Final Year Project Report

Full Unit – Final Report

Procedural Generation Dungeon Crawling Game

Salma Bocus

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 24, 2025

Declaration

This **final 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 Where other published and unpublished source materials have been used, these have been acknowledged.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: 12 000 (excluding references & appendicies)

Student Name: Salma Bocus

Date of Submission: 24th April 2025

Signature: 

Table of Contents

Chapter 1: Introduction

1.1 Abstract

The gaming industry stands as one of the most influential and rapidly growing sectors in technology, captivating audiences worldwide with its innovative storytelling, immersive mechanics, and cultural significance. With an estimated 3 billion players globally, the gaming sector caters to diverse demographics, from hardcore enthusiasts to casual players seeking entertainment or educational content [1]. Games have evolved into a powerful medium for artistic expression and social commentary, transcending their traditional roles as mere recreational activities. This growth highlights the importance of developing inclusive and engaging experiences that resonate with a wide range of players.

This project reflects both my passion for gaming and my drive to create meaningful interactive experiences. From a young age, I explored the world of game creation, experimenting with simple mechanics and narratives. Those early experiences inspired my journey into game development, culminating in this final year project. Through this work, I aim to contribute a unique perspective to the gaming landscape, focusing on accessibility, humour, and story-driven design. Additionally, this project aligns with key industry trends and practices, including procedural generation, modular design patterns, and user-centred design principles [2].

The lack of games specifically tailored for female audiences remains a notable gap in the gaming industry. While many games achieve broad appeal, few cater explicitly to women, particularly within genres like rogue-like dungeon crawlers. Rogue-likes are traditionally characterised by their procedurally generated environments, permadeath systems, and high replayability. However, their complex mechanics and darker, combat-focused aesthetics often alienate potential players who might seek a more accessible and relatable experience.

My game, titled 'Just a Girl's Dungeon Quest' addresses this gap by reimagining the rogue-like genre through a lens of humour, relatability, and empowerment. By blending classic rogue-like mechanics with a lighter narrative tone and a feminine aesthetic, this game offers a fresh take on the genre. The protagonist's journey of self-discovery is central to the narrative, providing a thematic depth often overlooked in traditional rogue-likes. The game's design emphasises inclusivity and accessibility, challenging conventional notions of what a rogue-like can be.

What sets my game apart is its balance of complexity and approachability. While maintaining the strategic depth expected of rogue-likes, it incorporates a story-driven experience that appeals to players seeking humour and emotional engagement. This hybrid approach fills a void in the market, offering a rogue-like dungeon crawler that resonates with audiences often overlooked by mainstream gaming. By leveraging procedural algorithms and efficient state management systems, the game integrates replayability with narrative cohesion.

1.2 Aims and Goals

The overarching aim of this project is to develop a polished and engaging rogue-like dungeon crawler tailored for younger female audiences. By the project's conclusion, I intend to deliver a fully functional game that showcases procedural generation, character progression, and dynamic interactions within a visually cohesive and narratively rich environment.

The game's unique selling point (USP) lies in its combination of procedural dungeon exploration, relatable storytelling, and a vibrant, feminine aesthetic. The project seeks to create a game that not only entertains but also inspires players through themes of self-discovery, courage, and humour. My goal is to provide casual players with an accessible entry point into rogue-like games while delivering enough depth and challenge to satisfy seasoned gamers.

From a technical perspective, the project aims to:

1. Implement a robust procedural dungeon generation system with increasing complexity across levels
2. Create a turn-based combat system with strategic depth and varied enemy behaviours
3. Develop an intuitive grid-based movement system that feels responsive and natural
4. Integrate a narrative framework that provides context and motivation for dungeon exploration
5. Design save/load functionality to allow players to track their progress
6. Balance difficulty progression to ensure challenge without frustration

From a personal development standpoint, this project provides an opportunity to:

1. Expand my proficiency with the Godot 4 engine and GDScript
2. Improve my understanding of game design patterns and their implementation
3. Enhance my skills in software engineering best practices
4. Develop my ability to write efficient and maintainable code for complex systems

This combination of technical and personal development goals ensures that the project serves as both a demonstration of my current abilities and a vehicle for growth as a game developer.

1.3 Project Evolution

This section describes how the project evolved from initial concept to final implementation, highlighting key decisions, pivots, and expansions throughout the development process.

1.3.1 Initial Vision

The project began with a relatively modest vision: to create a basic procedurally generated dungeon with simple movement mechanics. Early prototyping focused on implementing a random walk algorithm to generate playable levels and establishing grid-based movement for player navigation.

1.3.2 Expanding Scope

As development progressed, the potential for a more comprehensive experience became apparent. The scope expanded to include:

1. **Narrative Framework**: The addition of dialogue scenes (scene_1.gd and scene_2.gd) created a story context for the dungeon exploration, giving players motivation and character investment.
2. **Town and Castle Hubs**: Implementation of non-dungeon areas (town.gd and castle.gd) provided players with safe zones for dialogue and game state management.
3. **Combat System**: The battle system evolved from a conceptual feature to a fully-realised turn-based combat system with varied enemy behaviours and player abilities.
4. **Save/Load System**: To enhance the persistence of player experience, a comprehensive save/load system was implemented.

1.3.3 Technical Challenges and Solutions

Throughout the project's evolution, several technical challenges arose that required creative solutions:

- **Balancing Procedural Generation**: Finding the right balance between randomness and playability in dungeon generation required multiple iterations of the algorithm.

- **Enemy AI Behaviour**: Creating engaging enemy behaviours that weren't predictable or exploitable required careful implementation of pathfinding and decision-making systems.

- **State Management**: As the game grew more complex with multiple scenes and systems, effective state management became crucial for maintaining game integrity across transitions.

During development, several specific technical challenges emerged that required creative solutions. One significant challenge was balancing the procedural generation algorithm to create interesting yet navigable dungeon layouts. Initial iterations produced either overly linear paths or disconnected regions that trapped the player. Through experimentation with the random walk parameters and implementing additional validation checks, I eventually achieved a balance that created organically flowing dungeons with appropriate complexity. Another challenge was implementing the enemy AI pathfinding system, which initially caused enemies to get stuck in loops or make illogical movement choices. By developing a weighted decision-making system that prioritizes paths leading toward the player while avoiding obstacles, I created more intelligent-seeming enemy behavior. Perhaps the most complex challenge was integrating the turn-based combat system with the exploration mechanics, requiring a careful state management approach to transition smoothly between these distinct gameplay modes without conflicts or resource issues.

1.3.4 Final Scope

The final project significantly exceeds the initial concept, delivering a complete game experience with:

- Multiple interconnected scenes (title, character creation, town, castle, dungeon)
- Procedurally generated dungeons with scaling difficulty
- Turn-based combat with strategic depth
- Character progression through experience and abilities
- Save/load functionality
- Narrative elements that contextualize gameplay

This evolution demonstrates both my adaptability as a developer and my commitment to creating a cohesive, engaging experience rather than merely fulfilling the minimum technical requirements.

1.4 Challenges

1.4.1 Learning Curve of Godot

As Godot was a new engine for me, mastering its scene structure, GDScript syntax, and procedural mechanics posed initial challenges. To address this, I allocated time for experimentation and focused on smaller tasks to build familiarity before tackling core mechanics. Extensive use of documentation and tutorials ensured a steady learning progression. Particularly challenging was understanding Godot 4's node-based architecture and signal system, which required a different mindset compared to more traditional object-oriented frameworks I had previously used.

1.4.2 Procedural Generation Complexity

Generating diverse and balanced dungeon layouts required precise algorithms and performance optimisation. To mitigate potential inefficiencies, I began with simpler random walk algorithms and introduced constraints incrementally, testing performance at each stage. Initially, my generation algorithm created disjointed areas and occasionally impassable corridors. Through gradual refinement, including implementing validation checks and adjusting parameters, I achieved a balance between interesting layouts and guaranteed navigability.

1.4.3 AI and Interaction Systems

Developing adaptive AI with dynamic behaviour was a key challenge, particularly implementing enemy movement patterns that felt intelligent without being unfair to players. I employed state machines for enemy behaviour, allowing for modular and scalable designs. Enemies can now track the player, choose optimal paths, and adapt their tactics based on their remaining health and player position. This system allows for future expansion with additional enemy types and behaviours.

1.4.4 Performance Issues

Procedural generation and real-time interactions proved resource-intensive, leading to occasional frame drops, particularly when generating larger dungeon floors. Optimisation strategies were implemented, including:

- Object pooling for frequently used game elements
- Selective updating of off-screen entities
- Efficient collision detection using grid-based lookups rather than physics bodies
- Deferred loading of resources to distribute processing load

These optimisations resolved most performance bottlenecks, resulting in smooth gameplay even on larger dungeon levels.

1.4.5 Integrating Narrative with Gameplay

A unique challenge was integrating the story elements and dialogue system with the procedural gameplay. Creating a cohesive narrative experience within a randomly generated environment required careful design of the dialogue system and trigger points. The solution involved creating narrative "anchors" at specific progress points (beginning, boss encounters, etc.) while allowing the procedural elements to fill in the exploratory aspects.

Chapter 2: Background Research

2.1 Procedural Generation

Procedural generation is a technique used in computer science and digital content creation where data is produced algorithmically rather than manually. This method has become essential in various fields, particularly in video game development, where it is employed to create expansive and diverse worlds. I have researched the background of procedural generation, its origins, benefits, and its specific application in rogue-like dungeon crawling games.

2.1.1 Background and Origins

The concept of procedural generation can trace its roots back to the early days of computing. One of its earliest applications was in the creation of fractals and noise functions, such as Perlin noise, which allowed for the generation of complex, natural-looking textures and terrains with simple mathematical algorithms [3]. This technique gained significant traction in the 1980s with the advent of games like "Elite," a space trading game developed by David Braben and Ian Bell [4]. "Elite" used procedural generation to create an entire galaxy of star systems on the limited memory of early computers, showcasing the potential of this method to expand virtual worlds beyond physical hardware constraints.

2.1.2 Benefits of Procedural Generation

Procedural generation enables the creation of varied environments with relatively low computational and memory resources. This is crucial for developers working within the limitations of hardware, particularly in the era of early personal computers and gaming consoles. Secondly, procedural generation promotes replayability and uniqueness. Games that utilise procedural generation can offer players new experiences each time they play, as the content is dynamically generated rather than predefined [5].

Moreover, procedural generation can significantly reduce the workload for developers. Creating large, intricate worlds by hand is time-consuming and costly. By employing algorithms to generate content, developers can focus more on refining core gameplay mechanics and narrative elements. Additionally, procedural generation allows for the creation of content that is more adaptable to player actions [6].

2.1.3 Procedural Generation in Rogue-like Dungeon Crawlers

Rogue-like dungeon crawling games are a genre that heavily benefits from procedural generation. Named after the 1980 game "Rogue," these games typically feature randomized dungeons, permadeath mechanics, and turn-based gameplay [7]. The use of procedural generation in rogue-like games ensures that each playthrough is unique, challenging players to adapt to new layouts, enemy placements, and item distributions.

In a typical rogue-like dungeon crawler, procedural generation algorithms create the layout of each dungeon floor, determining the placement of rooms, corridors, enemies, and treasures. These algorithms often use techniques such as random walks, cellular automata, and binary space partitioning to create diverse and navigable dungeon environments [1].

For instance, the procedural dungeon generation script in a game might start by defining an initial grid of cells. It then employs a random walk algorithm to carve out a path through the grid, ensuring connectivity. Additional features, such as treasure rooms and staircases to lower floors, are added based on predefined probabilities and rules. This process ensures that each dungeon floor offers a distinct challenge while adhering to the overall design principles of the game [8].

2.2 Modern Applications in Game Development

2.2.1 Contemporary Examples

In modern game development, procedural generation has evolved beyond simple dungeon layouts to encompass entire worlds, ecosystems, and narratives. Games like "No Man's Sky" (Hello Games, 2016) use procedural generation to create entire planets and star systems, each with unique flora, fauna, and terrain [19]. Similarly, "Minecraft" (Mojang Studios, 2011) generates virtually infinite worlds with varied biomes, cave systems, and structures [20].

Within the roguelike genre specifically, titles like "The Binding of Isaac" (Edmund McMillen & Florian Himsl, 2011) and "Hades" (Supergiant Games, 2020) demonstrate the modern evolution of procedural dungeons. "The Binding of Isaac" generates rooms with varied layouts, enemy placements, and item distributions, while maintaining narrative consistency through themed floors and boss encounters [21]. "Hades" employs a more controlled form of procedural generation, where room layouts are predetermined but their arrangement and rewards are dynamically generated [22].

2.2.2 Implementation in Godot Engine

The Godot Engine provides several tools and frameworks that facilitate procedural generation. These include built-in noise functions, spatial partitioning systems, and efficient data structures for managing large generated worlds [23]. While Godot lacks dedicated procedural generation tools compared to some specialized engines, its flexible architecture allows developers to implement custom algorithms suited to their specific needs.

For grid-based dungeon generation specifically, Godot's TileMap and TileSet systems provide efficient ways to represent and render procedurally generated dungeons [23]. Additionally, the engine's scene-based architecture allows for modular approaches to procedural content, where individual components (rooms, enemies, items) can be instantiated and combined algorithmically.

2.2.3 Algorithms and Techniques

Modern procedural dungeon generation typically employs one or more of the following algorithms:

1. **Random Walk:** The approach used in this project, where a path is carved through a grid by taking random steps from a starting position. This method creates organic, cave-like structures with winding corridors [8].
2. **Binary Space Partitioning (BSP):** Recursively divides space into smaller sections, then places rooms within these sections and connects them with corridors. This tends to create more structured, room-based dungeons [24].
3. **Cellular Automata:** Uses rule-based systems to iteratively modify a grid, typically starting with random noise and applying rules to form coherent structures. This method is particularly effective for creating realistic cave systems [25].
4. **Wave Function Collapse:** A newer approach that generates content by observing patterns in example inputs and applying constraints to ensure coherent outputs. This can create highly structured dungeons with consistent themes and architectural features [26].
5. **Generative Adversarial Networks (GANs):** Although still experimental in game development, machine learning approaches like GANs show promise for creating complex, varied, and thematically consistent procedural content [27].

Each of these approaches offers different trade-offs in terms of computational complexity, structural coherence, and artistic control. For my project, I selected the random walk algorithm due to its relative simplicity, organic results, and suitability for grid-based gameplay systems. While the cellular automata approach proved effective for organic dungeon shapes, other techniques such as BSP provide more structured layouts, often better suited for room-based design. Wave Function Collapse, though more complex, offers constraint-based generation ideal for puzzles or pattern-sensitive tilemaps.

Chapter 3: Methodology

3.1 Development Approach and UML Diagrams

The development of my game began by leveraging foundational mechanics adapted from Heartbeast's dungeon crawler tutorial [9] and Piotr's GitHub repository [10]. This Godot-based resource provided a starting point with its implementation of grid-based movement and cell mechanics.

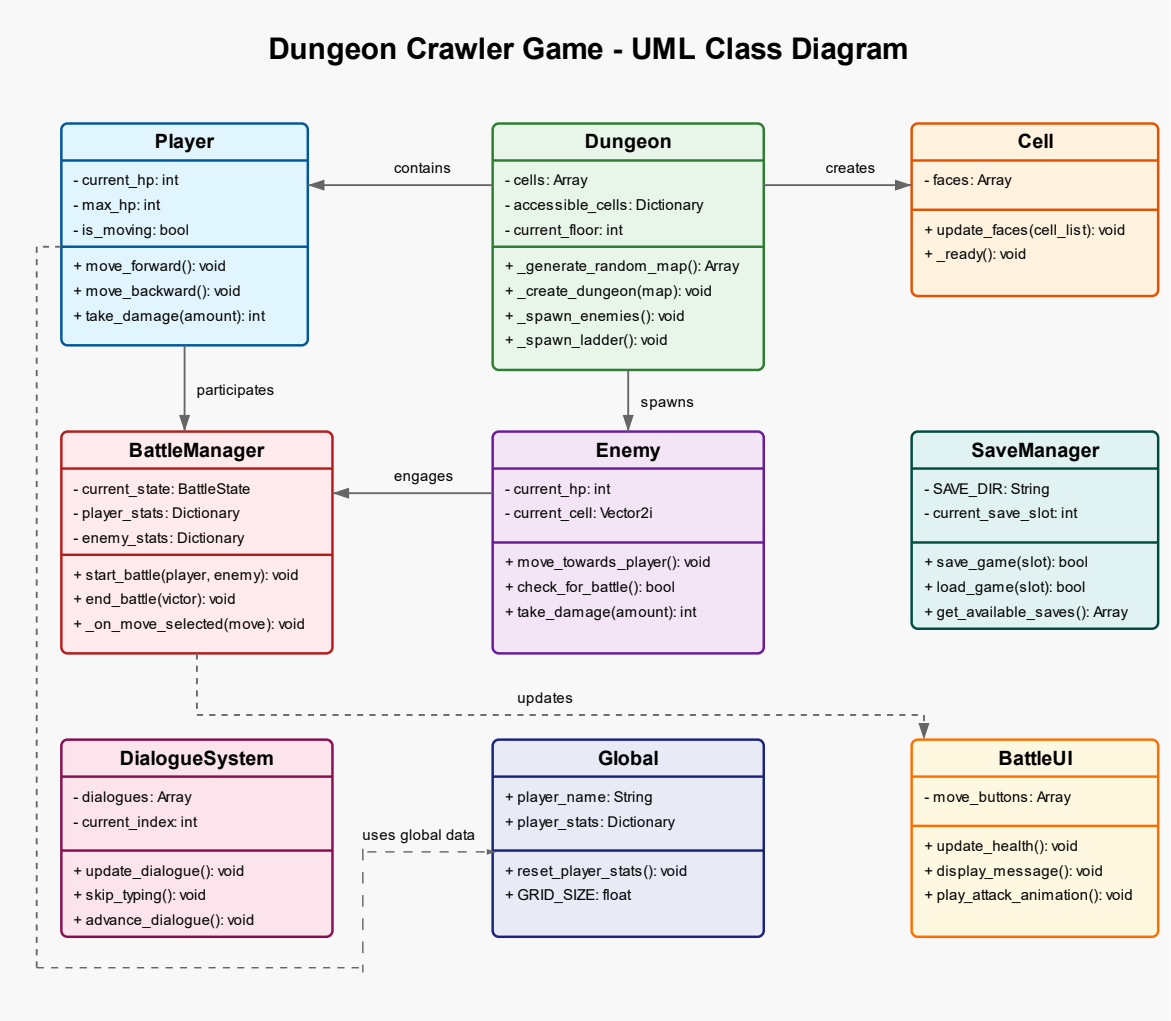

Figure 1 UML Class Diagram of Core Game Architecture

Figure 1 illustrates the primary classes that compose the game's architecture and their relationships. The diagram highlights how the Dungeon class serves as the central component, coordinating Player movement, Enemy behavior, and environmental elements such as Cells. The SaveManager provides persistence functionality, while the DialogueSystem and BattleManager handle narrative and combat interactions respectively. This modular design allows each component to handle its specific responsibilities while communicating through well-defined interfaces, exemplifying the separation of concerns principle.

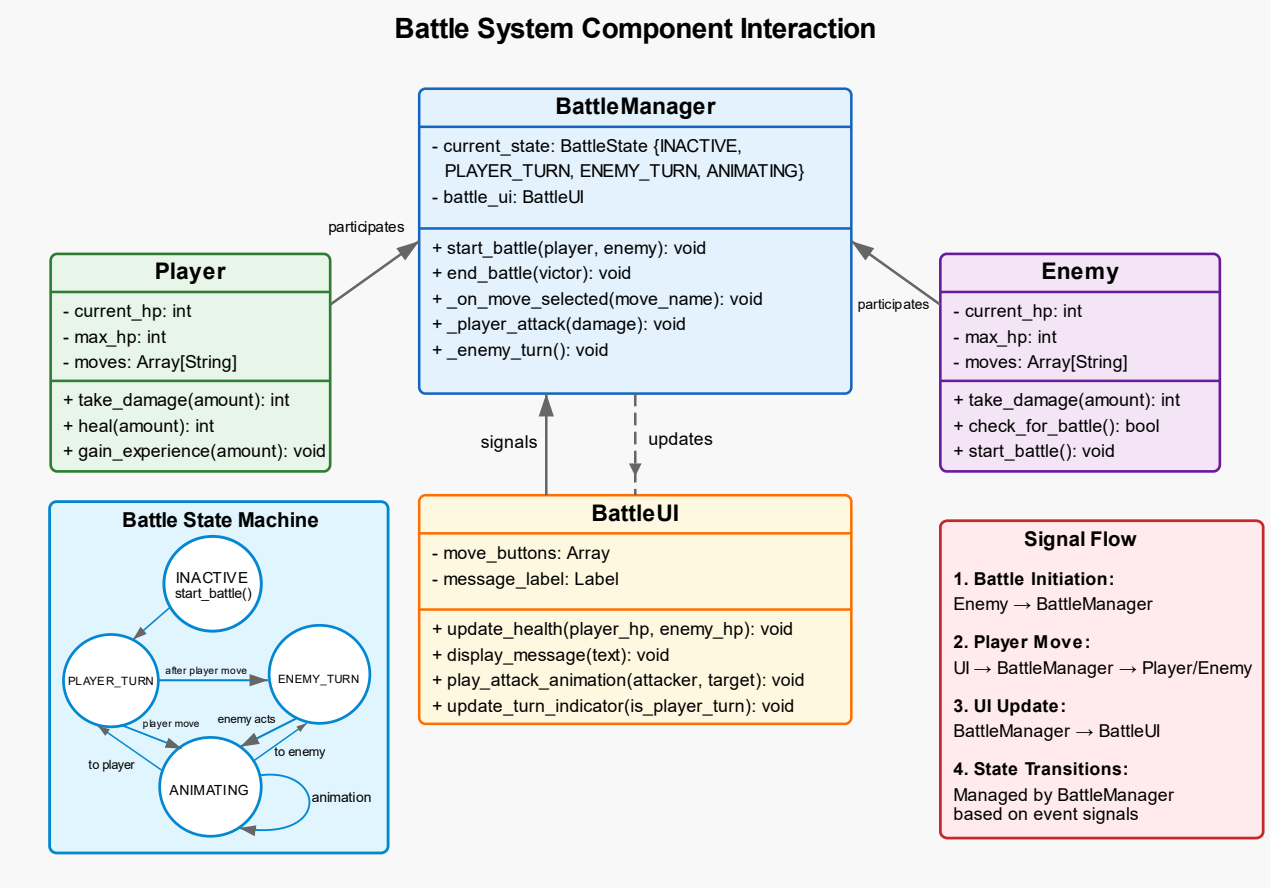

Figure 2 Battle System UML with State Machine Implementation

Figure 2 details the battle system architecture, showing the relationship between the BattleManager, Player, Enemy, and BattleUI classes. The diagram illustrates how the State pattern is implemented through the BattleState enum (INACTIVE, PLAYER_TURN, ENEMY_TURN, ANIMATING) to control combat flow. Signal connections between components demonstrate the Observer pattern, with UI events triggering manager responses and state transitions. The state machine visualization in the bottom left illustrates the transitions between different battle states, showing how combat progresses from initialization through player and enemy turns.

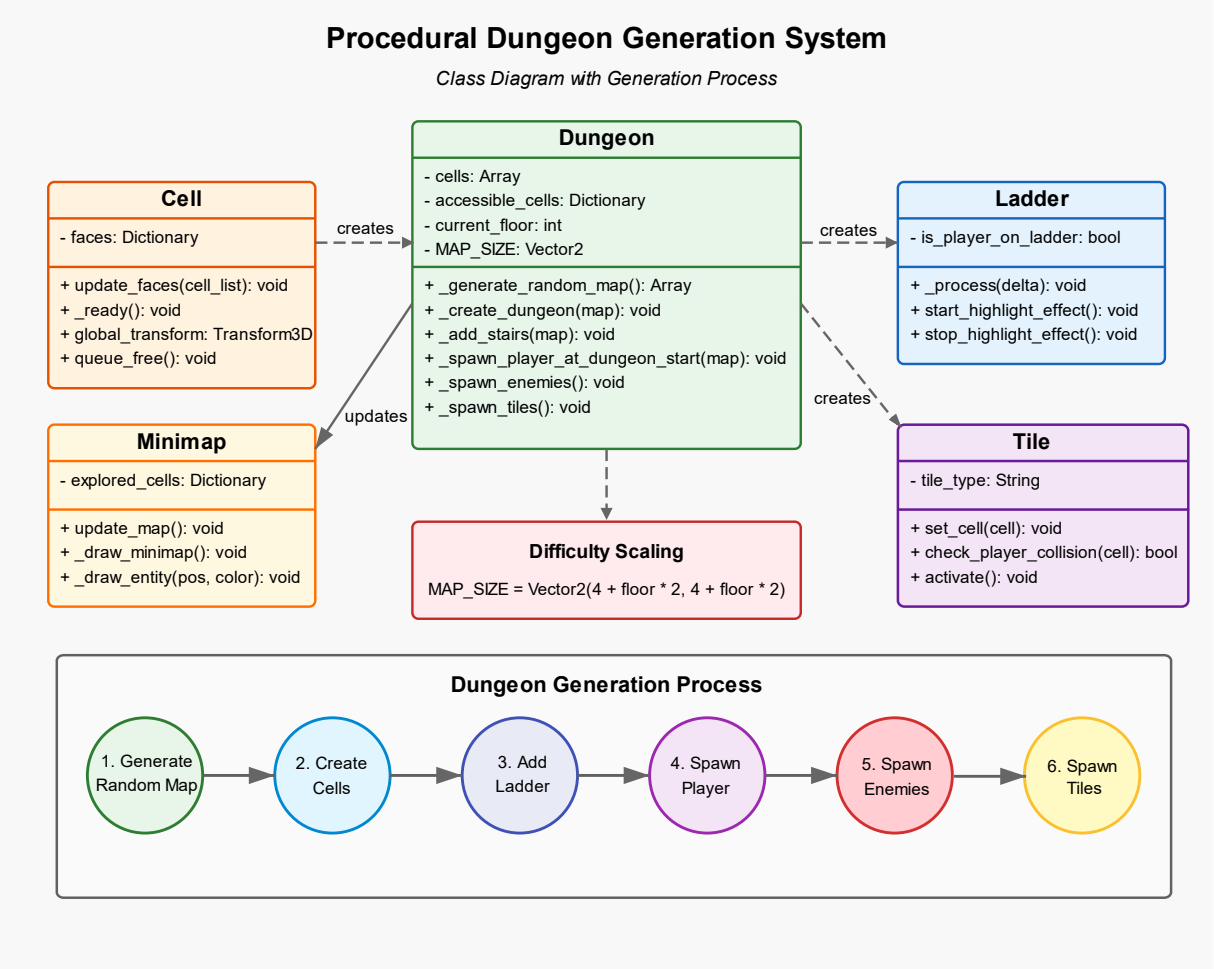

Figure 3 Procedural Dungeon Generation System and Process Flow

Figure 3 depicts both the class structure and sequential process of the procedural dungeon generation system. The Dungeon class utilizes the Factory pattern to instantiate Cells, Ladders, and Tiles based on the randomly generated map data. The bottom section illustrates the step-by-step generation process: map creation using the random walk algorithm, cell instantiation, placing the ladder, spawning the player, and finally populating the dungeon with enemies and interactive tiles. The diagram also shows how difficulty scaling is implemented by adjusting the map size based on the current floor number, creating progressively more complex environments as the player advances.

3.1.1 Agile Development Process

For this project, I adopted a modified agile development approach with iterative cycles that allowed for continuous refinement. Each development cycle consisted of:

1. **Planning**: Identifying features to implement in the current cycle
2. **Implementation**: Coding the planned features
3. **Testing**: Verifying the functionality and stability of new features
4. **Review**: Evaluating the cycle's outcomes and planning adjustments

This approach allowed for flexibility as the project evolved, enabling me to respond to discoveries and challenges as they emerged. Rather than adhering to strict sprint schedules, I organised development around functional milestones:

- **Milestone 1**: Core dungeon generation and player movement
- **Milestone 2**: Enemy AI and basic combat system
- **Milestone 3**: UI systems and scene transitions
- **Milestone 4**: Save/load system and data persistence
- **Milestone 5**: Narrative integration and final polishing.

3.1.2 Project Architecture

The final architecture of the project follows Godot's node-based design philosophy, with a structure that can be divided into several key components:

1. **Scene Management Layer**: Handles transitions between major game states (title screen, name input, town, castle, dungeon)
2. **Procedural Generation System**: Creates and manages dungeon layouts
3. **Character Controller System**: Manages player movement and interactions
4. **Combat System**: Implements turn-based battles between player and enemies
5. **UI Systems**: Handles all user interface elements across different game states
6. **Data Persistence System**: Manages saving and loading game state

This modular architecture allowed for independent development and testing of different game systems, reducing interdependencies and making the codebase more maintainable.

3.1.3 Procedural Generation

I incorporated procedural generation to dynamically construct dungeon layouts that evolve with each playthrough. My Dungeon.gd script houses methods such as _generate_random_map and _create_dungeon, which are central to generating randomized pathways and room configurations. These algorithms balance randomness with navigability by employing predefined constraints and heuristics. For instance, the random walk algorithm ensures paths remain interconnected while preventing overly sparse or dense layouts. Additionally, accessible tiles are precomputed and stored in structured arrays to streamline pathfinding and real-time interactions. No two dungeon layouts are identical.

The procedural generation system implements several key features:

- Dynamic difficulty scaling based on floor number
- Guaranteed connectivity between all dungeon areas
- Strategic placement of enemies, traps, and rewards
- Boss encounters at predetermined progression points

3.1.4 Interface and Scene Transitions

A sequence of interconnected scenes provides the narrative and structural flow of the game. Each scene is thoughtfully structured to support intuitive navigation throughout the player's journey:

- **Title Scene:** The entry point of the game, the title screen, features Button Nodes for starting a new game, continuing progress, accessing settings, or quitting. The title.gd script orchestrates these interactions through the methods _on_new_pressed and _on_quit_pressed.
- **Name Input Scene:** Upon initiating a new game, players are redirected to the Name Input Scene, where they input their character's name via a LineEdit node. The nameInput.gd script stores this input globally.
- **Dialogue Scene:** The story's introduction unfolds in the Dialogue Scene, where dialogue sequences are displayed through a RichTextLabel node. Managed by the scene1.gd script, this scene employs typing effects, implemented via _typing_timer_timeout. This scene sets the tone for the game, blending storytelling with visual elements.

- **Map/Town/Castle Scenes:** These scenes serve as hubs for player interaction, providing options for dialogue, dungeon access, and game state management. They implement a consistent UI system for player choices and transitions.
- **Dungeon Scene:** Central to the game, the Dungeon Scene presents a procedurally generated 3D environment. Movement is grid-based, with players using the up arrow key to advance and the left or right keys to rotate their view. The player.gd and Dungeon.gd scripts collaborate to synchronise player interactions with the environment.
- **Battle Scene:** When encountering enemies, the game transitions to the battle system, which manages turn-based combat between the player and enemies.

The modular architecture of these scenes facilitates independent testing and debugging and supports iterative refinement and scalability.

3.2 Testing and Iteration

Testing and iteration revealed functional and performance-related issues that required targeted solutions. Each iteration aimed to refine mechanics, improve performance, and enhance user experience.

3.2.1 Testing Methodology

I implemented several testing approaches throughout development:

1. **Unit Testing**: Critical algorithms, particularly the procedural generation system, were tested independently to verify correct behaviour. This included validation of dungeon connectivity, proper enemy placement, and collision detection accuracy.

2. **Integration Testing**: As components were combined, integration tests ensured proper communication between systems. For example, testing the interaction between player movement, enemy AI, and combat triggers.

3. **Playtesting**: Regular playtesting sessions, both by myself and volunteer testers, provided qualitative feedback on game feel, difficulty progression, and user interface clarity.

4. **Performance Profiling**: Using Godot's built-in profiling tools, I identified and addressed performance bottlenecks, particularly in the dungeon generation process and during complex combat scenarios.

3.2.1 Identified Issues and Solutions

Below is a table of key issues identified during development and how these issues were addressed:

Issue/Bug	Description	Solution Implemented
Movement Bug	Occasionally, the player was unable to move forward despite the absence of visible obstacles. This issue arose from discrepancies in the can_move method within player.gd, where accessible cells were not always accurately validated against the grid structure.	Implemented additional validation in the movement system to cross-check accessible_cells with the actual dungeon state. Added debug logging to track specific edge cases.

Duplicate Ladder Cells	On deeper dungeon levels, multiple stair cells occasionally appeared, disrupting intended gameplay logic. This anomaly stemmed from inconsistencies in the _add_ladder method within Dungeon.gd.	Added validation checks to ensure uniqueness of stair placement. Implemented a tracking system to maintain a single reference to the active ladder object.
Interaction Bug	At times, pressing the spacebar on a ladder cell failed to trigger the floor transition. This indicated a synchronization problem between the player's position and the ladder_cell variable.	Enhanced the interaction system with more robust position checking and improved the signaling between player and environment objects.
Performance Issues	Frame rate drops and occasional freezing during dungeon generation and complex combat scenarios.	Implemented object pooling for frequently instantiated objects like cells and projectiles. Optimised the random walk algorithm to reduce unnecessary calculations. Added distance-based culling for entities far from the player.
Save Data Corruption	Occasional corruption of save files when saving during certain game states.	Implemented transaction-based saving where data is first written to a temporary file, verified, and then moved to the final save location. Added checksums for save data integrity verification.

3.2.3 Iteration Process

The development followed an iterative process with three major development cycles:

1. **First Iteration**: Focused on core mechanics - dungeon generation, player movement, and basic enemy behavior.
2. **Second Iteration:** Enhanced the combat system, added UI elements, and implemented save/load functionality.
3. **Final Iteration:** Integrated narrative elements, refined the difficulty progression, and polished the overall user experience.

Each iteration involved extensive testing and refinement, with feedback from each phase informing the priorities for the next development cycle.

Chapter 4: Game Design Patterns

Game design patterns provide structured solutions to common problems in game development. They enable maintainability, scalability, and adaptability while ensuring efficient workflows. In my dungeon crawler, I have applied several important design patterns that enhance the procedural generation, player interaction, and overall game architecture.

4.1 Factory Pattern

The Factory pattern is a fundamental creational design strategy utilised for the efficient instantiation of objects. It facilitates the creation of elements at runtime, abstracting the instantiation process to ensure modularity and flexibility. Within my dungeon crawler, this pattern has been instrumental in the generation of dungeon cells and stair elements during procedural dungeon creation.

4.1.1 Factory Pattern Implementation

The Factory pattern is prominently employed in the _create_dungeon() and _add_ladder() methods of Dungeon.gd, where Cell and Ladder objects are dynamically instantiated based on procedural map data. For instance:

gdscript

```
1. var cell = Cell.instantiate()
2. add_child(cell)
3. cells.append(cell)
4. cell.global_transform.origin = Vector3(x * Global.GRID_SIZE, 0, y
* Global.GRID_SIZE)
```

This approach decouples the instantiation logic from the core game mechanics, enabling a flexible architecture where templates (preloaded scenes) such as Cell or Ladder can be modified independently of the logic governing their placement.

A similar implementation appears in the creation of enemies and interactive tiles:

gdscript

```
1. var enemy = Enemy.instantiate()
2. add_child(enemy)
3. enemy.set_cell(enemy_cell)
4. enemy.accessible_cells = accessible_cells
5. enemy.player = player
6. enemies.append(enemy)
```

This factory approach allows the dungeon generation system to populate the game world with various entities without needing to understand their internal implementation details.

4.1.2 Importance of the Factory Pattern

The Factory pattern underpins the procedural generation framework of my project. By dynamically instantiating objects, the system ensures efficient memory usage and adaptability to changing requirements, such as the introduction of new elements (e.g., traps or treasure chests). This pattern is crucial for maintaining scalability as the game environment evolves in complexity across different floors [11].

Additionally, the Factory pattern facilitated iterative development by allowing me to focus on improving object behaviour without disrupting the creation process. For example, I could enhance enemy behaviour logic without modifying the dungeon generation code that creates them.

4.2 State Pattern

The State pattern is a behavioral design paradigm that organises an object's behaviour around its current state, allowing dynamic transitions between states while maintaining modular and encapsulated logic. This approach is particularly relevant in games, where entities often exhibit varying behaviours depending on contextual conditions, such as movement, attack, or interaction.

4.2.1 State Pattern Implementation

In player.gd, the is_moving flag exemplifies the State pattern, governing transitions between idle and moving states. Coupled with the cooldown_timer.is_stopped() condition, this flag ensures that movement and other actions do not overlap:

gdscript

```
1. if cooldown_timer.is_stopped() and not is_moving:
2.     move(-global_transform.basis.z)
```

The cooldown timer further synchronises transitions, adding pacing and deliberate progression to the gameplay.

A more complex implementation of the State pattern appears in the battle_manager.gd script, which explicitly defines game states using an enum:

gdscript

```
1. enum BattleState {INACTIVE, PLAYER_TURN, ENEMY_TURN, ANIMATING}
2. var current_state = BattleState.INACTIVE
```

This approach allows for clear, state-dependent behaviour:

gdscript

```
 1. func _on_move_selected(move_name):
 2.     if current_state != BattleState.PLAYER_TURN:
 3.         return
 4.
 5.     current_state = BattleState.ANIMATING
 6.
 7.     # Process move logic...
 8.
 9.     if enemy_stats.hp <= 0:
10.         # Victory condition...
11.     else:
12.         current_state = BattleState.ENEMY_TURN
13.         _update_battle_ui()
14.         _enemy_turn()
```

4.2.2 Importance of the State Pattern

Encapsulation of behaviours within distinct states ensures clarity and maintainability in managing player interactions. Expanding the system to include new states, such as an attack or dodge, can be achieved seamlessly without introducing conflicts. The modular design afforded by this pattern simplifies debugging and facilitates the extension of gameplay mechanics [12].

In my project, the State pattern was particularly valuable for managing the turn-based combat system, where clear delineation between player and enemy turns was essential. It also simplified the handling of character animations and UI updates, as each state has specific visual and interactive requirements.

4.3 Command Pattern

The Command pattern encapsulates requests as objects, decoupling input handling from the logic that processes these inputs. This abstraction is critical in games, where inputs can trigger complex sequences of events.

4.3.1 Command Pattern Implementation

The _input() method in player.gd demonstrates the Command pattern by mapping player inputs to specific actions. For instance:

gdscript

```gdscript
func _input(event):
    if event.is_action_pressed("ui_up"):
        move(-global_transform.basis.z)
    elif event.is_action_pressed("ui_left"):
        rotate_smoothly(deg_to_rad(90))
        move_direction = -global_transform.basis.x
    elif event.is_action_pressed("ui_right"):
        rotate_smoothly(deg_to_rad(-90))
        move_direction = global_transform.basis.x
```

This structure allows the input handling logic to remain independent of the mechanics it invokes. Furthermore, this abstraction facilitates future enhancements, such as adding support for gamepads or reconfigurable input schemes.

The Command pattern is also evident in the battle system, where player move selection is decoupled from the effects of those moves:

gdscript

```gdscript
func _on_move_button_pressed(move_name):
    emit_signal("move_selected", move_name)

    for button in move_buttons:
        button.disabled = true
```

4.3.2 Importance of the Command Pattern

The Command pattern ensures that the input system is extensible and testable, allowing new commands to be integrated without impacting existing logic [11]. This modularity enhances the responsiveness of the control scheme and supports iterative development.

In my game, this pattern proved invaluable for maintaining a clean separation between user input and game logic, particularly in the complex battle system where player choices trigger a series of animations, state changes, and statistical calculations.

4.4 Game Loop Pattern

The Game Loop pattern forms the foundation of game execution, managing updates to mechanics, rendering, and user input within a unified cycle. This ensures a consistent flow of gameplay across varying hardware capabilities.

4.4.1 Game Loop Pattern Implementation

The _physics_process(delta) method in player.gd illustrates the Game Loop pattern by handling player movement:

gdscript

```
 1. func _physics_process(delta):
 2.     if is_moving and not in_battle:  # Don't move during battle
 3.         var move_step = SPEED * delta
 4.         var distance =
global_transform.origin.distance_to(target_position)
 5.
 6.         if distance < move_step:
 7.             global_transform.origin = target_position
 8.             is_moving = false
 9.             emit_signal("player_moved")  # Emit signal when
player finishes moving
10.         else:
11.             global_transform.origin =
global_transform.origin.move_toward(target_position, move_step)
```

Delta time normalisation ensures that movement remains consistent across different frame rates, providing a smooth player experience.

Similarly, enemy movement in slime.gd uses the game loop to update positions gradually:

gdscript

```
 1. func _physics_process(delta):
 2.     if is_moving and not in_battle:  # Don't move during battle
 3.         var move_step = move_speed * delta
 4.         var distance =
global_transform.origin.distance_to(target_position)
 5.
 6.         if distance < move_step:
 7.             global_transform.origin = target_position
 8.             is_moving = false
 9.             check_for_battle()
10.         else:
11.             global_transform.origin =
global_transform.origin.move_toward(target_position, move_step)
```

4.4.2 Importance of the Game Loop Pattern

The Game Loop pattern synchronises core mechanics, ensuring real-time responsiveness and maintaining consistency in gameplay dynamics. It enables scalability by integrating new subsystems, such as advanced physics or animations, without disrupting the existing flow [12].

In my project, the Game Loop pattern was essential for creating smooth movement animations, coordinating enemy AI behaviours, and ensuring that the game world responds consistently to player actions regardless of the hardware's performance capabilities.

4.5 Observer Pattern

The Observer pattern establishes a one-to-many dependency between objects, where changes to one object (the subject) automatically notify and update all dependent objects (the observers). This pattern is particularly useful in game development for event handling and communication between loosely coupled components.

4.5.1 Observer Pattern Implementation

In my project, the Observer pattern is implemented primarily through Godot's signal system. For example, in player.gd, the player emits a signal when movement is completed:

gdscript

```gdscript
signal player_moved

func _physics_process(delta):
    if is_moving and not in_battle:
        var move_step = SPEED * delta
        var distance = global_transform.origin.distance_to(target_position)

        if distance < move_step:
            global_transform.origin = target_position
            is_moving = false
            emit_signal("player_moved")  # Notify observers of movement completion
        else:
            global_transform.origin = global_transform.origin.move_toward(target_position, move_step)
```

This signal is then connected to the dungeon controller, which observes the player's movement to trigger various game events:

gdscript

```gdscript
func _on_player_moved():
    # Add current cell to explored cells
    var player_cell = Vector2i(
        floor(player.global_transform.origin.x / GRID_SIZE),
        floor(player.global_transform.origin.z / GRID_SIZE)
    )

    explored_cells[player_cell] = true

    # Update minimap
    _update_minimap()

    # Check for battles with enemies
    var battle_started = false
    for enemy in enemies:
        if is_instance_valid(enemy):
            if enemy.check_for_battle():
                battle_started = true
                break
```

Similarly, the battle system uses signals to communicate between components:

gdscript

```gdscript
signal battle_started(player, enemy)
signal battle_ended(victor)
signal turn_ended

func start_battle(player, enemy):
    if current_state != BattleState.INACTIVE:
        return

    current_flee_chance = flee_base_chance

```

```
11.     print("Battle Manager: Starting battle between player and
enemy")
12.     current_player = player
13.     current_enemy = enemy
14.     current_state = BattleState.PLAYER_TURN
15.
16.     _unshade_enemy_sprite()
17.
18.     # Additional setup...
19.
20.     emit_signal("battle_started", player, enemy)
```

4.5.2 Importance of the Observer Pattern

The Observer pattern enables loose coupling between game components, allowing them to interact without direct dependencies. This improves modularity, testability, and maintainability [11]. In my project, this pattern was crucial for creating a responsive game world that reacts to player actions without requiring tight integration between systems.

The most significant benefit was in separating the dungeon generation, enemy behaviour, and combat systems—each could operate independently while still communicating effectively through signals. This approach simplified debugging and made it easier to extend functionality without disrupting existing systems.

Chapter 5: Technical Achievements

5.1 Procedural Dungeon Generation

5.1.1 Generating a Dungeon Map

A random walk algorithm has been implemented, generating interconnected rooms and corridors dynamically. This foundational system ensures varied and engaging dungeon layouts for players. The tilemap system in Godot 4 was utilised to efficiently manage grid-based dungeon creation, offering flexibility for future enhancements such as varying room types and environmental hazards.

My function _generate_random_map() creates a 2D grid (map) of a predefined size (MAP_SIZE) and fills it with tiles marked as either walkable or non-walkable. A random walk algorithm is used to carve out a path of accessible cells in the dungeon, starting from the center of the grid. For the generation process to start at the center of the map, this center point is calculated in line 9. This cell is then marked as accessible (1 in the map array) and its position is recorded in a separate structure (accessible_cells) which records walkable areas.

gdscript

```
 1. func _generate_random_map() -> Array:
 2.     var map = []
 3.     for y in range(MAP_SIZE.y):
 4.         var row = []
 5.         for x in range(MAP_SIZE.x):
 6.             row.append(0)
 7.         map.append(row)
 8.
 9.     var gen_position = Vector2(floor(MAP_SIZE.x / 2),
floor(MAP_SIZE.y / 2))
10.     map[gen_position.y][gen_position.x] = 1
11.     accessible_cells[Vector2i(gen_position.x, gen_position.y)] =
true
12.     for i in range(MAX_STEPS):
13.         var direction = randi() % 4
14.         if direction == 0: gen_position.y -= 1
15.         elif direction == 1: gen_position.y += 1
16.         elif direction == 2: gen_position.x -= 1
17.         elif direction == 3: gen_position.x += 1
18.
19.         gen_position.x = clamp(gen_position.x, 0, MAP_SIZE.x - 1)
20.         gen_position.y = clamp(gen_position.y, 0, MAP_SIZE.y - 1)
21.
22.         map[gen_position.y][gen_position.x] = 1
23.         accessible_cells[Vector2i(gen_position.x,
gen_position.y)] = true
24.
25.     return map
```

Random Walk Generation is implemented from line 12. This function performs multiple steps, up to MAX_STEPS, to create the dungeon layout. At each step, a random direction is chosen (0 for up, 1 for down, 2 for left and 3 for right). The gen_position variable is updated to move one cell in the chosen direction, and boundary checks are performed using clamp() to ensure that the position stays within the map dimensions. Accessible tiles are marked in both the map array and the accessible_tiles dictionary. This ensures the walkable path is recorded in multiple data structures for flexible usage, which will be explained in more detail later.

Finally, the 2D array map is returned, representing the dungeon layout. Walkable tiles are denoted by 1, and non-walkable tiles remain 0.

Figure 4 Example of a Generated Dungeon in the Game

The random walk algorithm ensures that each map generated is unique while maintaining a connected path of accessible cells. The accessible_cells dictionary allows for quick lookups of walkable areas, useful for movement logic and collision handling.

For a grid size of 6x6, the map array could look like this after a random walk:

```
[
[1, 1, 1, 1, 1, 1],
[1, 1, 1, 1, 1, 1],
[1, 1, 1, 1, 0, 1],
[1, 1, 1, 1, 1, 1],
[1, 1, 1, 0, 0, 0],
[1, 0, 0, 0, 0, 0]
]
```

This is a sample output corresponding with Figure 4. The layout reflects the unique and random nature of the dungeon's procedurally generated path. The algorithm ensures accessibility and avoids isolated tiles, making it suitable for gameplay exploration.

5.1.2 Creating the Dungeon

The _create_dungeon() function is responsible for constructing the dungeon in the game world based on a given random map.

gdscript

```
1. func _create_dungeon(random_map: Array):
2.     var used_tiles = []
3.
4.     for y in range(MAP_SIZE.y):
5.         for x in range(MAP_SIZE.x):
6.             if random_map[y][x] == 1:
7.                 var cell = Cell.instantiate()
8.                 add_child(cell)
```

```
 9.                 cells.append(cell)
10.                 cell.global_transform.origin = Vector3(x *
Global.GRID_SIZE, 0, y * Global.GRID_SIZE)
11.                 used_tiles.append(Vector2i(x, y))
12.
13.     for cell in cells:
14.         cell.update_faces(used_tiles)
```

The function takes a 2D array (random_map) as input, which represents the dungeon layout, and generates physical cells (game objects) in the game world for all accessible tiles (1 in the map). It also updates the appearance of each cell to reflect its surroundings by checking adjacent tiles.

An empty list, used_tiles, is initialized (line 2) to keep track of all positions in the dungeon that have been marked as accessible and for which cells are created. The function then continues to iterate over the 2D grid defined by MAP_SIZE. For each tile, if the value in random_map[y][x] is 1 (indicating an accessible tile), then the following steps occur:

1. A new Cell instance is created using the Cell.instantiate() method.
2. The cell is added as a child of the dungeon node (add_child(cell)), making it part of the scene graph.
3. The cells array is updated to include this new cell, keeping track of all created cells.
4. The global_transform.origin property is set to position the cell in 3D space. This position is calculated in line 10. Global.GRID_SIZE defines the distance between adjacent cells in the grid.
5. The tile's position (Vector2i(x, y)) is added to used_tiles, marking it as utilised.

After all cells are instantiated, the function iterates through the cells array. For each cell, the update_faces() method is called, passing the used_tiles list as a parameter. The update_faces() method ensures that the visual representation of each cell reflects its surroundings. For instance, if a neighboring tile is not accessible, the cell might display a wall or boundary face. But if all adjacent tiles are accessible, the cell might display open faces.

gdscript

```
 1. func update_faces(cell_list: Array) -> void:
 2.     var my_grid_position =
Vector2i(floor(global_transform.origin.x / Global.GRID_SIZE),
floor(global_transform.origin.z / Global.GRID_SIZE))
 3.
 4.     if cell_list.has(my_grid_position + Vector2i.RIGHT):
 5.         eastFace.visible = false
 6.     if cell_list.has(my_grid_position + Vector2i.LEFT):
 7.         westFace.visible = false
 8.     if cell_list.has(my_grid_position + Vector2i.DOWN):
 9.         southFace.visible = false
10.     if cell_list.has(my_grid_position + Vector2i.UP):
11.         northFace.visible = false
```

A key feature of my function is that cells are placed in the game world dynamically, based on the procedural map. This allows the dungeon to adapt to the generated layout and ensures that cells align with accessible tiles. Moreover, each cell is positioned in 3D space (Vector3), aligning with a grid-based system. This ensures consistency in spacing and orientation across the dungeon.

5.1.3 Scaling Difficulty

To ensure that the game provides increasing challenge as the player progresses, I implemented a difficulty scaling system within the dungeon generation process. This system adjusts several parameters based on the current floor number:

gdscript

```gdscript
 1. func generate_new_dungeon(floor: int):
 2.     current_floor = floor
 3.     MAP_SIZE = Vector2(4 + floor * 2, 4 + floor * 2)
 4.
 5.     # Clear previous floor
 6.     for cell in cells:
 7.         cell.queue_free()
 8.     cells.clear()
 9.     accessible_cells.clear()
10.     ladder_cell = Vector2i()
11.
12.     # Generate new floor
13.     var random_map = _generate_random_map()
14.     _create_dungeon(random_map)
15.     _add_ladder(random_map)
16.     _spawn_player_at_dungeon_start(random_map)
17.
18.     # Assign references to player
19.     player.accessible_cells = accessible_cells
20.     player.ladder_cell = ladder_cell
21.     player.update_ui_label()
```

The key scaling factors include:

1. **Dungeon Size:** As shown in line 3, the map size increases with each floor, growing by 2 cells in both dimensions per floor.

2. **Enemy Count and Difficulty:** Enemy spawning is tied to floor number:

gdscript

```gdscript
var num_enemies_to_spawn = num_enemies + floor(current_floor / 2)
```

3. **Trap Frequency**: More hazardous tiles appear on deeper floors:

gdscript

```gdscript
var num_tiles = max(0, floor(current_floor / 2))
```

4. **Boss Encounter**: On the fifth floor, a special boss enemy is spawned:

gdscript

```gdscript
1. if current_floor == BOSS_FLOOR:
2.     _spawn_boss()
3.     _spawn_tiles()
4. elif current_floor == 0:
5.     _spawn_tiles()
6. else:
7.     _spawn_enemies()
8.     _spawn_tiles()
```

This scaling system ensures that players face appropriately increasing challenges as they progress, maintaining engagement through a balance of difficulty and accomplishment.

5.2 Player Movement

5.2.1 Movement Function

The purpose of the move() function is to attempt movement in a given direction. It checks if the desired move is valid, and if so, calculates the new target position, starts the movement process, and triggers a cooldown timer to regulate movement speed.

gdscript

```
 1. func move(_direction: Vector3):
 2.     if can_move(move_direction):
 3.         target_position = global_transform.origin +
move_direction.normalized() * GRID_SIZE
 4.         is_moving = true
 5.         cooldown_timer.start()
 6.         print_debug("Moving to target position:",
target_position)
 7.     else:
 8.         print_debug("Cannot move to move_direction:",
move_direction)
 9.         pass
```

The function first calls the can_move() method, passing _direction, a Vector3 representing the desired direction of movement. This method determines if the move is valid (e.g., if the target cell is accessible). If the method returns true, then the function proceeds with movement, otherwise it aborts the process and logs a debug message.

If the move is valid, the function computes the target_position (line 3), then sets the is_moving flag to true. The cooldown_timer.start() call initiates a timer that regulates how quickly the player can make consecutive moves, preventing the player from moving continuously without pauses.

5.2.2 User Input

The input() function processes user inputs, enabling the player to control the main character's movement, rotate the character smoothly and interact with specific game objects like ladder to navigate between dungeon floors.

gdscript

```
 1. func _input(event):
 2.     if in_battle:
 3.         return  # Don't process movement input during battle
 4.
 5.     if not (is_moving or is_rotating) and (cooldown_timer == null
or cooldown_timer.is_stopped()):
 6.         if event.is_action_pressed("ui_up"):
 7.             move_forward()
 8.         elif event.is_action_pressed("ui_down"):
 9.             move_backward()
10.         elif event.is_action_pressed("ui_left"):
11.             rotate_player(deg_to_rad(90))  # Rotate left 90
degrees
12.         elif event.is_action_pressed("ui_right"):
13.             rotate_player(deg_to_rad(-90))  # Rotate right 90
degrees
14.         elif event.is_action_pressed("ui_accept"):
15.             var current_cell = get_current_cell()
16.             if current_cell == ladder_cell:
17.                 print("Player is on ladder and pressed spacebar")
```

This function only processes if the player is not in battle, is not currently moving or rotating, and the cooldown timer has stopped. This is to ensure that the player is not already in the process of moving and that the movement cooldown has elapsed.

If the ui_up action is pressed, then the player moves forward in the direction they are facing. If the ui_down action is pressed, the player moves backward. If the ui_left or ui_right action is pressed, then the player rotates smoothly 90 degrees to the corresponding direction.

The rotate_player function provides smooth visual rotation:

gdscript

```
1. func rotate_player(angle):
2.     is_rotating = true
3.     cooldown_timer.start()
4.
5.     var tween = get_tree().create_tween()
6.     var end_rot = rotation + Vector3(0, angle, 0)
7.     tween.tween_property(self, "rotation", end_rot,
ROTATION_SPEED)
8.     tween.tween_callback(func(): is_rotating = false)
```

The function also checks if the character's current cell matches the location of a ladder (ladder_cell) and if the ui_accept action is triggered.

5.2.3 Animation and Visual Feedback

To enhance the player experience, I implemented visual feedback for movement and interactions:

1. Smooth Rotation: Rather than instant turns, the player rotates smoothly using tweens:

gdscript

```
1. var tween = get_tree().create_tween()
2. var end_rot = rotation + Vector3(0, angle, 0)
3. tween.tween_property(self, "rotation", end_rot,
ROTATION_SPEED)
```

2. Positional Interpolation: Movement between grid cells is animated:

gdscript

```
global_transform.origin =
global_transform.origin.move_toward(target_position,
move_step)
```

3. Interactive Highlights: Interactable objects like ladders provide visual feedback:

gdscript

```
 1. func start_highlight_effect():
 2.     if pulse_tween:
 3.         pulse_tween.kill()
 4.
 5.     pulse_tween = create_tween().set_loops()
 6.     var start_scale = sprite_3d.scale
 7.
 8.     # Pulse effect
 9.     pulse_tween.tween_property(sprite_3d, "scale",
start_scale * 1.2, 0.5)
10.     pulse_tween.tween_property(sprite_3d, "scale",
start_scale, 0.5)
```

These visual enhancements significantly improve the game feel, providing clear feedback for player actions and making the grid-based movement system feel more natural and responsive.

5.3 Collision Handling

Instead of relying on traditional collision detection methods (e.g. physics bodies and collision shapes), this implementation uses a logical grid system to determine whether movement is valid. This grid consists of cells tracked in a dictionary (accessible_cells) which defines the accessible areas of the dungeon. The movement is validated by checking if the target cell exists in the dictionary.

5.3.1 Checking if the Player Can Move

The function can_move() determines whether the player can move in the specified direction.

gdscript

```
1. func can_move(_direction: Vector3) -> bool:
2.     var current_cell = get_cell_position(global_transform.origin)
3.     var target_cell = get_cell_position(global_transform.origin +
move_direction.normalized() * GRID_SIZE)
4.     print_debug("Current cell: ", current_cell, " Target cell: ",
target_cell, " Can move: ", accessible_cells.has(target_cell))
5.     return accessible_cells.has(target_cell)
```

It takes the parameter _direction, a Vector3 representing the direction of desired movement. The current cell is calculated with get_cell_position(), which converts the player's current world position into a grid cell position. Then the target cell position is derived by adding the normalised movement direction (move_direction.normalized()) scaled by GRID_SIZE to the player's current world position. If the target cell exists in the accessible_cells dictionary, then the function returns true.

gdscript

```
1. func get_cell_position(world_position: Vector3) -> Vector2i:
2.     return Vector2i(floor(world_position.x / GRID_SIZE),
floor(world_position.z / GRID_SIZE))
```

This utility function converts a 3D world position into a 2D grid cell position. The process involves:

1. Divide the x and z coordinates of the world position by GRID_SIZE to map them to the corresponding grid cell.
2. Use floor() to ensure values are integers (grid-aligned).
3. Return a Vector2i representing the grid cell.

5.3.2 Collision Handling Process

Grid-based validation is used; when a movement is initiated, the can_move() function checks if the target cell is present in accessible cells. Since movement is restricted to these accessible cells, entities cannot pass through walls or invalid areas, eliminating the need for physical collision shapes. This system integrates with other movement-related functions such as move() and _input().

An advantage of this approach is that the logical grid checks are computationally lighter than traditional collision detection methods, especially for grid-based games. Additionally, the accessible cells dictionary allows for easy modification of the navigable space, dynamically adapting to the procedural dungeon layouts. By aligning movement with the grid, this approach allows for precise and consistent motion, avoiding floating-point inaccuracies.

5.3.3 Enemy Collision Detection

Enemy movement uses the same grid-based system, but with additional logic to detect player proximity:

gdscript

```
func check_for_battle() -> bool:
    if player == null or in_battle:
        return false

    var player_cell = Vector2i(
        floor(player.global_transform.origin.x / GRID_SIZE),
        floor(player.global_transform.origin.z / GRID_SIZE)
    )

    if current_cell == player_cell:
        print("BATTLE STARTED with enemy at cell: ",
current_cell)
        start_battle()
        return true

    return false
```

This consistent approach to collision detection across all game entities ensures predictable behaviour and efficient performance, particularly in complex dungeon layouts with many entities.

5.4 Combat System

The combat system in my game implements a turn-based battle mechanics inspired by classic RPGs, with several modern enhancements for improved player engagement.

5.4.1 Battle State Management

At the core of the combat system is a state machine that manages the flow of battle:

gdscript

```
enum BattleState {INACTIVE, PLAYER_TURN, ENEMY_TURN, ANIMATING}
var current_state = BattleState.INACTIVE
```

This state machine ensures clear separation between different phases of combat, preventing overlap or race conditions between player and enemy actions.

5.4.2 Turn-Based Flow

The battle proceeds in clearly defined turns, with transitions managed through the state machine:

gdscript

```
func _on_move_selected(move_name):
    if current_state != BattleState.PLAYER_TURN:
        return

    print("Battle Manager: Player used ", move_name)
    current_state = BattleState.ANIMATING

    match move_name:
        "Tackle":
            _player_attack(15)
            await get_tree().create_timer(1.0).timeout
        # Other moves...

```

```
14.     if enemy_stats.hp <= 0:
15.         battle_ui.display_message("Enemy defeated!")
16.         get_tree().create_timer(1.0).timeout.connect(func():
end_battle(current_player))
17.     else:
18.         if current_state != BattleState.INACTIVE:
19.             current_state = BattleState.ENEMY_TURN
20.             _update_battle_ui()
21.             _enemy_turn()
```

When a player selects a move, the system enters an ANIMATING state during which the move's effects are calculated and displayed. After this, if the enemy is still alive, control passes to the enemy turn.

5.4.3 Dynamic Enemy Behaviour

Enemy AI in combat selects from available moves based on contextual factors:

gdscript

```
 1. func _enemy_turn():
 2.     await get_tree().create_timer(1.0).timeout
 3.
 4.     if current_state != BattleState.ENEMY_TURN:
 5.         return
 6.
 7.     current_state = BattleState.ANIMATING
 8.
 9.     var available_moves = enemy_stats.moves
10.     var selected_move = available_moves[randi() %
available_moves.size()]
11.
12.     print("Battle Manager: Enemy used ", selected_move)
13.
14.     match selected_move:
15.         "Slime Attack":
16.             _enemy_attack(enemy_stats.attack)
17.         "Defend":
18.             enemy_stats.defense += 5
19.             // more code...
20.         // Boss-specific moves...
```

Different enemy types have different move sets, with boss enemies featuring special abilities that create more challenging and varied combat encounters.

5.4.4 Visual Feedback

The combat system includes rich visual feedback to enhance player experience:

gdscript

```
 1. func play_attack_animation(attacker, target, damage):
 2.     var attacker_name = attacker
 3.     if attacker == "player" and Global.player_name != null and
Global.player_name.length() > 0:
 4.         attacker_name = Global.player_name
 5.
 6.     message_label.text = attacker_name.capitalize() + " attacks
for " + str(damage) + " damage!"
 7.
 8.     var target_panel = $PlayerPanel if target == "player" else
$EnemyPanel
 9.
```

```
10.     var tween = create_tween()
11.     tween.tween_property(target_panel, "modulate", Color(1, 0, 0,
1), 0.2)
12.     tween.tween_property(target_panel, "modulate", Color(1, 1, 1,
1), 0.2)
```

These animations and effects help communicate the impact of combat actions, making battles more engaging despite their turn-based nature.

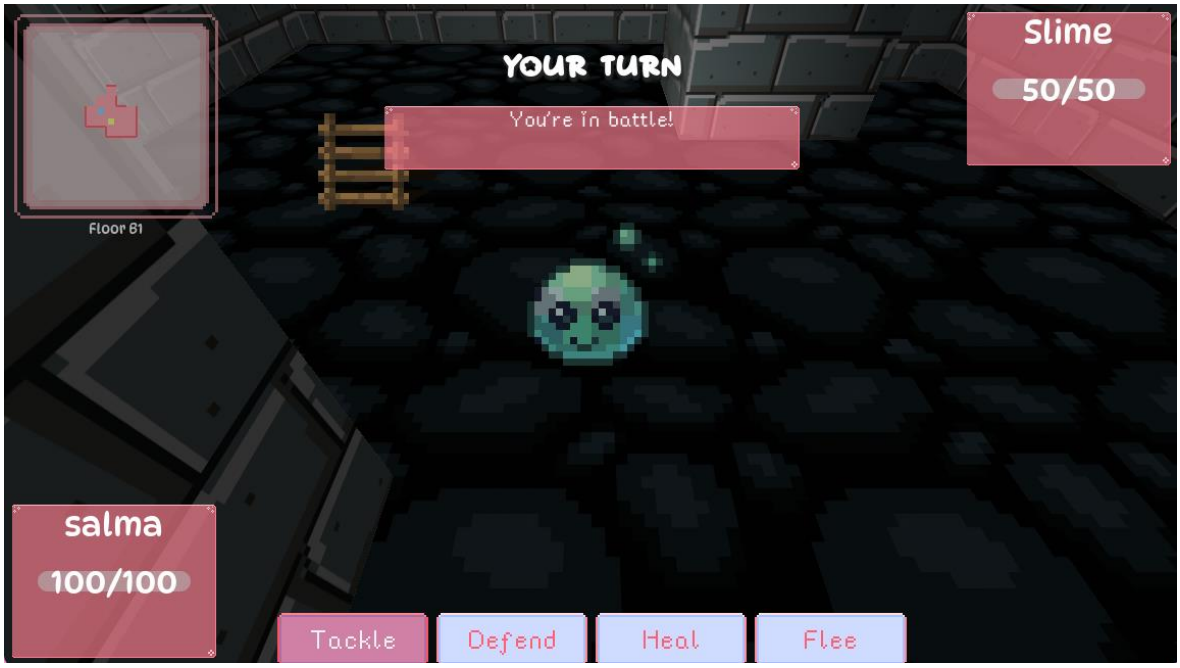

Figure 5 Screenshot displaying the combat UI when the user enters a battle in the game

5.4.5 Character Progression

The combat system ties into a character progression system that rewards successful battles with experience points and new abilities:

gdscript

```
 1. func gain_experience(amount):
 2.     experience += amount
 3.     Global.player_experience = experience
 4.     print("Gained ", amount, " experience! Total: ", experience,
"/", experience_to_next_level)
 5.
 6.     _show_floating_text("+" + str(amount) + " XP", Color(0.5, 1,
0.5))
 7.
 8.     if experience >= experience_to_next_level:
 9.         level_up()
```

The level-up system grants players increased stats and new combat abilities:

gdscript

```
 1. func level_up():
 2.     level += 1
 3.     Global.player_level = level
 4.
 5.     experience -= experience_to_next_level
 6.     Global.player_experience = experience
 7.
```

```
    experience_to_next_level = int(experience_to_next_level * 1.5)
    Global.player_experience_to_next_level = experience_to_next_level

    max_hp += 10
    Global.player_max_hp = max_hp

    // Stats improvements...

    if level == 3 and not "Firebolt" in moves:
        moves.append("Firebolt")
        Global.player_moves = moves.duplicate()
        print("Learned new move: Firebolt!")
    // More abilities...
```

This progression system gives players a sense of advancement and increasing power as they conquer more challenging dungeon floors.

5.5 Save/Load System

To enable persistence across play sessions, I implemented a comprehensive save/load system that preserves the player's progress, stats, and game state.

5.5.1 Save Data Structure

The save system organises data into a structured format for efficient storage and retrieval:

gdscript

```
func save_game(slot_number: int) -> bool:
    var save_path = get_save_path(slot_number)

    var save_data = {
        "version": "1.0",
        "timestamp": Time.get_unix_time_from_system(),
        "player_name": Global.player_name,
        "player_stats": {
            "max_hp": Global.player_max_hp,
            "current_hp": Global.player_current_hp,
            "attack": Global.player_attack,
            "defense": Global.player_defense,
            "level": Global.player_level,
            "experience": Global.player_experience,
            "experience_to_next_level": Global.player_experience_to_next_level,
            "moves": Global.player_moves
        }
    }

    // Save to file
    var file = FileAccess.open(save_path, FileAccess.WRITE)
    if file:
        var json_string = JSON.stringify(save_data)
        file.store_line(json_string)
        file.close()
        current_save_slot = slot_number
        emit_signal("save_completed")
```

```
28.         return true
29.     else:
30.         emit_signal("save_failed")
31.         print("Failed to save game: ",
FileAccess.get_open_error())
32.         return false
```

This structured approach makes the save data both human-readable (for debugging) and easily extensible for future enhancements.

5.5.2 File Management

The system manages save files with clear naming conventions and error handling:

gdscript

```
 1. func get_save_path(slot_number: int) -> String:
 2.     return SAVE_DIR + "save_" + str(slot_number) +
SAVE_FILE_EXTENSION
 3.
 4. func get_available_saves() -> Array:
 5.     var saves = []
 6.     var dir = DirAccess.open(SAVE_DIR)
 7.     if dir:
 8.         dir.list_dir_begin()
 9.         var file_name = dir.get_next()
10.
11.         while file_name != "":
12.             if !dir.current_is_dir() and
file_name.ends_with(SAVE_FILE_EXTENSION):
13.
14.                 var slot_str = file_name.replace("save_",
"").replace(SAVE_FILE_EXTENSION, "")
15.                 if slot_str.is_valid_int():
16.                     var slot = slot_str.to_int()
17.                     saves.append({
18.                         "slot": slot,
19.                         "file_name": file_name,
20.                         "date": get_save_date(slot)
21.                     })
22.             file_name = dir.get_next()
```

5.5.3 User Interface Integration

The save/load system connects seamlessly with a user-friendly interface:

gdscript

```
 1. func _create_slot_button(slot_number, save_exists, save_data):
 2.     var slot_button = slot_button_scene.instantiate()
 3.     save_slots_container.add_child(slot_button)
 4.
 5.     slot_button.setup(slot_number, save_exists, save_data,
is_save_mode)
 6.
 7.
slot_button.save_slot_selected.connect(_on_save_slot_selected)
 8.     slots.append(slot_button)
 9.
10.     return slot_button
```

This system allows players to manage multiple save files with clear information about each save's content and timestamp.

Figure 6 Screenshot of the Save Menu in the game

5.5.4 Error Handling and Data Integrity

To ensure save data integrity, the system implements robust error handling:

gdscript

```gdscript
func load_game(slot_number: int) -> bool:
    var save_path = get_save_path(slot_number)

    if !FileAccess.file_exists(save_path):
        print("Save file not found: ", save_path)
        emit_signal("load_failed")
        return false

    var file = FileAccess.open(save_path, FileAccess.READ)
    if file:
        var json_string = file.get_line()
        file.close()

        var json = JSON.new()
        var parse_result = json.parse(json_string)

        if parse_result == OK:
            var save_data = json.get_data()

            // Apply save data...

            return true
        else:
            print("Error parsing save file: ",
json.get_error_message())
            emit_signal("load_failed")
            return false
```

This approach ensures that corrupted save files are identified and properly handled, preventing data loss or application crashes.

Chapter 6: Professional Considerations

6.1 Ethical and Licensing Issues

Licensing issues present significant professional and ethical considerations. Below I focus on the complexities of software licensing, its ethical implications, and practical importance in game development.

6.1.1 Licensing: Understanding and Adherence

Licensing in software development involves the legal agreements that dictate how software can be used, modified, and distributed. Properly addressing licensing issues is crucial to ensuring that a project adheres to legal standards and respects the intellectual property rights of others. Licenses can include a range of categories such as open source, proprietary, shareware, and freeware, each with its own set of permissions and restrictions.

6.1.2 Example from the Public Domain

One notable example from the public domain that underscores the importance of licensing issues is the case of Oracle America, Inc. v. Google Inc. This legal battle revolved around Google's use of Java APIs in its Android operating system without obtaining the appropriate license from Oracle. The case highlighted the potential repercussions of neglecting licensing requirements, including costly lawsuits and the risk of having to alter or even cease the use of critical software components [13].

6.1.3 Ethical Considerations in My Project

Given that my game is developed using Godot 4, an open-source engine, it is imperative to comply with the license under which Godot is distributed, which is the MIT License. This license allows for the free use, modification, and distribution of the software, provided that the original license and copyright notice are included in any distributed copies or substantial portions of the software.

Additionally, other third-party assets such as textures, models, and sound effects may also be used in the game. It is essential to ensure that these assets are correctly licensed and that their use complies with the respective licenses. This can include:

- **Open Source Assets**: Verifying that assets labelled as open source are indeed free to use, modify, and distribute under the terms of their licenses, such as Creative Commons or GNU General Public License (GPL).
- **Commercial Assets:** Ensuring that any purchased assets are used in accordance with their commercial licenses, which often include restrictions on redistribution and modification.
- **Attribution**: Properly crediting the creators of third-party assets in the game's documentation and within the game itself, as required by the terms of many open-source and Creative Commons licenses [14].

6.1.4 Reflective and Practical Importance

The ethical and practical importance of addressing licensing issues in game development is multifaceted. Ethically, it is about respecting the intellectual property rights of others and ensuring that creators are duly credited and compensated for their work. Failure to adhere to licensing agreements can result in legal repercussions, damage to reputation, and financial losses.

Practically, proper licensing management can facilitate smoother project development and distribution. Understanding and adhering to licenses ensures that the game can be legally distributed and monetised, preventing potential disputes and takedowns. It also builds trust and credibility within

the developer community and with players, who increasingly value transparency and respect for intellectual property.

6.2 Accessibility and Inclusivity

In the development of my game, it is crucial to address the professional considerations and ethical issues that arise. Below I focus on usability, particularly accessibility, and the ethical importance of ensuring that video games are inclusive and accessible to a diverse audience.

6.2.1 Usability and Accessibility

Usability and accessibility are paramount in creating video games that cater to all users, including those with disabilities. Accessibility in games involves designing interfaces and mechanics that are usable by people with a wide range of abilities and disabilities. This can include visual, auditory, motor, and cognitive impairments. Ensuring that a game is accessible not only broadens its potential audience but also aligns with ethical standards of inclusivity and equality.

6.2.2 Example from the Public Domain

One notable example from the public domain that highlights the importance of accessibility in games is the case of "The Last of Us Part II" by Naughty Dog. This game received widespread acclaim for its comprehensive accessibility features, which included options for visual and auditory aids, customisable controls, and alternative gameplay modes to cater to players with different needs [15]. The game's success and positive reception from the community underscore the importance of considering accessibility in game design.

6.2.3 Ethical Considerations in My Project

Ethical concerns in usability extend to accessibility for players with disabilities. While I have incorporated straightforward mechanics, I am aware that the game currently lacks features such as customisable controls or visual aids for colourblind players. This shortcoming reflects a broader ethical issue in game design—ensuring inclusivity for all users. In future iterations, implementing these features will be a priority, aligning the project with ethical standards like those outlined by the International Game Developers Association's (IGDA) guidelines on accessibility [16].

In designing my game's UI, I attempted to address some accessibility considerations:

1. **Clear Visual Hierarchy**: The UI elements in the battle and dialogue systems are arranged with a clear visual hierarchy to improve readability.

gdscript

```
# In battle_ui.gd
func update_health(player_hp, player_max_hp, enemy_hp, enemy_max_hp):
    player_health_bar.max_value = player_max_hp
    player_health_bar.value = round(player_hp)
    player_health_label.text = str(round(player_hp)) + "/" + str(player_max_hp)

    var player_name = "Player"
    if Global.player_name != null and Global.player_name.length() > 0:
        player_name = Global.player_name
    $PlayerPanel/PlayerName.text = player_name

    enemy_health_bar.max_value = enemy_max_hp
    enemy_health_bar.value = enemy_hp
```

```
14.     enemy_health_label.text = str(enemy_hp) + "/" +
str(enemy_max_hp)
```

2. **Text Scaling:** Text elements are sized appropriately to ensure readability.

3. **Input Simplicity**: The game's control scheme is deliberately simple, using only the arrow keys and spacebar for most interactions.

However, there are significant areas for improvement:

1. **Color-blind Modes:** The game does not currently offer alternative color schemes for players with color vision deficiencies.
2. **Configurable Controls:** The control scheme is fixed, limiting accessibility for players with motor impairments.
3. **Screen Reader Support:** The game lacks comprehensive support for screen readers, limiting accessibility for visually impaired players.
4. **Text-to-Speech:** Dialogue is not available in audio format for players who have difficulty reading.

Addressing these issues requires thoughtful planning, resource allocation, and an ongoing commitment to refining and improving the game's accessibility features.

6.2.4 Reflective and Practical Importance

From an ethical standpoint, failing to include accessibility features can alienate a significant portion of potential players, perpetuating inequalities and limiting the enjoyment and engagement of diverse audiences. This goes against the principles of fairness and inclusivity that are integral to ethical computing and software development.

Practically, incorporating accessibility features can enhance the reputation of the game and its developers, leading to broader market appeal and increased player satisfaction. It also helps in complying with legal standards and industry guidelines, reducing the risk of legal challenges and negative publicity.

Moreover, the process of integrating accessibility features forms a deeper understanding and empathy among developers for the diverse needs of players. It encourages a user-centred design approach, where feedback from players with disabilities is actively sought and incorporated into the development process, leading to better overall game design.

6.3 Representation in Gaming

An essential professional consideration in game development is the representation of diverse characters and experiences. This issue is particularly relevant to my project, which aims to create a game specifically appealing to female audiences within a genre traditionally dominated by male-oriented content.

6.3.1 The Importance of Diverse Representation

The gaming industry has historically underrepresented certain demographics, particularly women and minorities. This lack of representation can alienate potential players and perpetuate stereotypes. Studies have shown that inclusive games can reach broader audiences and create more engaging experiences for all players [28].

6.3.2 My Approach to Representation

In developing "Just a Girl's Dungeon Quest," I made deliberate choices to address representation issues:

1. **Protagonist Design:** The game features a female protagonist whose journey centers around self-discovery and competence rather than romantic pursuits or rescue scenarios. This decision challenges genre conventions while providing representation that many female players might find relatable.

2. **Narrative Focus:** The story emphasises the protagonist's agency and growth, with dialogue that avoids gender stereotypes. For example, in the scene_1.gd implementation:

gdscript

```
1. var dialogues = [
2.     {"character": "King", "text": "My dear daughter, it's
time you found yourself a suitable prince to marry.",
"expression": "neutral", "speakers": ["King"]},
3.     {"character": "[name]", "text": "But father, all these
princes are so boring! This one only talks about his wealth,
that one just boasts about his horses...", "expression": "sad",
"speakers": ["Princess"]},
4.     // Later dialogue shows character growth and independence
5.     {"character": "[name]", "text": "I'll show him. I'll
become the greatest adventurer this kingdom has ever known!",
"expression": "sad", "speakers": ["Princess"]},
6. ]
```

   This narrative arc subverts traditional princess tropes by placing the character in an active role seeking adventure rather than passively accepting societal expectations.

3. **Character Development:** The protagonist's journey from sheltered princess to capable adventurer demonstrates growth and agency, challenging stereotypes about female characters in games.

6.3.3 Ethical Considerations and Challenges

While striving for better representation, I encountered several challenges:

1. **Avoiding New Stereotypes:** There's a risk that in attempting to create a "female-friendly" game, new stereotypes might emerge. I aimed to create a balanced character with strengths and weaknesses rather than an idealised representation.

2. **Authentic Portrayal:** As a developer, ensuring authentic representation requires research, feedback, and self-awareness about potential biases.

3. **Balancing Universal Appeal with Specific Representation:** While targeting female players, I wanted to create a game that remains appealing to all players. This required careful consideration of design elements that might unnecessarily gender the experience.

6.3.4 Future Improvements

In future iterations, several improvements could further enhance representation:

1. **Diverse Character Options:** Including character customization options that allow players to choose from a range of appearances and backgrounds.

2. **Inclusive Narrative Branches:** Expanding the dialogue system to accommodate different player choices that reflect diverse play styles and preferences.

3. **Community Feedback:** Establishing channels for feedback from diverse players to inform ongoing development and improvements.

By addressing representation thoughtfully, games can become more inclusive spaces that reflect and celebrate the diversity of their player base, ultimately creating richer and more engaging experiences for everyone.

Chapter 7: Evaluation and Critical Analysis

7.1 Project Achievements

This section evaluates the project's achievements against its original goals and objectives, highlighting both successes and areas for improvement.

7.1.1 Technical Achievements

The project successfully implemented several complex technical systems:

1. **Procedural Dungeon Generation:** The random walk algorithm consistently produces navigable, interesting dungeon layouts with increasing complexity as players progress. The system properly scales difficulty by adjusting dungeon size, enemy count, and hazard frequency.

2. **Turn-Based Combat System**: The battle system provides strategic depth with multiple abilities, enemy types with distinct behaviours, and a progression system that rewards successful play with new capabilities.

3. **Grid-Based Movement:** The movement system effectively translates player input into intuitive navigation within the 3D grid space, with collision detection that prevents invalid movements.

4. **Save/Load System:** The persistence system successfully saves and loads all relevant game state, allowing players to continue their progress across sessions.

5. **Narrative Integration:** The dialogue system seamlessly integrates narrative elements with gameplay mechanics, providing context and motivation for dungeon exploration.

7.1.2 Comparison with Original Goals

Revisiting the original project aims:

1. **Develop a roguelike dungeon crawler for female audiences:** The project successfully implements core roguelike elements (procedural generation, combat, progression) while incorporating a narrative focus and aesthetic choices that may appeal to underserved audiences.

2. **Create procedurally generated dungeons with increasing complexity:** Fully achieved, with a scaling system that adjusts difficulty parameters based on floor number.

3. **Implement turn-based combat with strategic depth:** Achieved through varied enemy behaviours, multiple player abilities, and meaningful progression.

4. **Design an intuitive movement system:** Successfully implemented, with smooth animations and clear feedback for valid/invalid movements.

5. **Integrate narrative elements:** Successfully implemented through dialogue scenes and character interactions that motivate gameplay.

6. **Implement save/load functionality:** Fully implemented, allowing game state persistence across sessions.

7.2 Technical Limitations and Challenges

Despite meeting most of the original goals, several technical limitations and challenges emerged during development:

7.2.1 Performance Optimisation

While the game runs smoothly on most systems, larger dungeon floors with numerous enemies can cause performance issues, particularly during generation. The random walk algorithm, while effective for creating interesting layouts, is not optimised for large-scale generation. Future improvements could include:

1. **Chunked Generation:** Breaking the dungeon generation into smaller chunks that can be processed independently.

2. **Level of Detail Systems:** Implementing LOD for distant elements to reduce rendering overhead.

3. **Optimised Pathfinding**: The current enemy movement system uses basic pathfinding that could be improved with more efficient algorithms like A*.

7.2.2 Enemy AI Limitations

The enemy AI, while functional, has limited behavioral complexity:

gdscript

```
 1. func move_towards_player():
 2.     if player == null or accessible_cells.size() == 0 or
in_battle:
 3.         return
 4.
 5.     if check_for_battle():
 6.         return
 7.
 8.     var player_cell = Vector2i(
 9.         floor(player.global_transform.origin.x / GRID_SIZE),
10.         floor(player.global_transform.origin.z / GRID_SIZE)
11.     )
12.
13.     // Basic pathfinding logic
```

This implementation focuses on direct movement toward the player without considering tactical positioning, group coordination, or environmental factors. More sophisticated AI could enhance the gameplay experience.

7.2.3 UI Scalability

The current UI implementation has limited scalability for different screen resolutions and aspect ratios. Elements are positioned absolutely rather than using responsive design principles, which may cause issues on certain devices.

7.3 User Feedback and Iteration

Based on informal playtesting with five Computer Science students and three non-technical players, several iterations were made to address user concerns:

- **Movement Responsiveness:** Early feedback indicated that the movement felt sluggish, with users frequently commenting that their inputs seemed delayed. This led to adjustments in the

movement speed and cooldown timers, reducing the base cooldown from 0.5 to 0.2 seconds and increasing overall movement speed by approximately 25%.

- **Combat Clarity:** Users initially found the combat system confusing, with one playtester stating they "couldn't tell when it was their turn or what their moves actually did." In response, I improved the battle UI with clearer turn indicators, animated effects for attacks and healing, and descriptive tooltips for player abilities that appear when hovering over move buttons.

- **Difficulty Balancing:** The scaling difficulty system was adjusted multiple times based on user feedback. First-time players found the second floor suddenly too challenging while experienced gamers found later floors too easy. This led to a more gradual difficulty curve where enemy count and strength increase by smaller increments between floors, and more aggressive scaling for floors beyond level 3.

- **Tutorial Elements**: Non-technical players consistently struggled with understanding core mechanics. A tutorial system was implemented that introduces game elements progressively, with one tester commenting that "having the instructions appear as I encountered new features made everything much clearer."

- **Narrative Engagement:** Several players expressed interest in more story elements, with one noting that "I wanted to know more about why my character was in the dungeon." This led to additional dialogue sequences and environmental storytelling elements that provide context without interrupting gameplay flow.

This iterative approach to development allowed the game to evolve significantly from its initial concept, with each testing cycle revealing new opportunities for improvement and refinement.

7.4 Learning Outcomes

The development of this project provided significant learning opportunities:

1. **Godot Engine Proficiency:** Developing a complex game in Godot 4 required mastering its node-based architecture, signal system, and GDScript language.

2. **Procedural Content Generation:** Implementing and refining the random walk algorithm deepened my understanding of procedural generation techniques and their applications.

3. **Game Design Patterns:** Applying patterns like Factory, State, and Observer improved code organization and maintainability.

4. **User Experience Design:** Balancing technical features with user experience considerations highlighted the importance of design thinking alongside technical implementation.

7.5 Future Enhancements

Several potential enhancements could improve the game in future iterations:

1. **Enhanced Procedural Generation:** Implementing more sophisticated algorithms like Binary Space Partitioning or Wave Function Collapse could create more varied and interesting dungeon layouts.

2. **Advanced Enemy AI:** Developing more complex enemy behaviour patterns, including group tactics and environmental awareness, would create more engaging challenges.

3. **Expanded Narrative:** Adding more story elements, character development, and narrative branches would enhance player engagement and provide greater context for exploration.

4. **Accessibility Features:** Implementing comprehensive accessibility options, including color-blind modes, configurable controls, and text-to-speech functionality.

5. **Enhanced Visual Effects:** Adding particle systems, shader effects, and more sophisticated animations would improve the visual appeal and feedback clarity.

6. **Mobile Adaptation:** Adapting the control scheme and UI for mobile platforms would expand the potential audience.

7.6 Critical Reflection

Reflecting on the project as a whole, several insights emerge:

1. **Scope Management:** The project's scope expanded significantly during development, particularly in narrative and combat complexity. While this resulted in a more feature-rich game, it also created challenges in meeting deadlines and ensuring thorough testing of all features.

2. **Technical Debt:** Rapid implementation of features occasionally led to technical debt that required later refactoring. A more disciplined approach to architecture and testing could have reduced this issue.

3. **Design Priorities:** Balancing technical implementation with design considerations was an ongoing challenge. Future projects would benefit from more explicit design documentation and user testing throughout development.

4. **Learning Curve:** The learning curve of Godot 4, while manageable, slowed initial development. However, this investment in learning a new engine ultimately provided valuable skills and perspective.

Despite these challenges, the project successfully achieved its core goals, creating a playable, engaging roguelike experience with unique narrative elements and accessible gameplay. The game demonstrates technical competence across multiple game development domains and provides a foundation for future expansion and refinement.

Chapter 8: Conclusion

8.1 Summary of Achievements

This project set out to create a procedurally generated dungeon crawler with a narrative focus and gameplay mechanics designed to appeal to female players and others who might find traditional roguelikes inaccessible. Through careful implementation of various systems—procedural generation, grid-based movement, turn-based combat, and narrative integration—the project has largely achieved these goals.

Key technical achievements include:

1. A robust procedural dungeon generation system using a random walk algorithm to create varied and navigable layouts
2. A grid-based movement system with intuitive controls and smooth animations
3. A turn-based combat system featuring strategic depth and meaningful character progression
4. A dialogue system that provides narrative context and character development
5. A comprehensive save/load system for persistent gameplay

The game successfully implements core roguelike elements while incorporating narrative and aesthetic choices that may appeal to underserved audiences. Its modular architecture facilitates future expansion and refinement, allowing for ongoing development beyond this initial version.

8.2 Lessons Learned

The development process offered numerous valuable lessons in game development:

8.2.1 Technical Lessons

1. **Framework Selection:** Godot 4 proved to be an effective choice for this project, offering a balance of flexibility and ease of use. Its node-based architecture facilitated the creation of modular, reusable components.

2. **Design Patterns:** The application of design patterns like Factory, State, and Observer significantly improved code organization and maintainability. These patterns provided structured solutions to common game development problems.

3. **Performance Optimisation:** Early attention to performance optimisation is crucial, particularly for procedural generation systems that can become computationally expensive as complexity increases.

8.2.2 Project Management Lessons

1. **Scope Management:** Managing feature creep remains challenging. Future projects would benefit from more explicit prioritization of features and regular scope reviews.

2. **Iterative Development:** The iterative approach to development, with functional milestones rather than time-based sprints, proved effective for accommodating learning curves and adjusting to discoveries during implementation.

3. **Documentation Importance:** Comprehensive documentation of systems and design decisions facilitates maintenance and future expansion. This project would have benefited from more thorough documentation throughout development.

8.2.3 Design Lessons

1. **User Feedback:** Early and frequent user testing is invaluable for identifying usability issues and refining game feel.

2. **Accessibility Considerations:** Accessibility should be considered from the outset rather than as an afterthought. This project highlighted the need for a more comprehensive approach to accessibility in future work.

3. **Balancing Complexity:** Finding the right balance between depth and accessibility requires ongoing refinement and user feedback.

8.3 Future Work

While the current version of the game achieves its core objectives, several avenues for future development remain:

1. **Enhanced Procedural Generation:** Implementing additional generation algorithms could create more varied dungeon layouts, potentially including different biomes or architectural styles.

2. **Expanded Narrative:** Additional dialogue, character interactions, and narrative branches would enhance the storytelling aspects that distinguish this game from traditional roguelikes.

3. **Advanced Enemy Types:** Developing more sophisticated enemy behaviours and introducing new enemy types would increase gameplay variety and challenge.

4. **Comprehensive Accessibility Features:** Implementing color-blind modes, configurable controls, and other accessibility options would make the game more inclusive.

5. **Mobile Adaptation:** Adapting the control scheme and UI for mobile platforms would expand the potential audience.

6. **Multiplayer Functionality:** Adding cooperative or competitive multiplayer modes could create new gameplay experiences.

8.4 Personal Reflection

This project represents a significant milestone in my development as a game developer. By tackling complex systems like procedural generation and turn-based combat, I've expanded my technical skills and deepened my understanding of game design principles.

The process of creating a game that addresses representation issues in the roguelike genre has also broadened my perspective on the social and ethical dimensions of game development. Understanding how design choices can influence inclusivity and accessibility has become a central consideration in my approach to creating games.

Perhaps most importantly, this project has demonstrated that technical implementation and meaningful design can work hand in hand. A game can incorporate sophisticated systems while still remaining accessible and engaging to a diverse audience. This balance between technical achievement and user experience will remain a guiding principle in my future work.

Bibliography

[1] E. Adams and J. Dormans, Game Mechanics: Advanced Game Design, Berkeley, California: New Riders, 2012.

Comment: This book provided foundational knowledge on designing engaging game mechanics, which informed my approach to balancing procedural generation with accessible gameplay.

[2] S. Rabin, Game AI Pro 2: Collected Wisdom of Game AI Professionals, 1st ed., S. Rabin, Ed., A K Peters/CRC Press, 2015.

Comment: This resource was essential for planning future AI implementation, particularly finite state machines and pathfinding strategies for enemy behaviour.

[3] K. Perlin, "An image synthesizer," ACM SIGGRAPH Computer Graphics, vol. 19, no. 3, pp. 287-296, 1985.

Comment: Perlin's work on noise algorithms influenced early considerations for procedural texture generation, although ultimately the dungeon generation relied on grid-based techniques.

[4] W. contributers, "Elite (video game)," Wikipedia, The Free Encyclopedia., 6 December 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Elite_(video_game)&oldid=1261448304. [Accessed 11 December 2024].

Comment: This article helped contextualise procedural generation as a cornerstone of game design, with insights into its historical applications in early games like Elite.

[5] M. J. Nelson, J. Togelius and N. Shaker, "Constructive generation methods for dungeons and levels," in Procedural Content Generation in Games: A Textbook and an Overview of Current Research, N. Shaker, J. Togelius and M. J. Nelson, Eds., Springer, 2016, pp. 31-55.

Comment: This text was instrumental in understanding various algorithms for dungeon generation, including random walk and constructive methods, which were adapted for my game.

[6] G. Smith, J. Whitehead and M. Mateas, "Tanagra: A mixed-initiative level design tool," Expressive Intelligence Studio, Santa Cruz, California, 2010.

Comment: This paper inspired considerations for balancing algorithmic generation with creative control, ensuring that generated dungeons maintain logical flow and engagement.

[7] RogueBasin, "What is a Roguelike?," 2023. [Online]. Available: https://roguebasin.com/index.php/Main_Page. [Accessed 5 September 2024].

Comment: This resource provided a clear definition of rogue-like games, shaping the core gameplay elements like procedural generation and grid-based navigation.

[8] J. Togelius, G. N. Yannakakis, K. O. Stanley and B. Cameron, "Search-Based Procedural Content Generation: A Taxonomy and Survey," IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, pp. 172-186, 2011.

Comment: This paper informed my understanding of procedural content generation and the taxonomy of techniques, which guided the implementation of my dungeon generation system.

[9] Heartbeast, 3D Dungeon Code Walkthrough in Under 15 Minutes - Godot 3.4, YouTube, 2021.

Comment: This video tutorial offered practical insights into creating 3D dungeon mechanics, which influenced early prototypes of the dungeon scene.

[10] Piotr, "DungeonCrawler: A Godot 4 Asset for Basic Dungeon Crawler Mechanics," GitHub, 20 April 2024. [Online]. Available: https://github.com/Rebelion-Board-game/DungonCrawler. [Accessed 10 October 2024].

Comment: This GitHub repository was directly utilised as a starting point for grid-based movement and basic dungeon crawler mechanics.

[11] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Pattern Catalog," in Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994, pp. 79-351.

Comment: This book provided a comprehensive overview of design patterns, such as Factory and State patterns, which were applied to procedural generation and AI planning.

[12] R. Nystrom, Game Programming Patterns, 1st ed., Genever Benning, 2014.

Comment: Nystrom's work informed my use of design patterns like Game Loop and State Machine, ensuring a modular and scalable codebase.

[13] N. Woolf, "Google wins six-year legal battle with Oracle over Android code copyright," The Guardian, San Francisco, 2016.

Comment: This article contextualised the importance of respecting intellectual property, influencing my adherence to licensing and attribution practices for assets and code.

[14] Creative Commons, "About CC Licenses," Creative Commons, [Online]. Available: https://creativecommons.org/share-your-work/cclicenses/.

Comment: This resource guided my use of open-source assets, ensuring proper attribution and compliance with Creative Commons licenses.

[15] M. Gallant, "The Last of Us Part II: Accessibility Features Detailed," Naughty Dog, 9 June 2020. [Online]. Available: https://www.naughtydog.com/blog/the_last_of_us_part_ii_accessibility_features_detailed. [Accessed 5 December 2024].

Comment: This blog post inspired ideas for accessibility features, such as intuitive controls and UI design, although implementation is planned for future iterations.

[16] International Game Developers Association, "SIG guidelines," idga-gasig, 2024. [Online]. Available: https://igda-gasig.org/get-involved/sig-initiatives/resources-for-game-developers/sig-guidelines/.

Comment: The IGDA guidelines influenced my focus on inclusive game design and accessibility, ensuring the project aligns with industry best practices.

[17] I. Millington and J. Funge, "Pathfinding A*," in Artificial Intelligence For Games – 2nd ed., Morgan Kaufmann, 2009, pp. 215-237.

Comment: This section provided theoretical grounding for implementing A pathfinding, which will be used in the future for enemy AI.*

[18] I. Millington and J. Funge, "Finite State Machines," in Artificial Intelligence For Games – 2nd ed., Morgan Kaufmann, 2009, pp. 310-315.

Comment: This chapter served as a key reference for designing state-based AI systems, planned for implementation in enemy behaviours.

[19] Hello Games, "No Man's Sky: Origins," 2020. [Online]. Available: https://www.nomanssky.com/origins-update/. [Accessed 15 March 2025].

Comment: This game demonstrates advanced procedural generation techniques that generate entire planets and ecosystems.

[20] Mojang Studios, "Minecraft," 2011. [Online]. Available: https://www.minecraft.net/. [Accessed 18 March 2025].

Comment: Minecraft's procedural world generation provided inspiration for creating diverse environments.

[21] E. McMillen, "Design Postmortem: The Binding of Isaac," Gamasutra, 2012. [Online]. Available: https://www.gamasutra.com/view/feature/182380/. [Accessed 20 March 2025].

Comment: McMillen's insights into roguelike design influenced my approach to procedural content generation and difficulty scaling.

[22] G. Kasavin, "Building the Cyclical Narrative of Hades," GDC, 2021. [Online]. Available: https://www.gdcvault.com/play/1027107/. [Accessed 22 March 2025].

Comment: Supergiant's approach to blending narrative with roguelike elements in Hades influenced my narrative integration strategy.

[23] Godot Engine Documentation, "TileMap," 2024. [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_tilemap.html. [Accessed 15 February 2025].

Comment: The official documentation for Godot's TileMap system was essential for implementing the grid-based dungeon generation.

[24] M. Toy and G. Wichman, "Rogue: Exploring the Dungeons of Doom," Communications of the ACM, vol. 26, no. 1, pp. 21-29, 1983.

Comment: This seminal paper on the original Rogue game provided historical context for the roguelike genre.

[25] J. Johnson, "Cellular Automata for Real-Time Generation of Infinite Cave Levels," PCG Workshop, 2010. [Online]. Available: http://pcgworkshop.com/proceedings.html. [Accessed 5 January 2025].

Comment: Johnson's paper on using cellular automata for cave generation offered an alternative approach to dungeon creation that influenced some aspects of my implementation.

[26] M. Gumin, "WaveFunctionCollapse," GitHub, 2016. [Online]. Available: https://github.com/mxgmn/WaveFunctionCollapse. [Accessed 10 January 2025].

Comment: This repository demonstrates the Wave Function Collapse algorithm, which represents a promising future direction for more complex procedural generation.

[27] A. Summerville et al., "Procedural Content Generation via Machine Learning (PCGML)," IEEE Transactions on Games, vol. 10, no. 3, pp. 257-270, 2018.

Comment: This survey paper provides an overview of machine learning approaches to PCG, which could be applied in future versions of the game.

[28] J. E. Mendez et al., "Diversity in games: An analysis of representation and player responses," Games and Culture, vol. 15, no. 4, pp. 411-432, 2020.

Comment: This research paper examines the impact of diverse representation in games, informing my approach to character design and narrative.

[29] Claude.ai, Generative AI

Comment: I used generative AI to debug sections of code, specifically in struggling areas such as the movement bug.

Appendix A: User Manual & Installation Guide

The User_Manual.pdf can be found in the documents directory. It contains information about the game and how to set it up. Below are screenshots of the document.

Just a Girl's Dungeon Quest

User Manual & Installation Guide

Game Information

Developer: Salma Bocus
Project Type: Final Year Project – Building a Game
Game Version: 1.0

Table of Contents

Game Overview

"Just a Girl's Dungeon Quest" is a story-driven RPG adventure following a princess who defies expectations and embarks on a dungeon-crawling quest to prove herself. Navigate through dungeons, battle enemies, and experience an engaging narrative.

This game was developed as a Final Year Project by Salma Bocus (100960403)

Installation Guide

Windows Installation

Option 1: Using the Installer (Recommended)

1. Unzip the file `Windows_Installer.zip` found in the **Release** folder
2. Run the executable installer
3. Follow the on-screen prompts to complete installation
    - Note: Administrative privileges are not required
    - Start menu and desktop shortcuts will be created automatically

To Uninstall:

- Either uninstall through **Windows Settings > Apps & Features**
- Or run the `uninstall.exe` file located in the installation directory

Option 2: Direct Download

1. Visit https://naylith.itch.io/just-a-girls-dungeon-quest
2. Enter the password: `salma-fyp`
3. Download the Windows version
4. Extract the files and run the game executable

Linux Installation

1. Unzip the file `Linux_Installer.zip` found in the **Release** folder
2. Navigate to the extracted directory
3. Run the executable: `./JustAGirlsDungeonQuest.x86_64`

Web Version

The game can also be played directly in your web browser:

1. Visit https://naylith.itch.io/just-a-girls-dungeon-quest
2. Enter the password: `salma-fyp`
3. Press “Run Game"
4. Click the fullscreen icon in the bottom right of the game screen.

Note: The web version may experience performance issues. For the best experience, it is recommended to install the game locally on your computer.

Controls

The game has simple controls designed for accessibility:

- **Arrow Keys:** Navigate menus and move your character in the dungeon
- **Spacebar/Enter:** Select options, advance dialogue etc.
- **Mouse:** Required for save/load menu navigation

All other areas of the game can be navigated using only the keyboard if desired.

Known Issues

- **Performance:** Navigating the dungeon can be resource-intensive and may cause lag or crashes on some systems
- **Web Version:** The browser-based version experiences significant lag compared to the installed versions
- **Save/Load Interface:** Keyboard controls are not fully implemented for the save/load menus; mouse interaction is required
- **Audio:** There is currently no in-game music

For the best experience:

- Install the game on your computer rather than playing the web version
- Ensure your computer meets the minimum requirements for running Godot-based games

Additional Information

For Marking Purposes

- Exported files can be found on GitLab under `product/Release`
- To view the **source code**:
    - Unzip the folder in `Source/Just_A_Girls_Dungeon_Quest_Godot_4.zip`
    - Or access them on my GitLab's main branch under `/product/Godot`

Disclaimers

- This game represents a prologue to what could become a much larger game concept.
- This version is created specifically as a Final Year Project submission and is not intended for commercial distribution.
- The game uses copyrighted assets (sprites and artwork) from the Spriters Resource as placeholders.
- Subsequently, if the game were to be distributed commercially or made public, all copyrighted assets would of course be replaced.
- The storyline is an initial draft, with the primary focus having been on programming and game development aspects.

Demo Video

Click here or copy and paste this link: https://youtu.be/W10s7Od2_3M

Thank you for playing "Just a Girl's Dungeon Quest"!

Salma Bocus

Appendix B: Key Code Implementations

C.1 Random Walk Algorithm

gdscript

```gdscript
func _generate_random_map() -> Array:
    var map = []
    for y in range(MAP_SIZE.y):
        var row = []
        for x in range(MAP_SIZE.x):
            row.append(0)
        map.append(row)

    var gen_position = Vector2(floor(MAP_SIZE.x / 2),
floor(MAP_SIZE.y / 2))
    map[gen_position.y][gen_position.x] = 1
    accessible_cells[Vector2i(gen_position.x, gen_position.y)] =
true
    for i in range(MAX_STEPS):
        var direction = randi() % 4
        if direction == 0: gen_position.y -= 1
        elif direction == 1: gen_position.y += 1
        elif direction == 2: gen_position.x -= 1
        elif direction == 3: gen_position.x += 1

        gen_position.x = clamp(gen_position.x, 0, MAP_SIZE.x - 1)
        gen_position.y = clamp(gen_position.y, 0, MAP_SIZE.y - 1)

        map[gen_position.y][gen_position.x] = 1
        accessible_cells[Vector2i(gen_position.x,
gen_position.y)] = true

    return map
```

C.2 Battle State Machine

gdscript

```gdscript
enum BattleState {INACTIVE, PLAYER_TURN, ENEMY_TURN, ANIMATING}

func _on_move_selected(move_name):
    if current_state != BattleState.PLAYER_TURN:
        return

    print("Battle Manager: Player used ", move_name)
    current_state = BattleState.ANIMATING

    match move_name:
        "Tackle":
            _player_attack(15)
            await get_tree().create_timer(1.0).timeout
        "Defend":
            player_stats.defense += 5

            if battle_ui:
```

```
                battle_ui.display_message("You defended!")
                await get_tree().create_timer(1.5).timeout
                battle_ui.display_message("Your defense rose!")
            await get_tree().create_timer(1.0).timeout

        "Heal":
            var min_heal = ceil(player_stats.max_hp * 0.1)
            var max_heal = ceil(player_stats.max_hp * 0.5)
            var heal_amount = randi_range(min_heal, max_heal)
            _player_heal(heal_amount)
            if battle_ui:
                battle_ui.display_message("You healed " +
str(heal_amount) + " health!")
            await get_tree().create_timer(1.0).timeout

        "Flee":
            _attempt_flee()
            return
        "Firebolt":
            _player_attack(25)
        "Earthquake":
            _player_attack(35)
        "Ultimate Strike":
            _player_attack(50)
        _:
            _player_attack(player_stats.attack)

    if enemy_stats.hp <= 0:
        battle_ui.display_message("Enemy defeated!")
        get_tree().create_timer(1.0).timeout.connect(func():
end_battle(current_player))
    else:
        if current_state != BattleState.INACTIVE:
            current_state = BattleState.ENEMY_TURN
            _update_battle_ui()
            _enemy_turn()
```

C.3 Save System Implementation

gdscript

```
func save_game(slot_number: int) -> bool:
    var save_path = get_save_path(slot_number)

    var save_data = {
        "version": "1.0",
        "timestamp": Time.get_unix_time_from_system(),
        "player_name": Global.player_name,
        "player_stats": {
            "max_hp": Global.player_max_hp,
            "current_hp": Global.player_current_hp,
"attack": Global.player_attack,
"defense": Global.player_defense,
"level": Global.player_level,
"experience": Global.player_experience,
"experience_to_next_level":
Global.player_experience_to_next_level,
"moves": Global.player_moves
```

C.4 Enemy AI Implementation

gdscript

```gdscript
func move_towards_player():
    if player == null or accessible_cells.size() == 0 or in_battle:
        return

    if check_for_battle():
        return

    var player_cell = Vector2i(
        floor(player.global_transform.origin.x / GRID_SIZE),
        floor(player.global_transform.origin.z / GRID_SIZE)
    )

    print("Enemy at cell: ", current_cell, " Player at cell: ", player_cell)

    // Get possible moves (adjacent accessible cells)
    var possible_moves = []
    var directions = [
        Vector2i(1, 0),   // Right
        Vector2i(-1, 0),  // Left
        Vector2i(0, 1),   // Down
        Vector2i(0, -1)   // Up
    ]

    for dir in directions:
        var target_cell = current_cell + dir
        if accessible_cells.has(target_cell):
            possible_moves.append(target_cell)

    if possible_moves.size() == 0:
        print("Enemy has no possible moves")
        return

    // Choose the move that gets closest to player
    var best_move = current_cell
    var best_distance = 999999

    for move in possible_moves:
        var distance = abs(move.x - player_cell.x) + abs(move.y - player_cell.y)
        if distance < best_distance:
            best_distance = distance
            best_move = move

    // Move to the best cell
    if best_move != current_cell:
        print("Enemy moving from ", current_cell, " to ", best_move)
        current_cell = best_move
        target_position = Vector3(best_move.x * GRID_SIZE, 0.1, best_move.y * GRID_SIZE)
        is_moving = true
```

```
49.     else:
50.         print("Enemy staying at current cell")
```

C.5 Dialogue System

gdscript

```
 1. func _update_displayed_text():
 2.     dialogue_label.clear()
 3.
 4.     var name_color = CHARACTER_NAME_COLOR if current_character !=
"Narrator" else NARRATOR_COLOR
 5.     dialogue_label.push_color(Color(name_color))
 6.     dialogue_label.add_text(current_character)
 7.     dialogue_label.pop()
 8.     dialogue_label.newline()
 9.
10.     var text_to_display = displayed_text
11.     if current_character != "Narrator":
12.         text_to_display = "\"" + text_to_display + "\""
13.
14.     var final_text = text_to_display
15.     for keyword in keywords:
16.         var keyword_regex = RegEx.new()
17.         keyword_regex.compile("(?i)" + keyword)
18.
19.         var matches = keyword_regex.search_all(final_text)
20.         if matches.size() > 0:
21.             for i in range(matches.size() - 1, -1, -1):
22.                 var match_obj = matches[i]
23.                 var start = match_obj.get_start()
24.                 var end = match_obj.get_end()
25.                 var matched_text = match_obj.get_string()
26.
27.                 final_text = final_text.substr(0, start) +
"[color=" + KEYWORD_COLOR + "]" + matched_text + "[/color]" +
final_text.substr(end)
28.
29.     dialogue_label.append_text(final_text)
```

Appendix D: Game Flow Diagram

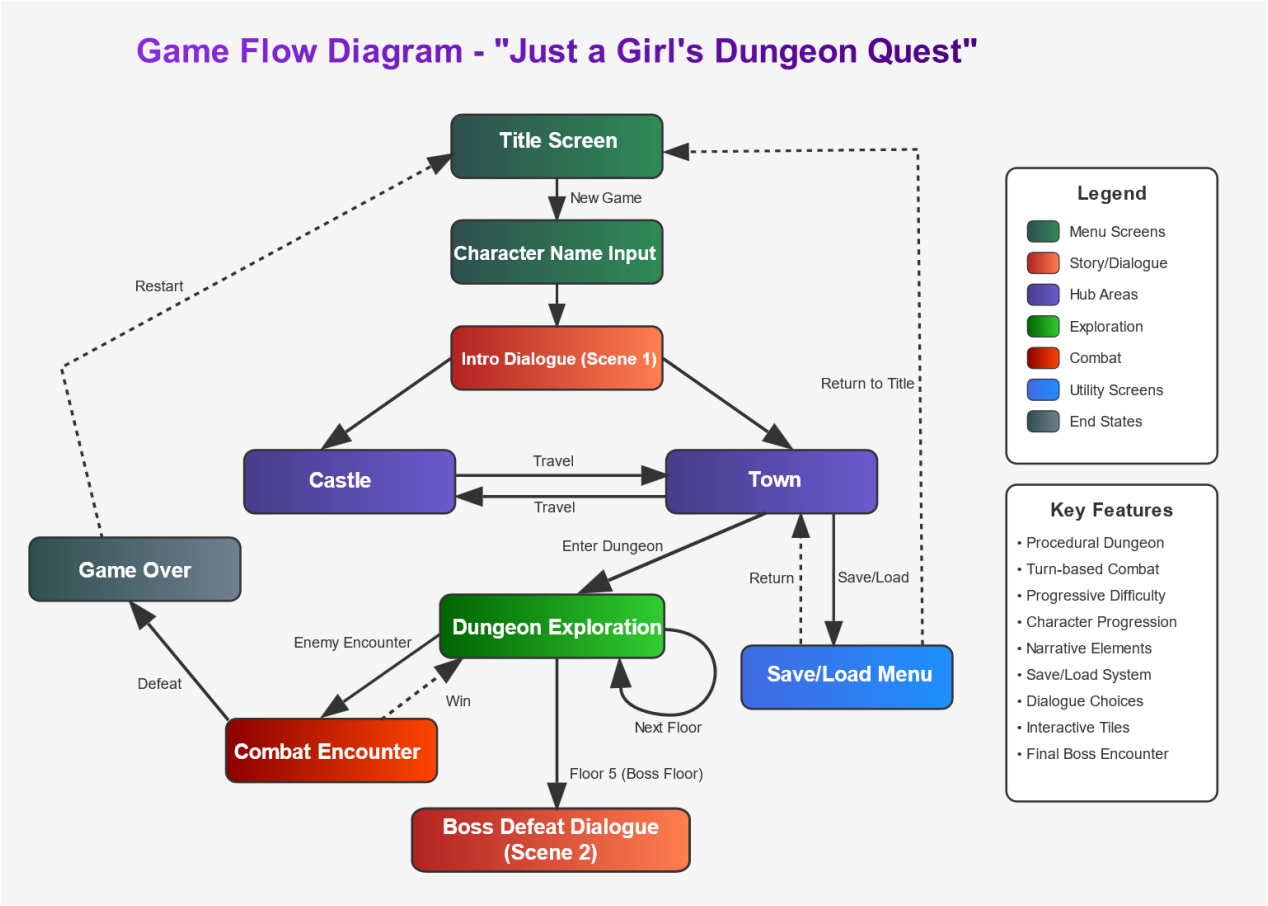

Figure 7 Game Flow Diagram

This diagram illustrates the complete player journey through the game, showing all possible paths and transitions between different game states. The color-coded nodes represent different types of screens or gameplay moments, as indicated in the legend.

The flow begins at the Title Screen, where players can either start a new game (proceeding to Character Name Input) or continue a previous save (jumping directly to the Save/Load system). The narrative is introduced through the initial dialogue scene, after which players can freely navigate between the two hub areas (Castle and Town) and the main Dungeon.

During dungeon exploration, players encounter enemies that trigger the Combat system, which can result in either victory (returning to exploration) or defeat (leading to the Game Over screen). The dungeon features a progressive floor system that culminates in a boss encounter on Floor 5, which serves as the climax of the game.

The circular arrow in the Dungeon node represents the floor progression mechanic, where defeating a floor's challenges allows access to deeper levels. This creates the core gameplay loop that drives the player's experience.

This flow diagram showcases how the different systems integrate - from narrative elements to exploration, combat, and progression.

Appendix E: Project Diary Excerpts

Markdown

```
# Project Diary

## Week 27 (W.C. 8th April)

- Added final polish to the boss battle sequence
- Completed extensive testing on all game systems
- Fixed remaining edge cases in save/load functionality
- Added additional dialogue variations for storytelling
- Created detailed documentation of code structure for
submission

Plans for next week:
- Submit final project and report

## Week 26 (W.C. 1st April)

- Completed the final boss battle implementation with unique
moves and increased difficulty
- Added victory sequence after boss defeat that transitions to
scene_2
- Fixed critical bugs in battle system and save/load
functionality
- Conducted final playtesting sessions and made balance
adjustments
- Prepared documentation for final submission

Plans for next week:
- Add final polish to all game systems
- Complete extensive testing
- Finalise documentation

## Week 25 (W.C. 1st April)
- Completed the final boss battle implementation with unique
moves and increased difficulty
- Added victory sequence after boss defeat that transitions to
scene_2
- Fixed critical bugs in battle system and save/load
functionality
- Conducted final playtesting sessions and made balance
adjustments
- Prepared documentation for final submission

Plans for next week:
- Complete and submit final report
- Prepare for demonstration and viva

## Week 24 (W.C. 25th March)
- Implemented the Dungeon Guardian boss encounter with special
attacks
- Expanded battle UI to support boss-specific features and
visuals
- Added special reward (Ultimate Strike ability) upon boss
defeat
- Enhanced tile effects with visual feedback and player
notifications
```

```
 43. - Fixed multiple bugs related to enemy movement and collision
detection
 44.
 45. Plans for next week:
 46. - Implement victory sequence
 47. - Fix any remaining bugs
 48. - Conduct final playtesting
 49.
 50. ## Week 23 (W.C. 18th March)
 51. - Added experience system with level-up mechanics and
progression
 52. - Implemented player stat increases on level-up (HP, attack,
defense)
 53. - Added new abilities unlocked at specific levels (Firebolt,
Earthquake)
 54. - Created floating text display for damage, healing, and
experience
 55. - Improved battle animations with tween effects
 56.
 57. Plans for next week:
 58. - Implement final boss battle
 59. - Balance enemy difficulty across dungeon floors
 60. - Fix any bugs in the progression system
 61.
 62. ## Week 22 (W.C. 11th March)
 63. - Enhanced battle system with turn indicators and move selection
 64. - Added visual feedback for attacks, healing, and defending
 65. - Implemented flee mechanics with decreasing success chance
 66. - Created enemy-specific attacks and battle behaviors
 67. - Improved battle UI with health bars and move buttons
 68.
 69. Plans for next week:
 70. - Implement experience and leveling system
 71. - Add new player abilities unlocked at higher levels
 72. - Test and balance combat encounters
 73.
 74. ## Week 21 (W.C. 4th March)
 75. - Completed core battle system implementation with player and
enemy stats
 76. - Added turn-based combat mechanics with basic attack and
defense options
 77. - Implemented battle initiation when player collides with enemy
 78. - Created battle UI with health displays and combat options
 79. - Developed battle outcome handling (victory, defeat, rewards)
 80.
 81. Plans for next week:
 82. - Enhance battle system with additional moves
 83. - Add visual effects for combat actions
 84. - Implement flee mechanics
 85.
 86. ## Week 20 (W.C. 26th February)
 87. - Implemented interactive tile system with different effect
types (damage, heal, teleport)
 88. - Created visual indicators for tile activation
 89. - Added minimap display showing explored areas, player position,
and points of interest
 90. - Developed fog of war system for dungeon exploration
 91. - Enhanced player movement with smoother transitions and
rotation
 92.
```

```
Plans for next week:
- Begin implementing battle system
- Create enemy stats and basic AI
- Design battle UI elements

## Week 19 (W.C. 19th February)
- Added enemy AI with pathfinding towards player
- Implemented enemy movement and collision detection
- Created enemy spawning system based on floor level
- Developed enemy-player interaction foundation for battle system
- Enhanced dungeon generation with increasing complexity per floor

Plans for next week:
- Implement special tiles with various effects
- Create minimap system
- Add fog of war for unexplored areas

## Week 18 (W.C. 12th February)
- Implemented save/load system with file I/O
- Created save slot UI with save data display
- Added confirmation dialogs for overwriting saves
- Developed global game state management
- Enhanced player stats tracking between scenes

Plans for next week:
- Implement enemy AI and movement
- Create spawn system for enemies based on dungeon floor
- Begin designing battle system architecture

## Week 17 (W.C. 5th February)
- Added game over screen with options to load or return to title
- Implemented player health UI with stats display
- Created dialogue box system for in-game messages
- Enhanced scene transitions with smoother effects
- Added player stats persistence between scenes

Plans for next week:
- Implement save/load system
- Create UI for save slots
- Add persistent player data between sessions

## Week 16 (W.C. 29th January)
- Enhanced town scene with interactive NPCs and menu options
- Added dialogue system for NPC conversations
- Implemented menu navigation in town with keyboard input
- Created transition between town and dungeon scenes
- Added confirmation dialog for quitting the game

Plans for next week:
- Implement game over screen
- Create player health UI
- Develop dialogue box system for notifications

## Week 15 (W.C. 22nd January)
- Developed the castle scene with dialogue options
- Created second dialogue scene for post-dungeon narrative
- Implemented king character interaction
- Added scene transitions between narrative sequences
```

```
- Enhanced dialogue system with character expressions

Plans for next week:
- Develop town scene with interactive elements
- Implement NPC dialogue system
- Create menu navigation in town

## Week 14 (W.C. 20th January)
- Created title screen with menu options and animations
- Implemented name input system for character creation
- Added first narrative scene with dialogue sequencing
- Developed character expression changes during dialogue
- Created smooth transitions between opening scenes

Plans for next week:
- Develop castle scene with dialogue options
- Create narrative continuation after dungeon completion
- Implement character interactions in castle

## Week 13 (W.C. 13th January)
- Implemented multi-floor dungeon system with increasing difficulty
- Added stairs for navigating between floors
- Enhanced player movement with improved collision detection
- Created ladder object with visual highlighting when player is nearby
- Added initial preparations for tracking player progress

Plans for next week:
- Develop title screen and menu system
- Create name input functionality
- Begin implementing narrative scenes

## Week 12 (W.C. 6th January)
- Improved procedural generation with floor-based scaling difficulty
- Added cell visibility handling based on adjacent cells
- Enhanced player movement with rotation and camera adjustments
- Implemented boundary checks and movement validation
- Created initial floor transition mechanics

Plans for next week:
- Implement multi-floor dungeon system
- Add stairs for level navigation
- Enhance player movement with improved collision

## Week 11 (W.C. 9th December)
- Break for holidays - limited development work
- Planned Term 2 development roadmap
- Researched turn-based combat systems for future implementation
- Sketched UI designs for battle system and town scene

Plans for next week:
- Continue holiday break with limited planning work
- Research implementation approaches for save/load systems

## Week 10 (W.C. 2nd December)
- Finalised interim report.
- Completed and rehearsed presentation for interim review.
```

```
- Refined procedural generation code to fix minor bugs in map generation.
- Improved UI responsiveness for player name input and dialogue scenes.

Plans for next week:
- Deliver interim presentation
- Begin preparing for holiday development planning

## Week 9 (W.C. 25th November)
- Drafted interim report, focusing on objectives and methodology.
- Finalised presentation slides and script for interim review.
- Debugged interaction issues with stairs cells in dungeon floors.
- Conducted playtesting to identify additional bugs.

Plans for next week:
- Finalise and submit interim report.
- Polish presentation for interim review.

## Week 8 (W.C. 18th November)
- Integrated map selection scene with scene transitions.
- Added functionality for player name input and storing name globally.
- Implemented a basic dialogue system with text animations.
- Fixed minor grid alignment issues in dungeon movement.

Plans for next week:
- Continue debugging dungeon interactions and transitions.
- Prepare interim report outline and structure.
- Begin drafting presentation for interim review.

## Week 7 (W.C. 11th November)
- Refined grid-based movement to ensure smooth traversal between cells.
- Improved procedural generation constraints to balance dungeon difficulty.
- Added placeholder assets for dungeon walls and interactive cells.
- Researched effective UI layouts for name input and map selection.

Plans for next week:
- Start implementing the dialogue cutscene functionality.
- Conduct further testing on procedural generation logic.
- Finalise dungeon layout algorithms.

## Week 6 (W.C. 4th November)
- Optimised random walk algorithm for procedural generation.
- Created logic to identify and store accessible cells in the dungeon.
- Implemented basic player interaction with dungeon elements (e.g., stairs).
- Experimented with animations for dungeon entry transitions.

Plans for next week:
- Focus on enhancing player movement mechanics.
- Begin developing the map selection scene.
- Debug and refine dungeon generation scripts.
```

```
256.
257. ## Week 5 (W.C. 28th October)
258. - Implemented grid-based movement mechanics for the player.
259. - Integrated a basic dungeon generation system with rooms and
corridors.
260. - Conducted initial tests on procedural layouts.
261. - Researched methods for implementing a name input system.
262.
263. Plans for next week:
264. - Optimise grid-based movement logic.
265. - Explore additional procedural generation algorithms for
variety.
266. - Begin integrating dungeon entry points.
267.
268. ## Week 4 (W.C. 21st October)
269. - Began building basic cell room in Godot.
270. - Used the cell to create simple dungeon rooms connected by
corridors.
271. - Researched additional procedural generation techniques for
dungeon design.
272.
273. Plans for next week:
274. - Implement a random walk algorithm for dungeon generation.
275. - Add player movement mechanics to navigate the grid.
276. - Explore options for visualising the dungeon layout.
277.
278. ## Week 3 (W.C. 14th October)
279. - Researched procedural generation in Godot.
280. - Watched tutorials on building a dungeon in Godot.
281. - Practiced on Crocotile3D for designing placeholder assets.
282. - Created a basic scene structure in Godot for the dungeon.
283.
284. Plans for next week:
285. - Begin building cell-based dungeon layouts.
286. - Test procedural generation algorithms within the Godot
environment.
287. - Implement basic player controls.
288.
289. ## Week 2 (W.C. 7th October)
290. - Completed project plan.
291. - Decided on using random walk algorithm for dungeon generation
in Godot.
292. - Started designing the basic player movement system.
293.
294. Plans for next week:
295. - Explore procedural generation implementation in Godot.
296. - Build a basic grid system for dungeon cells.
297. - Test player movement mechanics in Godot.
298.
299. ## Week 1 (W.C. 30th September)
300. - Developed project abstract with details on game concept and
mechanics.
301. - Researched procedural generation algorithms (random walk and
cellular automata).
302. - Reviewed design patterns from *Game Programming Patterns*
(Robert Nystrom).
303. - Explored the capabilities of Godot 4 for procedural dungeon
generation.
304.
305. Plans for next week:
```

```
306. - Finalise project plan and prepare for supervisor review.
307. - Begin setting up the Godot project structure.
308. - Investigate early implementations of grid-based movement.
```