionsMenu.tscn")
    var options_instance = options_scene.instantiate()
    add_child(options_instance)

    # Bring menu background back after options overlay
    menu_anim.play("BG_fade_in")
    await menu_anim.animation_finished
    if menu_anim.has_animation("fade_in"):
        menu_anim.play("fade_in")

func _on_exit_button_pressed():
    click_player.play()
    await click_player.finished
    get_tree().quit()

```

*Figure 18. The main menu script*

The main menu is constructed as a Control node, which handles scene navigation, button interactions, and menu animations. Upon launch, it plays a background fade-in animation to provide the player a polished first impression. See figure 18.

To speed up setup and prevent miswiring in the editor, each button (Start, Options, How To Play, Credits, Exit) is programmatically linked in `_ready()`. Transitions to different scenes (such as beginning the game or opening options) are handled by the specific `_play_click_and_transition()` function. Many of the other menu scripts are handled similar to this and are not too dissimilar, only changes being what buttons they have and what information they display to the player. Figure 19 displays the flow of the main menu, and what menus are linked together.

**Design Decision:** Rather than switching scenes suddenly, each change is accompanied by a fade animation and sound cue, which reinforces the game's tone and allows for seamless navigation. This improves the user's psychological experience of navigating a well-designed, immersive gaming world.

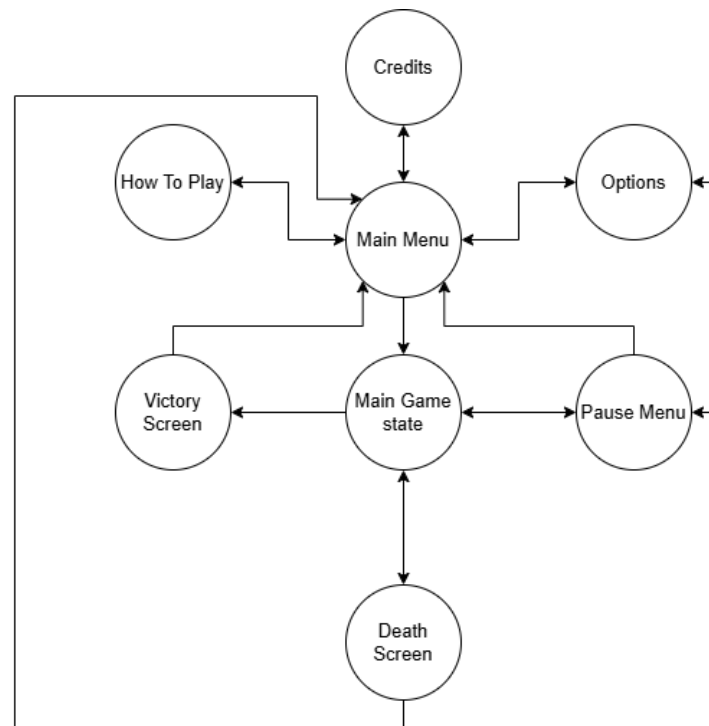


Figure 19. Menu Interaction Flowchart of the game's UI navigation, illustrating transitions between the different menus

### 3.5.2 HUD: Health and Score Management

extends CanvasLayer

```

func _ready() -> void:
    high_score = load_high_score()
    score = 0 # Reset score on game start or retry
    update_health(current_health, max_health)
    update_score_display()

# --- Score Management ---
func add_score(amount: int) -> void:
    score += amount
    if score > high_score:
        high_score = score
        show_new_high_score_message()
        save_score()

    update_score_display()

    if has_node("Points"):
        $Points.play()

func update_score(value: int) -> void:
    score = value
    score_label.text = "Score: " + str(score)

```



```

func update_score_display() -> void:
    if score_label:
        score_label.text = "Score: %d\nHigh Score: %d" % [score,
high_score]

func show_new_high_score_message() -> void:
    var label := $NewHighScoreLabel
    if label:
        label.visible = true
        await get_tree().create_timer(2.0).timeout
        label.visible = false

# --- Health Management ---
func update_health(current: int, max: int) -> void:
    current_health = current
    max_health = max

    for i in range(health_container.get_child_count()):
        var heart = health_container.get_child(i)
        heart.visible = i < current

# --- Save / Load ---
func save_score() -> void:
    var config := ConfigFile.new()
    config.set_value("Save", "high_score", high_score)
    config.save(save_path)

func load_score() -> void:
    # Load score from separate file
    if FileAccess.file_exists("user://score.save"):
        var file = FileAccess.open("user://score.save", FileAccess.READ)
        score = file.get_var()

    # Load high score
    high_score = load_high_score()
    update_score_display()

func load_high_score() -> int:
    var config := ConfigFile.new()
    if config.load(save_path) == OK:
        return config.get_value("Save", "high_score", 0)
    return 0

```

*Figure 20. The HUD script handling health, score and score saving*

The in-game HUD is built with a CanvasLayer, so it remains visible regardless of camera movement or the surrounding changes. It measures the player's health with a graphical heart container system and shows both the current and top score.

This heart system is scalable, with each visible heart representing 20 HP, providing immediate, straightforward feedback on the player's survival situation. The usage of segmented hearts rather of bars preserves battle clarity while also evoking the genre's roots.

The scoring system is dynamic, and it allows high score storing using Godot's ConfigFile system.

After setting a new record, a "New High Score" label appears briefly, emphasising a sense of accomplishment and growth.

## 3.6 Visual & Audio Design

The project's visual and audio design was purposefully chosen to match the game's fast-paced dungeon crawling action and enhancing the immersion. While the game has a low-poly stylised theme, it also incorporates modern design approaches such as subtle lighting effects, animations, and responsive audio cues to improve the experience.

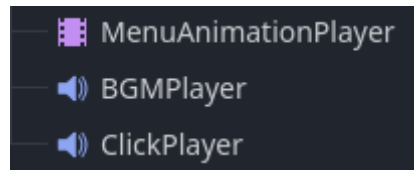


Figure 21. A Screenshot of GoDots Nodes for Audio and Animation

### 3.6.1 Visual Identity and Animation

The game has a consistent visual style centred on a fantasy dungeon atmosphere, with skeletons, items, and rooms all designed to preserve visual coherence. Player and enemy characters are animated with purpose-driven states — idle, walking, attacking, dying — ensuring visual clarity and smooth transitions during gameplay.

```
anim_player.play("Skeletons_Awaken_Floor_Long")  
await anim_player.animation_finished
```

Animations are handled through Godot's AnimationPlayer nodes and tightly integrated with logic. Wake-up sequences, combat, and death all have distinct animations for enemies. The player also switches between walking, running, attacking, and interacting actions depending on the game state. This layered animation approach delivers smooth input while also enhancing the individuality of each object.

```
if camera and camera.has_method("shake_camera"):  
    camera.shake_camera()
```

Camera shaking, handled during the player's take\_damage() method, offers an additional layer of visceral impact when the player is hit. This effect provides a tactile feeling of danger and urgency, along with an active response to show players that they have taken damage and need to play cautiously.

### 3.6.2 Sound Design and Feedback

Audio design plays a critical role in the player's sensory feedback loop. The game uses distinct sound effects for key interactions:

- Potion drinking
- Chest opening
- Footsteps that speed up while sprinting
- Enemy waking sounds

- Player attack slashes and spellcasting

These cues help differentiate enemy types, reinforce the consequences of actions, and anchor the game's atmosphere. For example, the Skeleton Mage's casting sequence includes a sound cue and visual effect.

All sounds are routed through dedicated audio buses (Master, Music, and SFX), which are managed by the SettingsManager singleton, enabling players to modify volume levels – a respect to accessibility and user desire.

## 3.7 Settings Management

A separate SettingsManager singleton was created to offer permanent control over audio and visual choices, improving both the user experience and technical polish. It is accessible from all game scenes and guarantees that player-selected parameters, such as audio levels or screen mode, are automatically loaded and implemented during runtime.

### 3.7.1 Autoload Singleton Structure

```
# --- Audio Settings ---
var master_volume := 1.0
var music_volume := 1.0
var sfx_volume := 1.0

# --- Video Settings ---
var fullscreen := false
var vsync := false

# --- Config Path ---
var config_path := "user://settings.cfg"

func _ready() -> void:
    # Load settings on game start
    load_settings()

func save_settings() -> void:
    # Saves current settings to config file
    var config = ConfigFile.new()
    config.set_value("Audio", "master", master_volume)
    config.set_value("Audio", "music", music_volume)
    config.set_value("Audio", "sfx", sfx_volume)
    config.set_value("Video", "fullscreen", fullscreen)
    config.set_value("Video", "vsync", vsync)
    config.save(config_path)

func load_settings() -> void:
    # Loads settings from config file and applies them
    var config = ConfigFile.new()
    if config.load(config_path) == OK:
        master_volume = config.get_value("Audio", "master", 1.0)
        music_volume = config.get_value("Audio", "music", 1.0)
        sfx_volume = config.get_value("Audio", "sfx", 1.0)
        fullscreen = config.get_value("Video", "fullscreen", false)
        vsync = config.get_value("Video", "vsync", false)
```

```

        apply_video_settings()
        apply_audio_settings()

func apply_video_settings() -> void:
    # Applies current video settings (fullscreen and vsync)
    DisplayServer.window_set_mode(
        DisplayServer.WINDOW_MODE_FULLSCREEN if fullscreen else
DisplayServer.WINDOW_MODE_WINDOWED
    )
    DisplayServer.window_set_vsync_mode(
        DisplayServer.VSYNC_ENABLED if vsync else
DisplayServer.VSYNC_DISABLED
    )

func apply_audio_settings() -> void:
    # Applies volume settings to respective audio buses
    AudioServer.set_bus_volume_db(AudioServer.get_bus_index("Master"),
linear_to_db(master_volume))
    AudioServer.set_bus_volume_db(AudioServer.get_bus_index("Music"),
linear_to_db(music_volume))
    AudioServer.set_bus_volume_db(AudioServer.get_bus_index("SFX"),
linear_to_db(sfx_volume))

```

*Figure 22. The autoload singleton script used to handle the players options*

The SettingsManager is set up as an autoload singleton in the Godot project settings. Its design ensures that settings remain consistent across scenes without the need to manually transmit data or reload configuration files several times. The ConfigFile API is used to store and retrieve values in a local user://settings.cfg file, preserving them between sessions.

### 3.7.2 Audio Control System

Audio levels are managed through three sliders:

- Master Volume
- Music Volume
- SFX Volume

These values are converted to decibel scale and applied to audio buses via the AudioServer API.

### 3.7.3 Video Preference

Fullscreen and vertical sync toggles are handled through DisplayServer settings. This allows the game to adapt to a variety of hardware configurations and display preferences, a feature that is increasingly expected in modern games.



*Figure 23. A screenshot from the game showing the player in the dungeon, with the score and health visible, this is the first thing players see once they press play*

## Chapter 4: Software Engineering

This project's software engineering methodology focuses on scalability, maintainability, and robustness in game design and implementation. This is especially essential because this is my first time utilising the Godot engine, a powerful yet unique tool with a node-based architecture and the GDScript programming language. This necessitated extensive consideration and exploration to determine its capabilities, limits, and best practices for attaining the project's objectives.

Throughout the development process, I emphasised modular design principles to ensure that separate components, such as combat, procedural generation, and AI systems, could be designed, tested, and integrated smoothly. Godot's open-source nature and copious documentation provided essential learning resources; nonetheless, integrating complex technologies such as procedural content creation and dynamic AI necessitated adapting old methods to this new context.

### 4.1 Software Methodologies

For this project, I used an Agile approach to oversee the development process. Agile was chosen because of its iterative and flexible nature, which complements the dynamic requirements of game production. By splitting the project down into small chunks, I was able to focus on individual aspects like enemy behaviour and combat mechanics while still adding regular testing and feedback. This strategy enabled refinement and adaptability, ensuring that the game progressed successfully over time.

Other techniques, such as the waterfall model, were examined but found unsuitable for this project. Waterfall's linear form would have made it difficult to handle changes or unexpected obstacles that are typical in creative undertakings such as game development. Similarly, while Extreme Programming (XP) places a strong emphasis on testing and pair programming, its stringent principles and requirement for frequent cooperation would have been difficult to accomplish in a solo development setting.

Agile gave me the ideal combination of structure and flexibility, allowing me to respond to difficulties, enhance features, and prioritise jobs quickly. This iterative strategy ensured that every stage of development—from prototyping and testing to integration—was in line with the project's objectives and timeframes.

In addition to Agile, I applied Object-Orientated Programming (OOP) principles throughout the development process. OOP offered a robust foundation for developing modular and reusable code, which is critical for a project of this size. For example, enemy behaviour and player mechanics were implemented as separate classes to facilitate maintenance and scalability. This method supported Agile by keeping the codebase organised and flexible to changes.

To keep track of progress across these Agile development cycles, I used Trello as a lightweight project management board. Each feature (e.g., "Implement AI Vision Logic", "Add Chest Interaction", "Design Pause Menu") was organised into task columns like "To Do", "In Progress", and "Completed." This visual format allowed for easy sprint planning and ensured steady progress even when balancing university work. A typical development sprint would span a few days to a week, focusing on a self-contained system such as enemy AI or procedural generation, followed by testing and adjustment before moving to the next module.

## 4.2 Testing

Testing has been an important part of the development process, ensuring that each feature works as expected and fits seamlessly into the game. A multi-tiered testing technique was implemented, including unit testing, integration testing, and playtesting, along with other testing plans such as feasibility testing, which allowed for a thorough review and refining of the game's mechanics and systems.

The combat mechanism, opponent behaviour, and procedural generation methods were all tested individually. These tests guaranteed that each module carried out its intended role in isolation, discovering mistakes early in the development process. To prevent unforeseen behaviours, the enemy's attack range, movement patterns, and damage logic were all extensively examined. Similarly, the procedural generation process, when implemented, was tested thoroughly with a wide range of settings to verify that each dungeon layout generated was both cohesive and passable.

Integration testing was performed to ensure that multiple game systems operated together. For example, the interactions between the player, opponents, and the health system were thoroughly tested to ensure accurate health reduction, proper game-over triggers, and smooth transitions between fighting and non-combat stages. This process was critical for discovering problems caused by dependencies or unanticipated interactions across modules, such as collision bugs or inaccurate health updates during combat.

Playtesting entailed manually playing the game to evaluate its user experience, gameplay balance, and overall flow. This step was extremely useful for detecting subtle issues, like pacing issues, imprecise visual feedback, or unexpected difficulty spikes, that automated testing would have missed. Playtesting also gave qualitative feedback on areas for development, for example, things such as improving combat mechanics, changing enemy AI behaviours, and adjusting the complexity of procedurally created levels. These give a good insight to how the players would play the game and prepare for cases that you do not expect, in a sense, play the game to break it. These sessions emphasised ways to improve the game's accessibility and overall enjoyment.

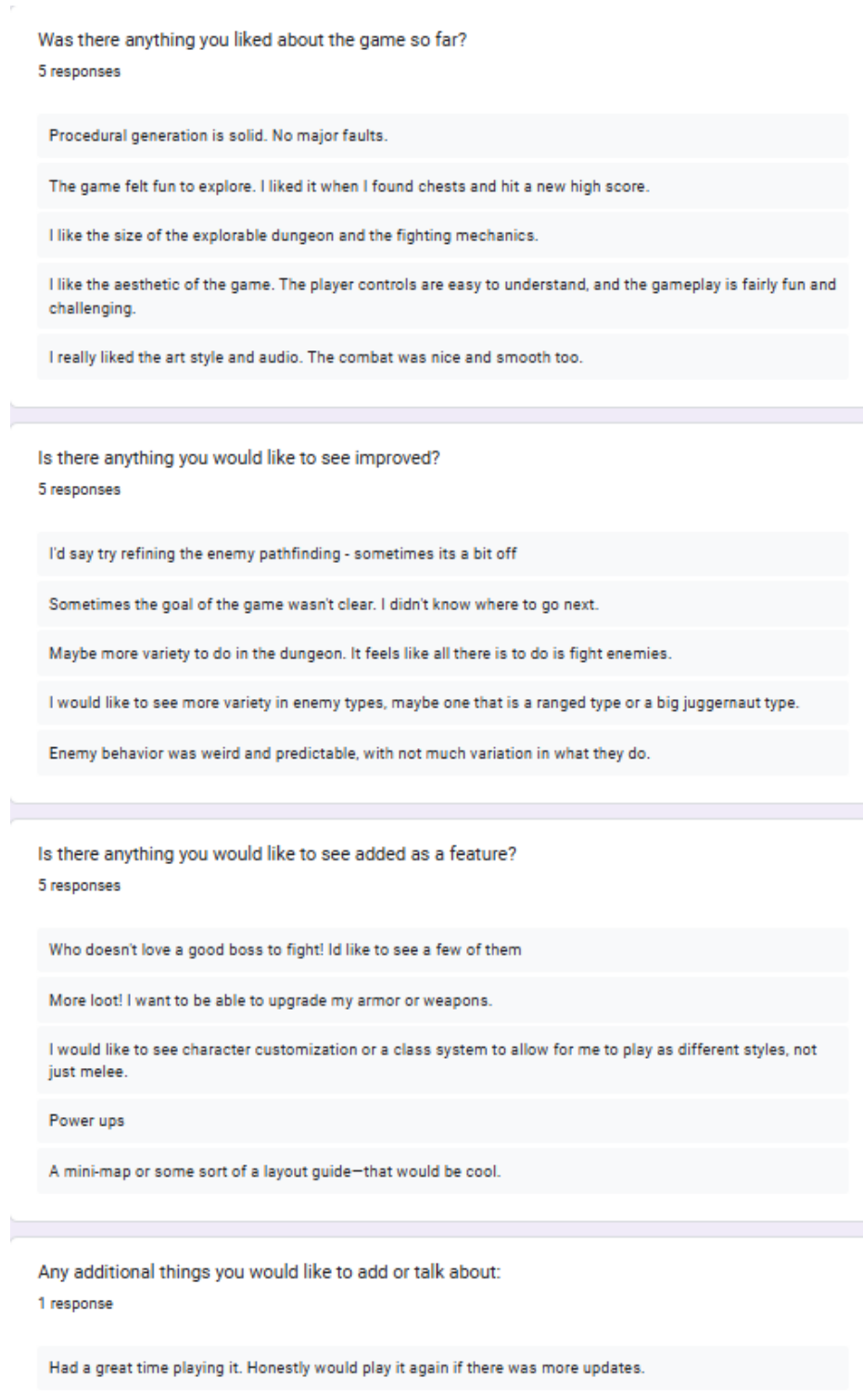
Feasibility Testing was planned to determine the game's performance and compatibility with various machine setups. Load times, memory utilisation, and frame rate stability are all important elements in providing a seamless and comfortable playing experience. Testing on computers with varied hardware capabilities to uncover optimisation opportunities keeps the game accessible to a wider audience. For example, profiling techniques built into the Godot engine will be utilised to identify performance bottlenecks such as unoptimised assets or wasteful code routes. The procedural generation mechanism will receive special attention to guarantee that it runs smoothly and does not cause delays during level transitions. These tests will provide valuable insights into how the game works under various settings, guiding any necessary improvements.

Additionally, edge-case testing was carried out to determine how the game handles uncommon conditions like quick inputs, overlapping opponent attacks, or excessively huge randomly generated landscapes. This ensures that the game is resilient and reliable under a variety of scenarios.

By iteratively employing these testing methodologies, the project maintains a high level of functionality and polish, resulting in a seamless and engaging player experience. The testing methods are used throughout development especially when met with new challenges as features are introduced or enhanced. This iterative testing approach, combined with player input and feasibility testing, guarantees that the end result is both dependable and engaging.

### 4.2.1 Player Testing Results (Forms)

In-house play testing during the winter break was conducted by no data or information was collected, only verbal feedback was provided to me after the play testers had finished. Forms for the final playtests were recorded once development has concluded. The responses recorded from this are presented below in figure 24:



The figure displays a summary of player feedback from a Google Form. It is organized into four sections, each with a question, a response count, and a list of individual responses in light blue boxes.

**Section 1:**  
Question: Was there anything you liked about the game so far?  
Responses: 5 responses  
- Procedural generation is solid. No major faults.  
- The game felt fun to explore. I liked it when I found chests and hit a new high score.  
- I like the size of the explorable dungeon and the fighting mechanics.  
- I like the aesthetic of the game. The player controls are easy to understand, and the gameplay is fairly fun and challenging.  
- I really liked the art style and audio. The combat was nice and smooth too.

**Section 2:**  
Question: Is there anything you would like to see improved?  
Responses: 5 responses  
- I'd say try refining the enemy pathfinding - sometimes its a bit off  
- Sometimes the goal of the game wasn't clear. I didn't know where to go next.  
- Maybe more variety to do in the dungeon. It feels like all there is to do is fight enemies.  
- I would like to see more variety in enemy types, maybe one that is a ranged type or a big juggernaut type.  
- Enemy behavior was weird and predictable, with not much variation in what they do.

**Section 3:**  
Question: Is there anything you would like to see added as a feature?  
Responses: 5 responses  
- Who doesn't love a good boss to fight! Id like to see a few of them  
- More loot! I want to be able to upgrade my armor or weapons.  
- I would like to see character customization or a class system to allow for me to play as different styles, not just melee.  
- Power ups  
- A mini-map or some sort of a layout guide—that would be cool.

**Section 4:**  
Question: Any additional things you would like to add or talk about:  
Responses: 1 response  
- Had a great time playing it. Honestly would play it again if there was more updates.

Figure 24. Summary of post-test player feedback collected via Google Form, taken place after development had concluded



## 4.3 Code Quality

Throughout the development process, excellent code quality has been prioritised to ensure the project's robustness, maintainability, and scalability. Several tactics and best practices have been used to reach this goal.

### 4.3.1 Modular Design

Code is divided into small, self-contained modules that each handle a single aspect of the game, such as player movement, combat, AI behaviour, or procedural creation. This modularity allows for easy debugging and updating, as changes in one module have minor impact on others. A key example of this modularity is the shared `interact()` method used across different objects like potions and chests. Each of these items handles its own internal logic but responds to the same interaction system, allowing the player to engage with them seamlessly without duplicating code. This abstraction made it easy to expand the game's interactable content with minimal overhead.

### 4.3.2 Coding Standards

Consistent naming conventions, unambiguous comments, and structured code formatting have been implemented to improve readability and collaboration. Variables and functions are named descriptively, such as `take_damage()` or `get_enemies_in_range()`, so the code's purpose is readily apparent. Comments and documentation are added to help clarify difficult logic.

### 4.3.3 Error Handling

Error management is carefully considered to avoid crashes or unexpected behaviour during gameplay. Functions contain checks for invalid states, such as ensuring that the player or enemy instance is legitimate before acting. For example, `is_instance_valid()` is used to check that objects exist before accessing their attributes or methods.

### 4.3.4 Testing and Iteration

The project includes unit and integration testing as part of the development process. To detect and address errors as soon as possible, both automated and manual tests are run on a regular basis. This iterative method ensures that new features are implemented while maintaining old functionality. A main feature was to use clear and helpful print statements to help figure out what sections of code were running when, drastically assisting the debugging process.

## 4.4 Version Control

Version control is an important part of the project since it ensures an organised and reversible development process. GitLab was chosen as the project's primary version control platform. GitLab's comprehensive capabilities, like branching, issue tracking, and continuous integration, make it a great solution for efficiently managing codebases.

Commit management is essential for tracking changes and recording progress. Descriptive and relevant commit messages guarantee that every change is fully documented, providing a chronological record of the development process. This method is especially valuable when returning to a project after a long absence, such as working on other university courses, because the thorough commit messages serve as a road map for understanding what was implemented, amended, or fixed.

Furthermore, GitLab's version control features make reverting changes simple. If a problem or unexpected behaviour is encountered, Git's ability to roll back commits or reset the codebase to a previous stable state allows for quick recovery. This has been quite useful during testing and feature implementation, as experimental modifications might occasionally impair functioning. Reverting to a known good state allows faults to be handled without jeopardising overall progress.

In terms of workflow, I committed code changes frequently — often after completing specific features or testing milestones. This helped isolate bugs and ensured that progress was tracked at a granular level.

## 4.5 Requirements Analysis

The project began with a clear description of essential needs to guide development and manage complexity. These were divided into three categories: minimum viable product (MVP) features, stretch goals, and nice-to-have additions. This tiered approach helps keep expectations reasonable while still allowing for innovation and progress. Given that this was my first time using the Godot engine, my first focus was on building attainable, core systems that could be developed iteratively.

### 4.5.1 Minimum Viable Product (MVP) Requirements

The MVP represented the heart of the gameplay loop, the mechanisms required for the game to work and feel complete. They included:

- **Procedural dungeon generation:** Randomly connected rooms with enemy and item placement.
- **Player movement and combat:** A responsive melee system with simple animations and health tracking.
- **Enemy AI:** Basic pursue and attack behaviours, as well as damage and death logic.
- **Healing items (potions):** Collectible items that could restore the player health.
- **HUD and health display:** A functional user interface that displays health and score, as well as game-over triggers.
- **Basic menus:** Main menu, pause menu, and transitions between scenes.

These requirements were prioritised in early sprints and their effective implementation meant that the fundamental mechanics — navigation, combat, and interaction — worked together before any new features were examined.

### 4.5.2 Stretch Goals and Additional Features

After stabilising the MVP, the work focused on more immersive and quality-of-life elements, such as:

- **Enemy variety:** Adding a skeleton mage with ranged attacks and casting animations.
- **Score tracking and high score saving:** Persistent saving using ConfigFile to add replay value.
- **Visual/audio polish:** Footsteps, death sounds, animations, camera shake, and transition effects.
- **Options menu:** Allowing the player to adjust volume, toggle fullscreen, and enable/disable VSync.

Each of these were chosen for their capacity to improve the feel of the gameplay or increase engagement without jeopardising stability. The implementation of them was timed to coincide with available development capacity, ensuring that it would not disrupt core system testing and polishing.

### 4.5.3 Deferred or Removed Features

A few originally planned features were postponed or removed due to time constraints, technical challenges, or the importance of stability. They included:

- **Inventory system with item slots:** While potions and chests were added, a full inventory system with equipment management was deemed too complicated for the present timeline.
- **Puzzle mechanics and traps:** Basic trap room placeholders were built, but an interactive puzzle system was postponed, minimising scope creep.
- **Boss encounters:** A multi-phase boss with sophisticated behaviour was originally intended, but it was replaced by a victory screen system once all enemies had been beaten.

These decisions were made as part of the Agile process, which involved adjusting scope based on evolving understanding of complexity and time limitations. The modular codebase enables these capabilities to be introduced in future versions if needed.

### 4.5.4 Evolving Requirements Through Development

As development proceeded and early systems were evaluated, certain criteria were revised based on arising new information. For example:

- Originally, all adversaries would utilise the same script; however, to enable variation, inheritance was removed in favour of independent AI logic for melee and ranged varieties.
- The procedural generation method grew beyond simple room layout to allow item spawning (enemies, potions, treasure) and door connecting using random offsets and accessible directions.

This adaptability was critical for managing a single development project on a tight deadline, and it reflects a practical approach to software engineering in the game development setting.

The utilisation of staged, developing requirements enabled structured development while remaining flexible. By prioritising fundamental mechanics and carefully integrating the stretch features, the project was able to be accomplished with its objectives, without overreaching. This technique demonstrates a grasp of both project management best practices and the significance of scope control in software development, particularly in creative, exploratory areas such as game design.

## 4.6 Godots Architecture and Technical Design

Throughout the development of this project, Godot's distinctive architecture had a considerable impact on the structural and design choices. Unlike traditional engines, which depend largely on inheritance hierarchies, Godot uses a node-based scene structure that prioritises composition over inheritance – a paradigm that is especially well-suited to game creation. Adopting this paradigm required a mentality adjustment, but it resulted in highly modular, reusable, and decoupled systems that exactly fit with the project's scalability and maintainability requirements.

### 4.6.1 Node-Based Scene System

Godot's architecture is built around a tree-based node system, with each item in a scene consisting of nodes that inherit from a basic Node class. This framework encourages scene composition by allowing developers to consider huge collections of nodes as standalone scenes that can be instanced and reused.

For example, the Player character was made up of:

- CharacterBody3D for Physics/Motion
- AnimationPlayer for managing animations.
- AudioStreamPlayer3D for Footstep SFX
- A script is connected to manage input, health, and interactions.

This framework makes it simple to modify or expand the player's behaviour without rewriting code or duplicating assets. Enemies shared a similar modular design, allowing for diversity (e.g., melee versus ranged AI) while maintaining fundamental structure and signals.

#### 4.6.2 Signals and Event Handling

Godot's design is notable for its robust signal system, which is used to decouple systems and reduce interdependence. Rather than having hardcoded relationships to objects, signals enabled flexible communication between various elements of the game. For example:

- When an enemy's health level drops to zero, they transmit a dead signal.
- The dungeon generator uses this signal to track progress and trigger the victory screen.
- The HUD responds to events such as score changes and health updates with similarly separated logic.

This method kept the code clean and scalable, especially when new features (such as ranged enemies and potion pickups) were implemented.

#### 4.6.3 Scene Instancing and Procedural Generation

Scene instancing was an important aspect of developing the procedural generation of the dungeon. Instead of generating rooms or enemies manually, the game used pre-made scenes (e.g., `dungeonRoom.tscn`, `room_trap.tscn`) and instanced them dynamically at runtime. These scenes were chosen and set using spatial logic, and they were subsequently filled with child nodes like spawn positions, chests, or enemies.

Each scene was self-contained, with its own logic and structure, allowing rooms to be put into the world without requiring further configuration. This method accelerated development and testing while making the architecture adaptable for future changes such as additional room types or environmental hazards.

#### 4.6.4 Separation of Concerns

Following best practices in game design, logic was divided across many scripts and layers:

- Interaction logic (e.g., `interact()` methods) was centralised and shared across items such as chests and potions.
- The Visuals and audio were handled by specific node children (e.g., `AnimationPlayer`, `AudioStreamPlayer3D`) rather than being included in gameplay code.
- The `DungeonGenerator` script regulated game flow (win/loss states, enemy numbers), keeping control centralised without bloating player or enemy code.

This careful separation of responsibilities resulted in a cleaner codebase that was easier to maintain and test.

Godot's architecture not only helped quick development, but it also promoted design concepts consistent with professional software engineering, such as modularity, separation of concerns,

decoupling, and reusable components. The mix of nodes, signals, instancing, and singletons resulted in a flexible foundation that could be expanded to accommodate new features without disturbing any current systems. As a consequence, the project was able to reach ambitious technical goals while maintaining a manageable and strong framework, offering an excellent learning experience and setting the groundwork for more advanced game development projects in the future.

## Chapter 5: Professional Issues

This chapter delves into the legal, ethical, and professional issues that must be addressed as the dungeon crawler game moves forward. Although this game was created as an academic project, professional standards were followed and considered throughout to match industry expectations and guarantee appropriate software development methods.

### 5.1.1 Intellectual Property and Licensing

Respect for intellectual property (IP) is an important aspect of professional software development. Throughout the project, all third-party materials, including sound effects, textures, and fonts, were either self-created, sourced from royalty-free asset packs, or Creative Commons-licensed. Where open-source resources were used (e.g., CC0 or CC-BY licensed music or sound), proper acknowledgement was provided. This approach ensured legal compliance and ethical standards, in turn avoiding plagiarism.

Furthermore, the Godot engine is released under the MIT licence, which permits unlimited use, modification, and distribution, consistent with the project's open-source philosophy. Understanding license conditions helped to avoid copyright infringement and emphasised the need for software law compliance.

### 5.1.2 Data Protection and User Privacy

Although the game does not collect or send any players personal information, it was important to consider appropriate data handling. Local data is saved using Godot's ConfigFile system to record high scores and settings. These files are saved in the user:// directory, sandboxed for each user, and no sensitive information is stored.

If the game had network elements or online leaderboards, applicable data protection rules such as the UK GDPR and the Data Protection Act 2018 would have been more thoroughly applied to guarantee that player data was handled lawfully, transparently, and securely.

### 5.1.3 Inclusivity and Accessibility

In professional game development, diversity is becoming important. While the scope of this project was restricted, early decisions focused on making the game more accessible. For example:

- Audio and visual feedback were added to critical activities such as healing, attacking, and picking up things.
- Font sizes and UI contrast were assessed to guarantee basic readability.

Although more sophisticated accessibility features (such as colour-blind modes, complete button remapping, and difficulty scaling) were not included owing to time restrictions, their consideration demonstrates a commitment to inclusive design standards.

In future development, I would like to introduce said features. These aren't just quality-of-life improvements — they can make the difference between someone being able to enjoy the game or not. Learning about accessibility through this project gave me insight into how inclusive design directly impacts real people.

### 5.1.4 Ethical Considerations in Game Design

Games can influence behaviour, attitudes, and attention. While this game does not include sensitive or violent material other than fantasy combat, ethical issues were considered throughout enemy design, feedback systems, and level pacing.

Care was made to ensure that opponents do not display unsettling or explicit behaviour and that the player's activities are portrayed in a stylised, cartoon-fantasy setting. If the project develops into a commercial product, additional efforts will be made to age-rate the content appropriately and conform with relevant rules (for example, PEGI or ESRB).

In dungeon crawlers, there is always a temptation to include "shock value" in the form of dark or violent imagery. I made a deliberate choice to keep the game stylised and fantasy-themed, without grotesque violence or disturbing imagery. This aligns with my personal ethical stance on media and ensures the game remains widely accessible, including to younger audiences.

### **5.1.5 Professional Conduct and Self-Management**

As a solo developer, following good professional practice was critical. The use of Agile techniques, version control, and modular code architecture mimics industrial procedures while encouraging discipline and uniformity throughout development. Features were rigorously tested prior to deployment, and careful time management was used to balance education, part-time work, and iterative game creation.

Furthermore, meeting deadlines, seeking supervisor feedback, and keeping adequate documentation indicated accountability and congruence with the expectations of a professional software engineer.

One of the key lessons I learnt during this project was the importance of accountability and structured progress. In a professional setting, no one chases you — staying on track and holding yourself to deadlines is a core part of responsible software engineering. I often had to balance this project with part-time work and academic deadlines, and managing those competing priorities mirrored real-world working environments more than any previous coursework.

### **5.1.6 Conclusion**

Professional concerns were handled with care and intention throughout the development of this project. From responsible licensing and accessible design to ethical game mechanics and development workflow, every attempt was taken to mimic the standards required in real-world software engineering. While there is always more to learn and improve on, this project has provided a solid foundation in professional behaviour and awareness, which are necessary qualities for any aspiring game producer.

Overall, this project served not just as a practical demonstration of my technical ability but as a platform to think deeply about the ethical, professional, and social dimensions of being a game developer. From responsible use of open-source tools and respecting copyright laws to creating a fair, inclusive, and accessible user experience — every decision I made was guided by an understanding of my role not just as a programmer but as a digital creator with responsibilities to others. These lessons will continue to shape how I approach future projects, whether in games, software, or any field where people interact with the tools I help build.

## Chapter 6: Reflections

### 6.1 Reflection on the Project

Looking back on the entire development process, this project was both the most demanding and satisfying piece of work I've done during my degree. The initial goal, which was to create a randomly generated 3D dungeon crawler with real-time combat and AI, seemed ambitious at the time, especially given my limited knowledge with Godot and minimal experience developing a whole game from scratch. Despite this, I tackled the project with a strong feeling of commitment and optimism, which in turn, paid off as each milestone was gradually reached. While certain concepts were improved or cut down along the process (for example, adding an inventory system or complex trap logic), the essential gameplay loop, aesthetic, and design principles remained intact. The iterative aspect of developing games became obvious soon; things rarely worked properly the first time, and I spent a lot of time testing, improving, and altering mechanisms depending on how they felt while playing. These factors made the process more dynamic and imaginative than earlier academic projects.

### 6.2 Personal Self-Assessment

It was clear from the outset that this project would need an elevated level of independence, discipline, and technical exploration. Having never used Godot previously, I knew the learning curve would be severe, especially given the node-based architecture and unique scripting language, GDScript. Despite this, I set aside time early on to completely study the engine and more time over the winter term break to fully understand the logic and mechanisms behind the functionalities I needed to use, using documentation, tutorials, and open-source examples. This paid off, as I was able to confidently start creating prototypes before the conclusion of the first term and after the winter break.

Adaptability is one of the areas where I felt I excelled. I had various obstacles, ranging from player collisions and UI state tracking to logic faults in room generation, but I stayed proactive in finding answers. I relied significantly on modular code design and frequent testing to decrease errors and enhance maintainability. Another quality was my ability to manage my time. Balancing the project with other coursework and part-time jobs was tough at times, but I constantly set aside dedicated development hours and tracked progress using Trello.

However, there were areas in which I could improve. There were times, particularly during development, when I grew overly focused on visual refinement and failed to emphasise technical depth or performance optimisation. I also often misjudged the time necessary to create specific dynamics, such as enemy AI coordination or inter-room connectivity. Because of these errors, things such as puzzles and traps were omitted because of time restrictions.

That being said, I am proud of the work I created and the systems I established. This project tested my abilities as a programmer, designer, and problem solver. It gave me confidence in working with larger codebases and underlined the significance of clean architecture, testing, and user experience — all of which I will apply to future projects.

### 6.3 Analysis of Code and Challenges Encountered

Several technical and design challenges arose throughout development, putting my problem-solving skills and understanding of game architecture to the test. Because this was my first



significant project in Godot, many of these difficulties resulted from my unfamiliarity with the engine's distinct structure, specifically its node-based scene system and how it handles instancing, signals, and scene transitions.

One of the first and most persistent issues was with the procedural dungeon generation system. While the final technique used a modular grid-based layout, getting rooms to spawn without overlapping, align correctly, and link with doors required a significant amount of trial and error. Initially, doors were misaligned because of irregular offsets and scene points. To address this, I built position snapping and a mechanism that dynamically detects available directions prior to room instancing — however, this took a significant amount of time and debugging incorrect transformations, room rotations, and node parenting.

The opponent AI added another level of difficulty. I had to create distinct logic for melee and ranged enemies while keeping consistent behaviour states like idle, chase, attack, and death. Overlapping opponent states were a recurrent issue; for example, if the player exited their range, certain enemies might begin attacking mid-animation or remain in a locked loop. To solve this, I implemented state Enums and tighter Boolean markers such as `is_awake`, `is_casting`, and `can_attack`, which helped to clearly outline AI transitions and prevent unexpected behaviour.

Managing animations also proved more difficult than expected. Tying combat animations to actual damage-dealing moment required synchronisation via await calls, timers, and precise animation frame sequencing. Initially, when the player was to attack, the attacking animation would be cut off and not play, due to there being the constant Idle animation, which disrupted immersion. Through iteration, I created a reliable structure that controlled the priority of animations.

Finally, while not a problem, managing scene transitions (particularly in menus and overlays) demonstrated how delicate Godot's scene tree can be. If transitions were not correctly timed, such as switching a scene before its animation concluded, it might result in crashes or unresponsive input. To overcome this, I utilised `AnimationPlayer`'s `animation_finished` signal and await to sequence UI updates, resulting in a significantly better user experience.

Overall, as the codebase grew larger and more complicated, each difficulty drove me to create cleaner, more modular, and smarter solutions. These challenges were tough at the time, but they ultimately improved the reliability and maintainability of the game's key components.

## 6.4 Milestones Achieved

Despite the technical scale and hurdles, the project successfully met the bulk of its key objectives that were set. The most significant achievement was the creation of a working procedural dungeon-generating system. The game creates a non-linear structure by constantly spawning modular rooms and connecting them via doors, which promotes exploration and unpredictability – a fundamental aspect of the roguelike genre.

Combat was another big accomplishment. A full melee system was built, which included timed attacks, animation synchronisation, and range-based hit detection. This was complemented by a variety of enemies, including a melee skeleton and a ranged mage enemy, both of which used autonomous state-based AI. Each adversary responds intelligently to the player's position, alternates between patrol and attack modes, and has responsive animations and sound effects.

Interactive components were also an important milestone. The healing potions and score-granting chests were implemented by proximity detection and state-controlled interactions. These pickups not only improve gameplay, but they also flow directly into the player feedback loop through the HUD system. The HUD itself contains dynamic health updates, a real-time score tracker, and persistent high score saving via Godot's file system.

A full main menu, pause functionality, options screen, credits, and a ‘how to play’ section were also completed — all with animated transitions, button sound effects, and scene management. These add to the overall quality and usefulness of the game.

Other key milestones include:

- A comprehensive audio system with context-based sound effects (e.g., footsteps, spellcasting, chest openings).
- A responsive player movement system, including sprinting and animation mixing.
- Version control throughout, allowing for effective rollbacks, modular commits, and accurate project tracking.

These accomplishments together demonstrate a fully operational game that largely adheres to the initial project objectives, despite the deferring of certain extended goals—such as traps, puzzle elements, and inventory systems—due to time limitations.

## 6.5 Future Plans

Although the project is completely playable and has met its major objectives, there are a few areas I would like to build on in the future to improve the game's depth, polish, and replayability. One of the most immediate improvements would be the implementation of an inventory system. While potions and chests now provide one-time interactions or quick effects, adding a comprehensive inventory with item categories (e.g., consumables, equipment) would allow for strategic decision-making and player customisation. This would also allow for elements like weapon equipping, loot storage, and the use of keys to access locked places, further pushing the players ability to explore the dungeon.

Another area of focus would be to include puzzles and traps. These were initially proposed in the plan but were not added owing to time constraints. Adding logic-based puzzles, risky environments, and hidden treasures would break up combat sequences and provide additional diversity to the dungeon adventure. These features might be integrated procedurally or assigned to specific room types to provide additional difficulty and reward.

AI behaviour may also be extended. While current opponents use simple state machines and line-of-sight logic, future upgrades may feature more complicated behaviours like group cooperation, ranged evading, or scripted mini-bosses with phase-based battles. This would improve the gameplay loop by providing more dynamic encounters.

Visually, greater variation in level themes, lighting, and dungeon biomes would make each run feel unique. On the technical side, optimising procedural generation to accommodate multi-floor dungeons or themed zones will increase the game's replayability even further.

Future thoughts include upgrades to accessibility, such as complete gamepad compatibility, colourblind features, configurable difficulty settings and many more accessibility features. These characteristics would increase the possible audience and indicate a greater degree of design inclusiveness.

Finally, I'd release the game in some form, whether as a free indie release or as the foundation for a larger project. The positive feedback loop of creating, testing, and improving has been really fulfilling, and this project has reaffirmed my passion for game production. I now feel prepared and inspired to continue working on games after graduation, whether individually or professionally.

## Chapter 7: Bibliography

- [1] Adams, E., & Rollings, A. (2010). *Fundamentals of game design* (2nd ed.). New Riders.
- [2] Artofttransformation. (2008). *Rogue screenshot CAR [Creative artistic rendering]*. Reprinted from Wikipedia. [https://en.wikipedia.org/wiki/File:Rogue\\_Screen\\_Shot\\_CAR.PNG](https://en.wikipedia.org/wiki/File:Rogue_Screen_Shot_CAR.PNG)
- [3] Bidarra, R., & Smelik, R. M. (2011). Guest editorial: Procedural content generation in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3), 169–171. <https://doi.org/10.1109/TCIAIG.2011.2164060>
- [4] Blizzard Entertainment. (1996). *Diablo* (Version 1.0) [Video game]. PC. Blizzard Entertainment.
- [5] Blizzard Entertainment. (2000). *Diablo II* (Version 1.0) [Video game]. PC. Blizzard Entertainment.
- [6] Brazie, A. (2023, October 6). Designing the core gameplay loop: A beginner's guide. *Game Design Skills*. <https://gamedesignskills.com/game-design/core-loops-in-gameplay/>
- [7] Cartlidge, J. (2024, September). Genre, prototype theory and the Berlin interpretation of roguelikes. *Game Studies*, 24(3). <https://gamestudies.org/2403/articles/cartlidge>
- [8] Coles, J. (2020, December 31). 2020 in review – The year of the roguelike. *TheSixthAxis*. <https://www.thesixthaxis.com/2020/12/31/2020-in-review-the-year-of-the-roguelike/>
- [9] Dahlskog, S., Björk, S., & Togelius, J. (2015). Patterns, dungeons and generators. In *Proceedings of the Foundations of Digital Games Conference (FDG)* (pp. 1–8). Pacific Grove, USA.
- [10] Doan, D. (2016, December 6). GameDev protips: How to design a truly compelling roguelike game. *Medium*. <https://medium.com/@doandaniel/gamedev-protips-how-to-design-a-truly-compelling-roguelike-game-d4e7e00dee4>
- [11] Farrokhi Maleki, M., & Zhao, R. (2024). Procedural content generation in games: A survey with insights on emerging LLM integration. *arXiv preprint arXiv:2410.15644*. <https://arxiv.org/abs/2410.15644>
- [12] Garda, M. B. (2013). Neo-rogue and the essence of roguelikeness. *Homo Ludens*, 1(5), 59–72.
- [13] Gellel, A., & Sweetser, P. (2020). A hybrid approach to procedural generation of roguelike video game levels. In *Proceedings of the International Conference on the Foundations of Digital Games (FDG '20)* (pp. 1–18). ACM. <https://doi.org/10.1145/3402942.3402950>
- [14] Game Design Strategies: Creating engaging and immersive experiences. (2024, August 22). *Main Leaf Games*. <https://mainleaf.com/game-design-strategies/>
- [15] Game-Wisdom. (2025). Making roguelikes more accessible. <https://game-wisdom.com/critical/rogue-like-accessible>
- [16] Godot Engine. (2024). *Godot engine documentation*. <https://docs.godotengine.org>
- [17] Harris, J. (2011, February 2). Analysis: The eight rules of roguelike design. *Game Developer*. <https://www.gamedeveloper.com/game-platforms/analysis-the-eight-rules-of-roguelike-design>

- [18] Harris, J. (2020). *Exploring roguelike games*. CRC Press.
- [19] Hendrikx, M., Meijer, S., Van der Velden, J., & Iosup, A. (2013). Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 9(1), 1–22. <https://doi.org/10.1145/2422956.2422957>
- [20] Hunicke, R., LeBlanc, M., & Zubek, R. (2004). MDA: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*. <https://users.cs.northwestern.edu/~hunicke/MDA.pdf>
- [21] IEEE-USA InSight. (n.d.). Going rogue: A brief history of the computerized dungeon crawl. *IEEE-USA InSight*. <https://insight.ieeeusa.org/articles/going-rogue-a-brief-history-of-the-computerized-dungeon-crawl/>
- [22] Izgi, E. (n.d.). *Framework for roguelike video games development* [Master’s thesis, Charles University]. Charles University Digital Repository. <https://dspace.cuni.cz/bitstream/handle/20.500.11956/94724/120289816.pdf>
- [23] Khalifa, A., Bontrager, P., Earle, S., & Togelius, J. (2020). PCGRL: Procedural content generation via reinforcement learning. *arXiv preprint arXiv:2001.09212*. <https://arxiv.org/abs/2001.09212>
- [24] Khalifa, A., Gallotta, R., Barthet, M., Liapis, A., Togelius, J., & Yannakakis, G. N. (2025). The procedural content generation benchmark: An open-source testbed for generative challenges in games. *arXiv preprint arXiv:2503.21474*. <https://arxiv.org/abs/2503.21474>
- [25] McMillen, E. (2011). *The Binding of Isaac* (Version 1.666) [Video game]. PC. Edmund McMillen.
- [26] Millington, I. (2019). *AI for games* (3rd ed.). CRC Press.
- [27] Nystrom, R. (2014). *Game programming patterns*. Genever Benning.
- [28] Patterson, B., & Ward, M. (2022). Adaptive procedural generation in Minecraft. *Game Developer*. <https://www.gamedeveloper.com/design/adaptive-procedural-generation-in-minecraft>
- [29] Schell, J. (2019). *The art of game design: A book of lenses* (3rd ed.). CRC Press.
- [30] Shaker, N., Togelius, J., & Nelson, M. J. (2016). *Procedural content generation in games: A textbook and an overview of current research*. Springer International Publishing.
- [31] Sicart, M. (2008). Defining game mechanics. *Game Studies*, 8(2). <http://gamestudies.org/0802/articles/sicart>
- [32] Smith, G. (2014, April). Understanding procedural content generation: A design-centric analysis of the role of PCG in games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1–10). ACM.
- [33] Valtchanov, V., & Brown, J. A. (2012, June). Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International Conference on Computer Science and Software Engineering* (pp. 1–8).
- [34] Volz, V., Naujoks, B., Kerschke, P., & Tušar, T. (2023). Tools for landscape analysis of optimisation problems in procedural content generation for games. *arXiv preprint arXiv:2302.08479*. <https://arxiv.org/abs/2302.08479>

[35] Wikipedia contributors. (2024, November 25). *Rogue (video game)*. Wikipedia.  
[https://en.wikipedia.org/wiki/Rogue\\_\(video\\_game\)](https://en.wikipedia.org/wiki/Rogue_(video_game))

## Chapter 8: **Appendix**

### 8.1 User Manual

#### 8.1.1 Running Instructions

##### **Prerequisites:**

Before running the game, make sure to have the Godot 4 engine installed: download from: [here](#)

##### **Setting up the project:**

###### **USING GIT:**

1. Open terminal/command prompt.
2. Clone the repository.
3. Navigate to the project folder.

###### **WITHOUT GIT:**

1. Download the project as a .zip file from the repository.
2. Extract the contents to a folder on your system.

##### **Running the game in Godot:**

1. Open the Godot engine.
2. Click import project and find the folder containing the relevant files.
3. Select the file and open
4. Once opened, press F5 to run the game

##### **OR:**

Load the game through the .exe application

## 8.2 Gantt Chart – Project Timeline

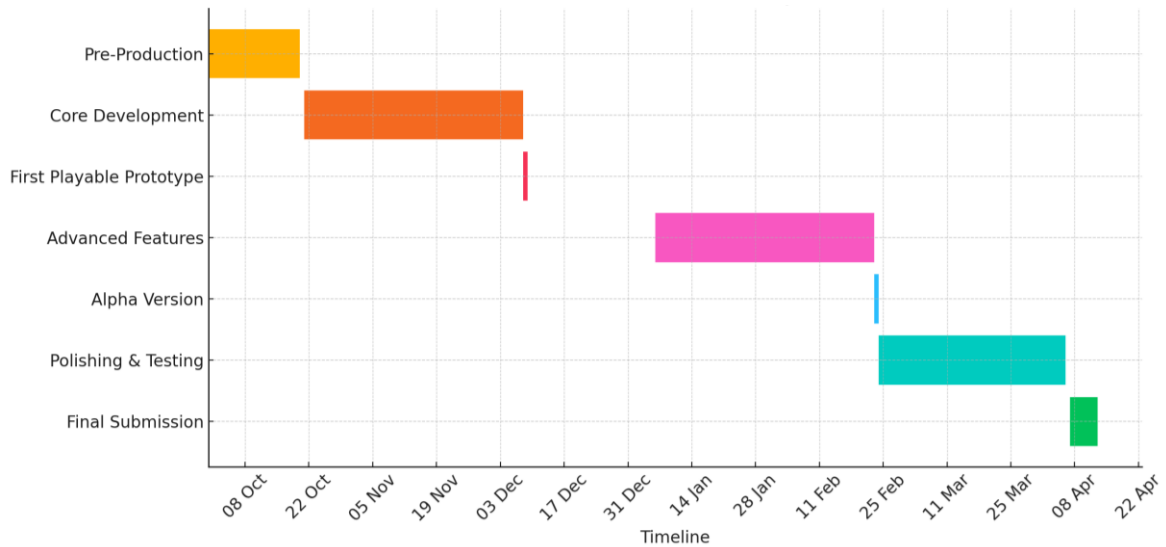


Figure 25. Gantt chart outlining the projects development phases

## 8.3 User Play-Testing Form

<https://docs.google.com/forms/d/e/1FAIpQLSfcYFHFGb9z3JQBJKcNejxOazvYlyeCAVIWFjGL56cvqYGwrw/viewform?usp=header>

## 8.4 Project Diary

30/09/2024

The initial ideas for this game are that it is to be built in a game engine such as Unreal or Godot. Research into both of these have been done but more thorough work is to be done.

Ideas for the game are a FPS, drawing inspiration from titles such as Doom or Half-life. Other ideas do include top-down dungeon crawlers with puzzle-based elements.

02/10/2024

With much consideration to the ideas and functionality of both engines I have decided on using Godot. This engine is more suitable for me and this project and aligns with the skills I possess to excel with some of the concepts. The documentation for this engine has been read and tutorials provided have been attempted.

Further with this the project plan will be started, finding relevant books and papers to aid in my work.

04/10/2024

On the 3rd of October I had met with my supervisor and discussed the project ideas that I had come up with and talked about any relevant help with the project plan. Following this a draft plan has been started.

07/10/2024

The plan timeline has been drafted up and the abstract has been partially drafted too. Research into relevant papers has also been conducted with papers such as 'Designing procedurally generated levels' - Linden, R., Lopes, R., & Bidarra, R. (2021). Designing Procedurally Generated Levels. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 9(3), 41-47. <https://doi.org/10.1609/aiide.v9i3.12592>. Risks and mitigations for those risks have been drafted into the daybook.

08/10/2024

The risks and mitigations section of the project plan has been completed. Once other parts of the plan are complete a draft will be sent to my supervisor for any final comments and tweaks and will be changed. With this, the timeline will now be started, building upon the initial draft I did for it. Abstract is still to be finished as the bibliography for it is still limited.

09/10/2024

The timeline for the project has been completed, with specified milestone and mapped to the term weeks. The abstract still needs work and further research is being conducted. Once a rough abstract is complete, I will send it to my supervisor for comments, and with these, on the 10th the plan will be completed, ready for submission on the 11th.

I have sent what I have done of the plan so far to my supervisor for further guidance, once feedback is received from them, I will continue in the correct direction.

10/10/2024

I have received positive feedback about my plan from my supervisor with further tips to finish off the abstract and bibliography. Work on this is underway and should be completed by tonight. Further research on papers have also been done.

The project plan is pretty much finished. The timeline, risks and bibliography is done, the abstract is almost finished, it needs a touch up and a bit more writing and the plan will be ready for the submission. Once done, the IDE will be setup and coding will begin.

11/10/2024

The project plan has been completed and uploaded to Moodle. Now that it is complete, the coding for the game will begin.

The file has also been uploaded to the Gitlab documents directory.



The interim report will be drafted once the coding starts.

13/10/2024

The Godot IDE is setup, linked with the GitLab, a master branch has been created for the initial work to begin, this is the first objective met that was proposed in the timeline. I will start, as stated in the project plan, with the game's menu screens. I have created the buttons with a control node and have attached a script to allow for functionality of these menu buttons, this is not finished yet.

14/10/2024

I have created placeholder menu screens such as the main menu, game screen, and the options menu. I have partially scripted the menu buttons to be connected to each screen when they are pressed, this is another objective completed from the plan from week 4-5.

I am facing an error while getting the buttons to correspond to their appropriate screen, the `change_Screen` functions.

I have realised the documentation I was using was for Godot v3 and not v4 which is the version I am using. In v3 they used just `change_scene`, whereas in v4 they changed it to `change_scene_to_file(...)`, which now works. I will now refer to using the v4 documentation, opposed to the v3 which I wrongly opened. The buttons for the main menu screen now have functionality and take me to the correct screen. I will now begin on creating a back button for the menus and visually sort them out.

I have centred the main menu buttons and gave them labels

I have added a back button to options menu, created a script in the options menu to add functionality to get the player back to the main menu.

15/10/2024

Today I looked at some potential free assets that I can use for the UI, however still undecided on which to go for. I have created a 3D player with a mesh and collision nodes, I will now add functionality to these. I also attempt to do the player movement and try to attempt the above style camera.

I have created a basic rudimentary movement script for the player and mapped the physical buttons on the keyboard to work with the script. I will work on the camera and test to see if it works.

I have created a test environment, basic plane with collision, the movement for the player works along with collision, the camera however is not working as expected. Therefore I need to work on fixing that.

I have sorted the camera angle out but now the controls seem to be inverted, I will need to look into this and fix it.

I have had to do a temporary inversion of the controls which has seem to do the job, I have sorted out the camera angle as well. I have replaced the placeholder game menu with the current screen with the player and environment.

21/10/2024

After a few days off to work on other university work and commitments, it is the start of week, and I will start the next stages of the work required. This week I am to implement the player HUD, implement the enemies and a basic combat system.

22/10/2024

I have created a 2D scene for the players HUD, with a placeholder health bar. The rest of the HUD will be added soon. I have also linked and overlayed it on the main game screen, having tested it, it is functional. I cannot test this health bar till I create some enemies, so my next goal is to create a simple enemy factory.

23/10/2024

I have worked on the enemy 3D model and given it a script, which lets it move randomly and have speed and health. I then created a spawn factory in the main game state which will spawn the enemy randomly on the map. This is all tested and works as needs be. One error encountered was the enemy kept spawning into the map then falling below, this was a simple fix by changing its y coords higher.

I had some time to work on the assets, I added textures to the buttons and gave them a slight animation of hovered and pressed. I also added some basic lighting to the main game state along with some colour differentiations.

I had a look at some further assets such as animations for the player. I'm still yet to decide on one but I may get a free animated one and just use it as a placeholder and use it to understand how to implement the animations in the project.

24/10/2024

I implemented a very basic combat system where the player presses the space bar near an enemy and they take damage, once zero they are killed and taken off the screen. Here many of the early objectives and deliverables I had in the plan have been ticked off, requiring further refinement with time. I will continue to improve and work on this combat system, but this very basic system will suffice for now.

28/10/2024

To start off the new week I spent some time reflecting on what had been done and what needs doing, consulting the timeline I created in the project plan. I have realised that functionality wise I am ahead on what I had predicted which is a good sign, which means I can spend some time on finding and implementing assets and animations into the game.

I have visited websites such as Kenney and itch.io for free assets to use.

28/10/2024

I have started to work on the enemy attack script.

12/11/2024

Coming back to the project I have finished the enemy attack's function and their pathfinding. Fixed errors that arise such as the players attack mechanism affecting the players health instead of the enemies. Also removed redundant comments left for me when starting off as they no longer serve any purpose.

The next steps to follow is to work on the procedural generation of the dungeon itself and finalise and apply the assets.

18/11/2024

I have been spending my time researching procedural map generation these past few days and watching YouTube videos on it. I'm gaining a better understanding of how to implement it and the usage of it.

28/11/2024

I have been making separate projects to test the procedural generation. so far, it's very simple and not exactly how I want it and figuring out how to implement it into my code and project is difficult.

04/12/2024

I have found some assets and animations to use as placeholder for now for my game. I have imported them and tested to see if they are compatible, which they are.

I have added simple animation and the model to the player, which are functional and work, only thing that isn't is the attack animation. Once this is done I will continue work on my separate file on procedural map generation.

11/12/2024

Today I had my project demonstration for the interim submission.

12/12/2024

The interim report is all written and the creation of the video demo will be done as well. All ready for the submission date on the 13th.

16/2/2025

Milestone one was achieved as stated in the project plan, but a lot more work is to be done, many of the basic features stated however are functional.

During the Christmas break I conducted in house play testing to gather initial feedback on my game, this helped identify key areas for improvement and things that are working well so far, this is going to guide me for development the next few weeks.

17/2/2025

Today the main goal was to get the procedural generation sorted. I ensured that rooms spawn dynamically without overlapping, and linked doors correctly between them, so rooms can be traversed, this was a big objective to get sorted that was stated in the development plan, ticking this off is a huge goal. I also implemented a player camera system with zoom functionality to improve visibility in the isometric view.

I also rebuilt the main menu, restructuring the button layout using VBoxContainer for better organization. To improve UI polish, I added a fade-in animation for the menu using Animation Player and the Modulate property, making transitions feel smoother, along with a background image for the main menu.

18/2/2025

Today I plan on revamping the enemy spawning and AI to work with the procedurally generated dungeon.

I worked on this and have got it partially working however it is highly bugged right now enemies are spawning outside the allocated dungeon rooms.

20/2/2025

Today I plan to fix the enemy spawn issue, then work on actually finding assets for the game, build different rooms and get a proper UI implemented.

I have added a loading screen to my game with my logo at the start.

11/3/2025

I have got assets for my game and have created dungeon rooms with the assets, I now need to implement that room with the procedural generation code that I have, after that I will work on the UI before proceeding with further features.

12/3/2025

The procedural generation with the new assets dungeon now works. time to fix and improve the player asset and movement and animations.

I have improved the player movement by having them face the correct way they are moving. I have also implemented a player sprint by pressing the shift button. I tried to get the attack animation to work when space is pressed but this is bugged, and have pushed it to the side to work on it when I get to working on the attacking and enemies. Next I need to improve the camera as it is a bit jittery when moving and set a zoom limit. then work on the main menus along with the options menu.

I have been having serious problems with the camera so have reverted back to standard and will now focus on UI.

I have created a new menu background and introduction animation, I will now add assets to my buttons.

I have added the button assets to my main menu with some decent effects. I will next work on sorting out the wall collision and make sure the doors in the dungeon are functional.

13/3/2025

Today was a mix of progress and frustration while working on the dungeon generation. I focused on refining how rooms spawn, making sure they align properly and connect through doors without gaps. I spent a lot of time trying to get trap rooms to generate correctly, ensuring they only have one entrance, but they kept getting rejected due to a lack of valid placements so I reverted back to them having 4 rooms. At one point, the dungeon was only spawning rooms in a straight line, which I had to fix by shuffling the placement directions. Another issue popped up where only one room would spawn, forcing me to roll back to a previous working version. Despite the setbacks, I managed to get the dungeon generator functioning again, though there are still some alignment issues with doors and occasional room clipping. Tomorrow, I'll refine the trap room placement, work on adding more room variety like puzzle and loot rooms, and start planning the loot and inventory system.

19/3/2025

Today I was working on getting the dungeon generation to finally work as intended. I discovered that the doors weren't being recognized properly because the Area3D nodes were children of the door assets instead of being directly referenced. Fixing that allowed rooms to start linking correctly. I also experimented with different approaches to trap rooms, trying to restrict them to one entrance, limit their count, and ensure they linked properly, but these changes kept breaking the system, either causing clipping, floating rooms, or failing to spawn at all.

Eventually, I decided to match the trap room size to the standard dungeon room and allow them to spawn like regular rooms, which finally worked. So now I will move onto getting the enemy and HUD with inventory sorted.

22/3/2025

So far I have found good assets to use for my enemies and have implemented a basic spawning routine for them in the dungeon, along with fixing their AI, next I need to polish the player HUD and add it so I can attack enemies and enemies can attack the player.

I polished the enemy AI and animation system. I fixed the rotation so they now properly face the player, and got all the animations working in sync — including the floor idle pose, the wake-up animation, walking, attacking, dying, and the new idle combat pose when the player leaves their vision range. I also refined their behaviour so they stop chasing when out of range but re-engage properly when the player returns, and prevented them from constantly walking into the player during combat.

I implemented proper melee combat for the player, starting with hit detection and damage dealing using a proximity-based check. I ran into some issues with the attack animation not playing correctly, but after some debugging, I realized the idle animation was cutting it off. I fixed it by waiting for the attack animation to finish before returning to idle, and now it all works smoothly.

24/3/2025

I focused on improving the player's HUD and combat feedback. I rewired the HUD to properly reflect the player's health using heart icons, with each heart representing 20 HP (totalling 5 hearts for 100 HP). I also integrated a score system that updates every time an enemy is killed. Then, I added an animation to the score label — had some hiccups with containers restricting scale edits, but I figured out a workaround. Now the plan is to work on getting all the menus sorted and polished, along with the inventory and music.

I implemented a fully functional pause menu in my game. Pressing ESC now pauses gameplay and overlays the menu on top of the scene. The menu includes buttons for resuming, accessing options, and returning to the main menu. Right now, pressing ESC again in the pause menu crashes the game, so I need to fix that.

I fixed the crash caused by pressing ESC while the pause menu was open, the issue was due to a null reference when trying to call `toggle_pause()`. I solved it by safely checking if the `PauseMenu` node exists before calling the method. I also moved the input handling to `_unhandled_input()` instead of `_input()` to make sure it only triggers once and doesn't interfere with other input events.

27/3/2025

I focused on rewiring the death screen to fit the current system, it works fine and intended. I am now working on a new feature which is where chests and potions spawn dynamically in the dungeon to aid exploration and to help the player health and get more score.

I have managed to implement some kind of working chest system, they spawn randomly and in different rooms but there is a bug where they are spawning outside the rooms. But they work as intended, when the player interacts it adds to their score. Now it's time to work on the health potion.

28/3/2025

I worked on getting a proper score system in place. I updated the HUD to display both the current score and the high score, and after a bit of trial and error, I got it to save the high score to a file on death. I ran into a weird issue where the high score kept resetting when I pressed retry, but I eventually figured it out by saving the score before reloading the scene. I tested thoroughly by setting a high score, beating it to see if it updated, and got a lower one to see if it wasn't replaced, then tested by exiting to main menu and playing again to see if it was still there. Next will be to polish a few things then start working on adding music and SFX along with the options menu.

I added a quick polish where it tells the player using a pop up that they have a new high score.

Milestone 2 was achieved later than it was stated in the plan but non the less the game is looking functional with a decent amount of polish.

31/3/2025

I am working on adding audio and SFX to the game.

Today I fully integrated sound into the game, adding audio for both gameplay and UI. The player now has footstep and attack sounds, potions and chests play pickup effects, skeleton enemies have unique wake-up, walking, and death SFX.

The options menu for the game is now fully functional, it can be accessed in the main menu and the pause menu of the play state. The settings contain audio and video settings. Now I will work on polishing some things and some refactoring of the code.

01/4/2025

I added some simple fade animations to the menus, I have also added shadows to the player and enemy to make them look more natural. I will now work on further polishes. I have implemented a ambience of fog and light flicker, along with damage feedback in the camera shaking to show the player they've taken damage.

Despite minimum time being left, after the play test, players did ask for another enemy, so I added a mage who does ranged attacks.

I have now added a how to play menu, along with my wall collisions now functional

02/4/2025

I added a void death to the player to prevent out of bounds traversal. I have also added a credits scene for all assets used. I also removed some files that were unused, such as assets.

Refactoring for the code has begun. The camera, chest, control, potion and credits have been done so far.

03/4/2025

I've refactored mageSpell.gd, loading screen, HUD, HowToPlay, game over, flicker, pauseMenu.

It's now time for the big refactor and comments for the important sections of code, smaller scripts have been completed.

Options and Settings scripts done. Along with player, and mage enemy. Full refactor and comments of code now complete.

05/4/2025

A final play test of the game will be conducted, and feedback will be taken as usual. With this, looking back at the project plan and the timeline that was stated, everything had worked out, with some delays here and there, we ultimately ended up in the desired path, all objectives set in the last weeks have been achieved. Now it's time to tidy up the codebase and what else and then work on finalising the final report.

All project files have been organised, docs needed to be added to explain.

Documentation for each folder had been added.

07/4/2025

I have added the final feature for the game which is a victory screen for when the player kills all enemies in the dungeon.

Working code branch now successfully merged with the main branch. This is now the final stage in the timeline and development for the game has now fully concluded. Time will now be focused on finalising the report and preparing for the final submission. What a journey this has been, this project has helped me significantly grow as a developer.

08/4/2025

Finalisation of the report, demo video and game.exe will be ready for submission on the 11th.

Prithvi S. Veeran - Signing out

## 8.5 Link to Demo Video

<https://youtu.be/6oARWrsu7kI>