Final Year Project Report

Full Unit – Interim 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

December 13, 2024

Declaration

Word Count: 6257

Student Name: Salma Bocus

Date of Submission: 13th December 2024

Signature: [illegible]

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. Ultimately, this project is both a creative endeavour and a technical exploration of modern game development practices.

1.3 Milestones

My timeline for this project is now updated from my project plan to reflect the progress I have made in Term 1, as well as the goals I plan to achieve in Term 2. The milestones cover both completed tasks and future objectives. During Term 1, I focused on research and familiarisation with Godot, implementing core mechanics such as procedural dungeon generation, player movement, and basic interaction systems. Additionally, I developed foundational narrative components, including dialogue cutscenes and map selection functionality. As I move into Term 2, my focus will shift to refining the combat system, introducing animations, and expanding the game's mechanics with advanced AI behaviors, traps, and visual enhancements. The final stages will involve rigorous testing, optimisation, and deployment on itch.io.

1.3.1 Term 1

A timeline of my progression this term:

Week 1:	Familiarised myself with Godot's interface, scene tree structure, and GDScript.
Week 2:	Researched procedural generation algorithms, including random walk techniques, and reviewed Factory, Observer, and State design patterns.
Week 3-4:	Implemented a basic random walk algorithm in Godot and used a grid-based system to dynamically generate dungeon layouts.
Week 5-6:	Developed grid-based movement for the player and integrated interaction mechanics, such as transitioning between rooms and interacting with stairs.
Week 7-8:	Introduced a basic dialogue system, refined dungeon generation constraints, and added map selection functionality for transitioning to dungeon scenes.
Week 9-10:	Prepared for the interim review by refining procedural generation, debugging interactions, completing the presentation, and writing the interim report.
Week 11:	Finalised the first term report and submitted a working dungeon prototype showcasing procedural generation and basic gameplay mechanics.

1.3.2 Term 2

A timeline depicting what I hope to achieve:

Week 1-2:	Expand enemy AI behaviors using finite state machines and introduce adaptive difficulty for challenges.
Week 3-4:	Develop advanced player abilities, animations, and visual effects for spells and abilities.
Week 5-6:	Finalise UI elements, such as dungeon maps and an inventory system, while integrating sound effects and background music.
Week 7-8:	Conduct playtesting sessions to gather feedback, address usability issues, and optimise performance for smooth gameplay.
Week 9-10:	Prepare the final game build for deployment on itch.io and complete the final report in preparation for the viva.

1.4 Challenges

1.4.1 Learning Curve of Godot

As Godot is 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.

1.4.2 Procedural Generation Complexity

Generating diverse and balanced dungeon layouts requires 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.

1.4.3 AI and Interaction Systems

Developing adaptive AI with dynamic behaviour is a key goal for the next phase of the project. I plan to employ state machines and pathfinding algorithms to create modular and scalable designs. Enemy behaviours will be tested in isolated environments to ensure iterative refinement before full integration into the game.

1.4.4 Performance Issues

Procedural generation and real-time interactions can be resource-intensive, leading to occasional frame drops. Optimisation strategies such as object pooling, spatial partitioning, and selective updating are being implemented to enhance performance without compromising functionality.

1.4.5 Limited Feedback

Obtaining consistent playtesting feedback has been a challenge. I plan to address this by organising formal playtesting sessions and distributing surveys to gather insights on usability, accessibility, and player satisfaction. Regular iterations based on feedback ensure continuous improvement.

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 randomised 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].

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. Below is my UML Class diagram depicting the scenes and nodes within my Godot 4 project, as well as the GDScript methods and variables.

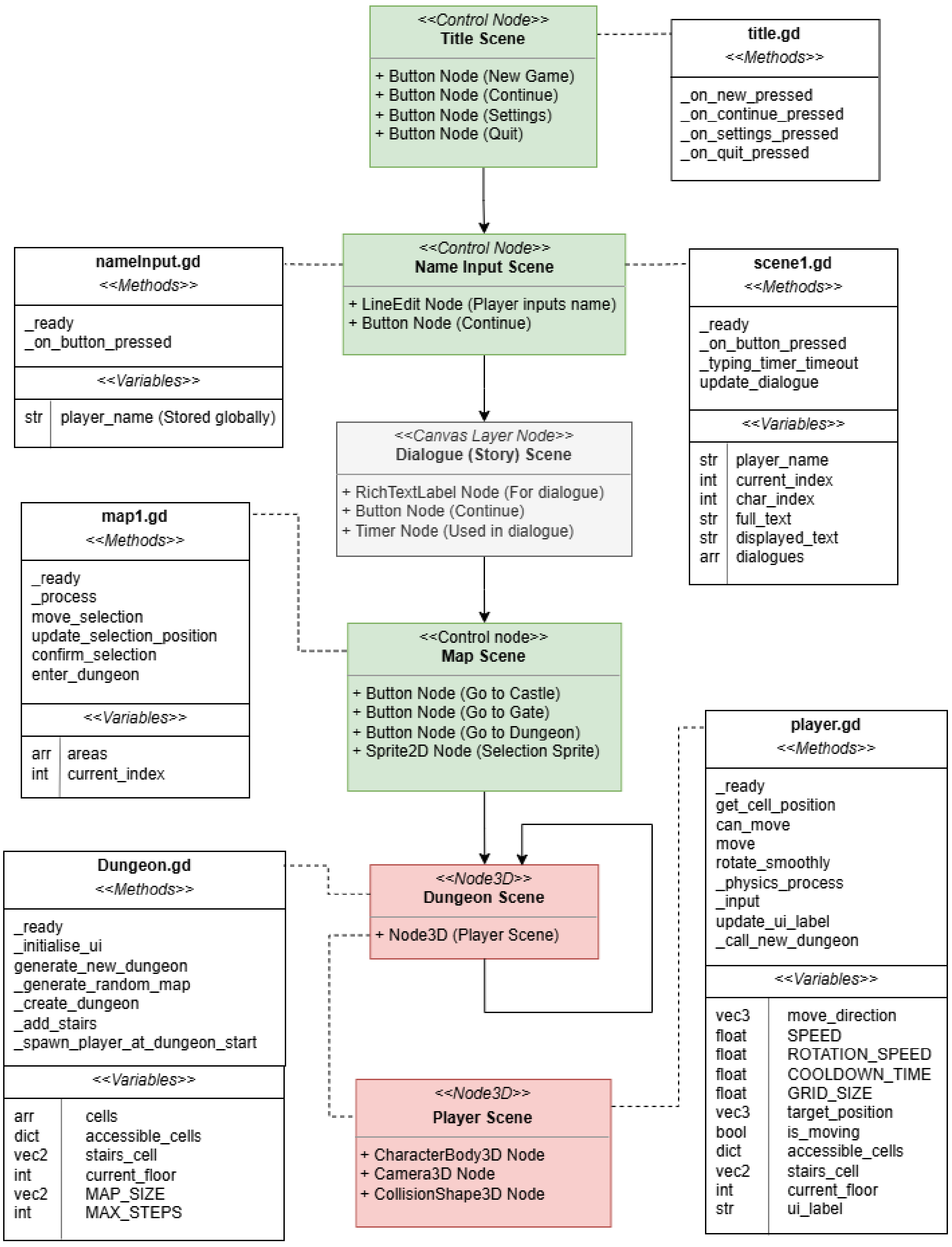

Figure 1 Godot 4 UML Diagram

3.1.1 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 randomised 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.

3.1.2 Interface and Scene Transitions

A sequence of interconnected scenes provides the narrative and structural flow of the game. Each scene, as outlined in the accompanying diagram (Figure 1), 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 Scene**: Transitioning from the narrative, players enter the Map Scene, which introduces a navigable interface for selecting locations. A `Sprite2D` selection marker, controlled by arrow key inputs, allows players to highlight options such as Castle, Dungeon, and Gate. Methods `move_selection` and `confirm_selection`, defined in the `map1.gd` script, allow for interaction. Pressing the spacebar on the Dungeon option transitions the player to the core gameplay environment.

- **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. For example, pressing the spacebar on a designated stairs cell triggers a transition to the next dungeon floor.

The modular architecture of these scenes, nodes, and scripts 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 require targeted solutions. Each iteration aims to refine mechanics, improve performance, and enhance user experience.

3.2.1 Identified Issues

Below is a table of identified issues from my current project and how these issues can be addressed.

No.	Issue/Bug	Description	Possible Solutions
1	Movement Bug	Occasionally, the player is unable to move forward despite the absence of visible obstacles. This issue likely arises from discrepancies in the `can_move` method within `player.gd`, where accessible cells may not always be accurately validated against the grid structure.	Debugging this issue involves tracing player movement logic and validating collision conditions against precomputed cell data.
2	Duplicate Stair Cells	On deeper dungeon levels, multiple stair cells occasionally appear, disrupting intended gameplay logic. This anomaly stems from inconsistencies in the `_add_stairs` method within `Dungeon.gd`. The randomisation process lacks sufficient constraints to prevent redundant placement, necessitating stricter parameterisation of the algorithm.	Introduce validation step within the `_add_stairs` method to ensure that only one stairs cell is placed per floor. This will involve maintaining a set of potential stair cell positions and applying additional checks to confirm uniqueness before finalising placement.
3	Interaction Bug	At times, pressing the spacebar on a stairs cell fails to trigger the floor transition. This issue indicates a potential synchronization problem between the player's position and the stairs_cell variable.	Ensuring real-time updates to cell states and refining interaction checks will address this inconsistency.
4	Performance Issues	Frame rate drops and occasional freezing have been observed during gameplay, particularly during dungeon generation and movement transitions. These issues likely stem from inefficiencies in the procedural algorithms and the high computational demands of dynamic object handling.	Optimising the random walk algorithm to reduce redundant calculations, implementing object pooling for frequently used assets, and leveraging spatial partitioning techniques such as quadtree or grid-based culling can alleviate performance bottlenecks and enhance overall stability.

3.2.2 Testing Plan

To systematically address these challenges, I have devised a comprehensive testing framework:

1. Playtesting Sessions: Regular playtesting with diverse participants will provide qualitative insights into gameplay mechanics, usability, and performance.
2. Surveys: Structured surveys will collect player feedback on specific aspects such as accessibility, responsiveness, and perceived issues.

3. Performance Profiling: Leveraging Godot’s debugging and profiling tools will help identify bottlenecks and optimise resource utilisation during procedural generation and gameplay.
4. Targeted Prototyping: Isolating specific mechanics, such as grid-based movement and dungeon generation, for focused testing and refinement.

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. This section explores the application of the Factory, State, Command, Spatial Partitioning, Game Loop, and Decorator patterns within my project.

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_stairs()` methods of `Dungeon.gd`, where `Cell` and `Stairs` objects are dynamically instantiated based on procedural map data. For instance:

```
var cell = Cell.instantiate()
add_child(cell)
cells.append(cell)
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 `Stairs` can be modified independently of the logic governing their placement.

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].

4.2 State Pattern

The State pattern is a behavioural 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:

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

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

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].

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:

```
if event.is_action_pressed("ui_up"):
    move(-global_transform.basis.z)
```

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.

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.

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:

```
if is_moving:
    global_transform.origin = 
global_transform.origin.move_toward(target_position, SPEED * delta)
```

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

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].

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 centre of the grid. For the generation process to start at the centre of the map, this centre 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.

```
 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.

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 5x5, the map array could look like this after a random walk:

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

This is a sample output. 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.

```
 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 initialised (**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 neighbouring 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.

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.

5.2 Player Movement

5.2.1 Movement Function

This 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.

```
 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 stairs to navigate between dungeon floors.

```
func _input(event):
    if cooldown_timer.is_stopped() and not is_moving:
            if event.is_action_pressed("ui_up"):
                    move(-global_transform.basis.z)
            if event.is_action_pressed("ui_left"):
                    rotate_smoothly(deg_to_rad(90))
                    move_direction = -global_transform.basis.x
            if event.is_action_pressed("ui_right"):
                    rotate_smoothly(deg_to_rad(-90))
                    move_direction = global_transform.basis.x
    if get_cell_position(global_transform.origin) == stairs_cell and event.is_action_pressed("ui_accept"):
            current_floor += 1
            update_ui_label()
            print_debug("Changing floor")
            _call_new_dungeon()
```

This function only processes if `cooldown_timer` is stopped and the `is_moving` flag is `false`. 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_left` or `ui_right` action is pressed, then the player rotates smoothly 90 degrees to the corresponding direction and `move_direction` is updated to reflect this. The function also checks if the character's current cell matches the location of a stairway (`stairs_cell`) and if the `ui_accept` action is triggered then the player advances to the next floor. The `current_floor` counter is incremented by 1 and the user interface is updated to reflect the new floor.

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.

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

```
func get_cell_position(world_position: Vector3) -> Vector2i:
    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 in 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.

5.4 Changing Dungeon Floors

Below is a description of the functions that handle the logic for changing dungeon floors, including generating a new dungeon layout, adding stairs for navigation and updating the player’s state and position.

5.4.1 Adding Stairs

This function places a stair object in the dungeon, allowing the player to transition to the next floor.

```
1. func _add_stairs(_random_map: Array):
2.      var possible_stairs = accessible_cells.keys()
3.      stairs_cell = possible_stairs[randi() % possible_stairs.size()]
4.      var stairs = Stairs.instantiate()
5.      add_child(stairs)
6.      stairs.global_transform.origin = Vector3(stairs_cell.x *
Global.GRID_SIZE, 0, stairs_cell.y * Global.GRID_SIZE)
```

It takes the parameter `random_map`, a 2D array representing the current dungeon layout, where accessible cells are marked.

The `possible_stairs` (**line 2**) list is populated with all keys from the `accessible_cells` dictionary, representing all valid positions in the dungeon. A random position is selected from `possible_stairs` in line 3, and then a new `Stairs` object is instantiated and added to the scene tree. The `stairs` object is positioned at the selected `stairs_cell`, converted to world coordinates.

5.4.2 Generating a New Dungeon

The following function resets the current dungeon and generates a new one tailored to the given floor. This function is called when the player interacts with the stairs.

```
func generate_new_dungeon(floor: int):
    current_floor = floor
    MAP_SIZE = Vector2(4 + floor * 2, 4 + floor * 2)

    for cell in cells:
            cell.queue_free()
    cells.clear()
    accessible_cells.clear()
    stairs_cell = Vector2i()

    var random_map = _generate_random_map()
    _create_dungeon(random_map)
    _add_stairs(random_map)
    _spawn_player_at_dungeon_start(random_map)
```

```
15.
16.     player.accessible_cells = accessible_cells
17.     player.stairs_cell = stairs_cell
18.     player.update_ui_label()
```

The function takes the parameter floor, an integer representing the current dungeon floor number. Higher floors increase dungeon complexity, so the dungeon size (`MAP_SIZE`) is adjusted accordingly for the size to increase with each floor. The current dungeon is cleared in **lines 4-7**, and the `accessible_cells` dictionary and `stairs_cell` variables are reset in **lines 8 and 9**.

The function `generate_random_map()` that was mentioned previously is then called to create a new procedural map. This new map is passed to `_create_dungeon()` to instantiate and place cells in the scene. The stair object is placed and then the player is spawned at the starting point.

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].

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.

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] J. Gregory, Game Engine Architecture, Third Edition, CRC Press, 2018.

Comment: This book helped refine my understanding of Godot’s architecture, particularly its node-based system, enabling efficient scene organisation.

[20] P. Hatton, “Godot Engine vs Unity: which is right for you?,” Creative Bloq, 16 July 2024. [Online]. Available: https://www.creativebloq.com/3d/video-game-design/godot-engine-vs-unity-which-is-right-for-you.

Comment: This article validated my choice of Godot over Unity, highlighting its advantages for procedural generation and grid-based systems.

[21] J. Johnson, Godot 4 Game Development Cookbook, Packt Publishing, 2023.

Comment: This resource provided practical tips and code snippets for implementing features like scene transitions and player movement in Godot 4.

[22] GDQuest, “Learn GDScript,” GDQuest, [Online]. Available: https://gdquest.github.io/learn-gdscript/.

Comment: This online guide was invaluable for learning GDScript syntax and best practices.

[23] J. Linietsky and A. Manzur, “Godot Docs,” Godot Engine, 2024. [Online]. Available: https://docs.godotengine.org/en/stable/index.html.

Comment: The official Godot documentation was a constant reference throughout the project, offering detailed explanations of engine functionality and scripting techniques.