

Procedural Dungeon Generation

A Project Report
Presented to
The Faculty of the College of
Engineering
San Jose State University
In Partial Fulfillment
Of the Requirements for the Degree
Master of Science in Computer Engineering
Master of Science in Software Engineering

By
Akash Mattaparth
Ganesh S Tulshibagwale
Shyam Kumar Nalluri
Srikari Veerubhotla

August 2024

Copyright © August 2024

Akash Mattaparth
Ganesh S Tulshibagwale
Shyam Kumar Nalluri
Srikari Veerubhotla

ALL RIGHTS RESERVED

APPROVED

DocuSigned by:

Sinn, Richard

AESB4AC2A265432...

8/3/2024

Richard Sinn, Project Advisor

Haonan Wang, Director, MS Computer Engineering

Dan Harkey, Director, MS Software Engineering

Rod Fatoohi, Department Chair

ABSTRACT

Procedural Dungeon Generation

By Akash Mattaparth, Shyam Nalluri, Ganesh S Tulshibagwale, Srikari Veerubhotla

Procedurally generated content has been a tool of video game development for decades. Used carefully, it allows for increased variety and replayability without needing to hand make everything. One particular longstanding usage is the generation of “dungeons,” enclosed environments the exploration of which presents a discrete challenge to the player.

Dungeon design is a continuing challenge in game development, and procedural generation even more so, with a tradeoff between creative control and output variety. Existing popular algorithms and tools for dungeon generation too often are only concerned with creating a complicated structure with no further context or narrative, presumably under the assumption that further details are crafted bespoke to each game, with few attempts to instead describe the dungeon in a generic way that can be transferred between the contexts of different games.

We intend to create a mixed-initiative editor for designing video game dungeons. A human user will be able to design the logical narrative structure and draw the floor plan with some algorithmic assistance, hence the term “mixed-initiative.” The editor will be able to generate some starting designs and as the user works will be able to provide feedback on the dungeon. This includes providing information about the player’s expected flow through the dungeon, marking problem points such as unreachable keys or overly complex junctions or overly simple linear passages, and demonstrating the dungeon in play with simulated adventurers.

Acknowledgements

The authors are indebted to Professor Richard Sinn for his guidance during the development of this work.

Chapter 1. Project Overview.....	8
1.1 Introduction.....	8
1.2 Proposed Areas of Study and Academic Contributions.....	9
Academic Contribution:.....	9
1.3 Current State of the Art.....	9
Chapter 2. Project Architecture.....	11
2.1 Introduction.....	11
2.2 System Architecture Overview.....	11
2.3 Subsystem Design.....	11
Chapter 3. Technology Descriptions.....	15
3.1 Client Technologies.....	15
3.1.1 Godot Engine Overview.....	15
3.1.1.1 Features.....	15
3.1.2 GDScript.....	16
3.1.2.1 Features.....	16
3.2 Middle-Tier Technologies.....	17
3.2.1 Procedural Content Generation Algorithms.....	17
3.2.1.1 Graph Rewriting Algorithm.....	17
3.2.2 AI for Gameplay Testing.....	18
3.3 Data-Tier Technologies.....	18
3.3.1 Data Structures.....	18
3.3.1.1 Graphs and Trees.....	18
3.3.1.2 Tile-based Systems.....	18
Chapter 4. Project Design.....	19
4.1 Introduction.....	19
4.2 Design Methodology.....	19
4.3 System Components Design.....	19
4.3.1 Map Design.....	19
4.3.2 Map Smoothing and Refinement.....	20
4.3.3 User Interface Design.....	20
4.3.4 Feedback Collection Mechanism.....	21
Chapter 5. Project Implementation.....	22
5.1 Introduction.....	22
5.2 Algorithm Implementation.....	22
5.3 Visualization Implementation.....	23
5.4 User Interface Implementation.....	23
Chapter 6. Testing and Verification.....	26

- 6.1 Testing Strategy..... 26
- 6.2 Unit Testing..... 26
- 6.3 Integration Testing.....26
- 6.4 Verification Results..... 26
- Chapter 7. Performance and Benchmarks..... 27**
 - 7.1 Performance Metrics.....27
 - 7.1.1 Scalability..... 27
 - 7.1.2 Resource Utilization..... 27
 - 7.2 Analysis of Results.....28
 - 7.2.1 Performance Data..... 28
 - 7.2.2 Interpretation of Results..... 28
- Chapter 8. Deployment, Operations, Maintenance..... 29**
 - 8.1 Deployment Strategy..... 29
 - 8.2 Operational Considerations..... 29
 - 8.2.1 User Documentation..... 30
 - 8.3 Maintenance Plan..... 30
 - 8.3.1 Regular updates.....30
 - 8.3.2 Handling user feedback and Bug reports.....30
- Chapter 9. Summary, Conclusions, and Recommendations..... 31**
 - 9.1 Summary of Work..... 31
 - 9.2 Conclusions..... 31
 - 9.3 Recommendations for Further Research.....32
 - Glossary..... 32
- References.....33**
- Appendices.....34**
 - Appendix A: Additional Code Snippets..... 34
 - Appendix B: Detailed Data and Results.....35

Chapter 1. Project Overview

1.1 Introduction

Procedural content generation (PCG) is a longstanding method for video game development, renowned for increasing variety and replayability while minimizing manual design labor. A classic application is dungeon generation, which presents players with an enclosed, structured environment designed for exploration and challenge. However, striking a balance between creative control and variety remains a critical challenge in procedural dungeon generation.

Pereira et al. introduced an evolutionary algorithm for generating dungeon maps containing locked-door missions, aiming to align with user specifications through a tree-based structure that encodes room arrangements, connections, and positioning. Their approach also integrates narrative elements through strategic placement of keys and locks, providing challenging gameplay that was validated by 70 players who perceived these procedurally generated levels to be more enjoyable and difficult than their human-designed counterparts [3].

In another study, Torres and Gustavsson explored the application of L-systems to generate room shapes for 3D dungeon creation. Their research highlighted the adaptability of L-systems in forming dungeon layouts while adhering to predefined room patterns that align with different game requirements [5].

Similarly, in "Evolving Roguelike Dungeons with Deluged," the authors employed evolutionary algorithms to facilitate dungeon generation. The emphasis on replayability, challenge, and narrative cohesion echoes Pereira's findings, demonstrating the potential of evolutionary algorithms in PCG for consistent yet varied dungeon layouts

Lastly, the paper "Procedural Generation of Dungeons" further reinforced these concepts by discussing how PCG could dynamically evolve dungeon configurations, presenting new challenges and puzzles to the players .

These works underline the need for a flexible, user-driven dungeon generation tool that can adapt to varying user requirements while delivering engaging, context-independent challenges. Building upon their foundational contributions, this project seeks to harness universal design patterns to create procedurally generated dungeons, leveraging player feedback and analysis of in-game behavior to refine and customize the resulting layouts for enhanced gameplay.

1.2 Proposed Areas of Study and Academic Contributions

The primary aim of this study is to refine the use of PCG algorithms for dungeon generation. The proposed system will generate dungeons based on user-defined parameters and include a feedback module to refine these parameters based on player interactions. This iterative approach ensures dungeons not only meet aesthetic and functional requirements but also adapt to enhance player engagement and challenge.

Academic Contribution:

1. **Algorithm Enhancement:** This project intends to advance the current understanding and implementation of PCG algorithms in dungeon generation by focusing on the dynamic aspects of dungeon design, such as variability and complexity of layout, which are less explored in current literature.
2. **Feedback-Driven Adaptation:** The inclusion of a feedback loop, where generated content is refined based on user interaction, contributes to the fields of adaptive PCG and human-centered design in game development.

1.3 Current State of the Art

At the moment, the most well known dungeon generation tool is the one found on the donjon website. [1] It is useful as a starting point, but is very much aimed at and

tethered to the jargon and expectations of the Dungeons and Dragons fanbase. The user can select certain broad settings like size and shape pre-generation but has no power to tweak the output post-generation. It provides a floor plan of the dungeon's space and a list of room contents, but not the logical structure or how contents might interact with each other. It generates locked doors, but no corresponding keys.

Chapter 2. Project Architecture

2.1 Introduction

Procedural Dungeon Generation (PDG) is a game-changing approach in video game development that allows for the automatic creation of complex and varied dungeon environments. Unlike traditional methods where levels are manually designed piece-by-piece, PDG provides a powerful advantage by generating entire dungeon layouts on the fly. This system is made up of different modules that work together to create playable and interesting dungeon experiences. These modules follow the player's preferences to seamlessly fit into the game's development process.

2.2 System Architecture Overview

In this report, we explore the fundamental modules driving our procedural dungeon generation architecture. These modules form the backbone of our system, enabling the creation of diverse and immersive dungeon environments that align with user preferences and game design goals.

2.3 Subsystem Design

2.3.1 User Input Module: This module allows users to customize how their dungeon will be generated. Users can adjust different settings like the overall size of the dungeon, how complex it should be, what theme or style it should have, and whether to include specific features like puzzles, traps, or secret rooms. The User Input Module has an easy-to-use interface with clear controls and helpful explanations, so users can understand how their choices will affect the generated dungeon. This module is where users can provide their preferences, so the dungeon that gets created is personalized to their liking.

2.3.2 Generation Engine Module: This is the main module that actually creates the dungeon layouts. It uses a large collection of pre-design dungeon pieces and advanced algorithms to build complete and exciting dungeon environments based

on what the user has chosen. The module is smart and adapts to the user's preferences to make sure the generated dungeons are varied, appropriately difficult, and look good visually. It carefully places rooms, corridors, obstacles, and interactive elements in the dungeon, considering important factors like how rooms are connected, level pacing, and maintaining a consistent theme. The algorithms in this module are designed to optimize the dungeon layout, creating challenging but fun experiences that match what the user wants.

We start with a graph rewriting algorithm to generate the logical flow of the dungeon before worrying about a floor plan. If you can assure quality at this stage the actual floor plan, generated later, becomes a cosmetic bonus - the icing on the cake.

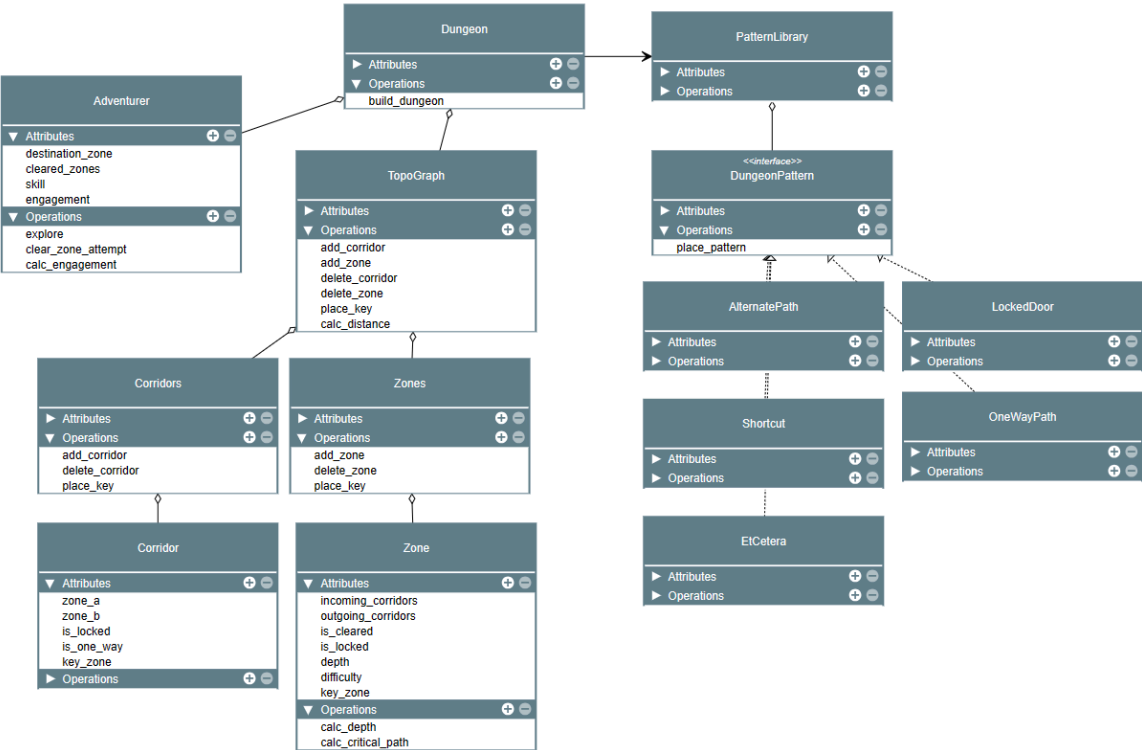


Fig 1. Generation Engine Module

2.3.3 Dungeon Evaluation Module: Once the initial layout is generated, we plan to use ‘procedural personas’ for playtesting and give feedback on the user. The personas used will have a default settings based on commonly understood player

style and game play. There will also be an option to adjust the default parameters of an additional persona if the user wishes so. The personas will use Monte Carlo Tree Search (MCTS) algorithm to play the game and are evolved using genetic programming to match their given utility functions (preferences). The resultant personas will give play and generate metrics to the user.

2.3.4 Feedback Interface Module: This module presents the results of the dungeon evaluation in an easy-to-understand way for the developers. It uses interactive visuals and summary reports to clearly show the key findings and recommendations from the analysis. This helps users make informed decisions on how to improve their dungeon designs. Through this module, developers can get valuable feedback from users and incorporate it into optimizing the algorithms, continuously enhancing the quality and effectiveness of the procedural generation process.

2.3.5 Output Module: This module takes the final generated and evaluated dungeon and prepares it for delivery. It combines all the dungeon content into playable demos or prototypes so users can actually experience the dungeon themselves. The module also allows users to save and export the dungeon in different file formats that can be used in games. This makes it easy to integrate the procedurally generated dungeons directly into existing game environments, streamlining the development process for game creators. The Output Module focuses on making sure the generated dungeon content is compatible and easy to incorporate into the final game product.

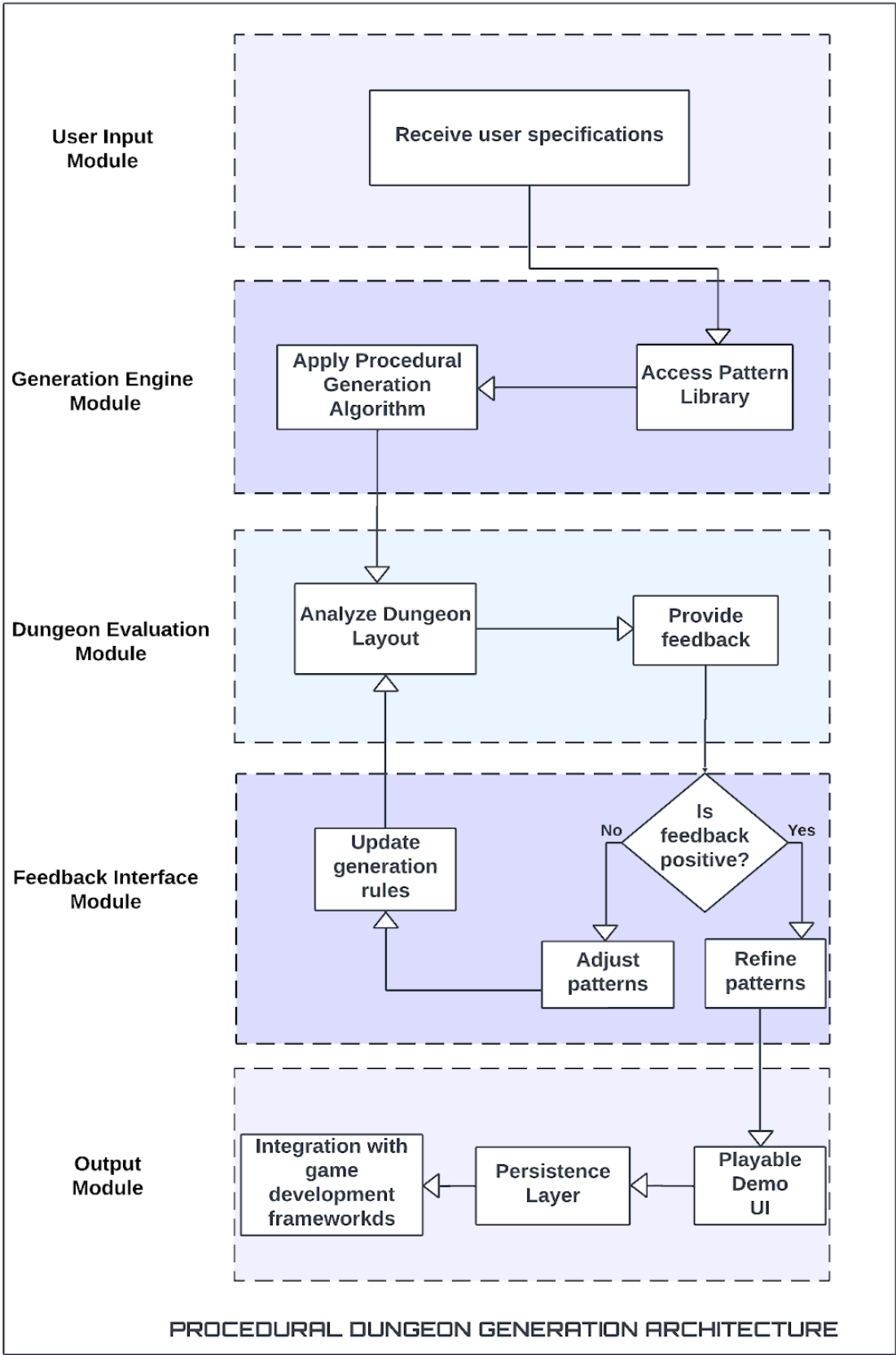


Fig 2. Procedural Dungeon Generation Architecture

Chapter 3. Technology Descriptions

3.1 Client Technologies

In this section, we explore the client-side technologies used in our Procedural Dungeon Generation (PDG) system. Procedurally generated dungeons can be created, modified, and implemented using these technologies as a base.

3.1.1 Godot Engine Overview

Godot is an open-source game engine known for its versatility and ease of use in both 2D and 3D game development. It provides a comprehensive set of tools for creating interactive game environments.

3.1.1.1 Features

- a. Intuitive scene-driven design:** Godot uses a distinct node-based architecture that enables developers to create complex game scenes by combining multiple nodes. This modular approach simplifies the design and organization of game elements.
- b. Specialized 2D workflow:** Godot offers a dedicated 2D engine that is optimized for 2D game development. It provides tools such as tilemaps, parallax layers, and a powerful animation editor.
- c. Powerful 3D engine:** Godot's 3D engine is highly capable, featuring modern rendering techniques, real-time global illumination, and robust physics simulations, enabling the creation of high-quality 3D games.
- d. Cross platform support:** Godot supports exporting games to multiple platforms, including Windows, macOS, Linux, Android, iOS, and web browsers, ensuring wide reach and compatibility.

- e. **Open and free software:** As an open-source engine under the MIT license, Godot is free to use, modify, and distribute. This openness encourages community contributions and ensures that there are no licensing fees or restrictions.

3.1.2 GDScript

GDScript is a high-level programming language designed specifically for the Godot engine. It is object-oriented, imperative, and supports gradual typing. The syntax of GDScript is indentation-based, similar to Python, which makes it easy to read and write. The primary goal of GDScript is to be optimized for seamless integration with the Godot engine, providing developers with significant flexibility for content creation and integration.

3.1.2.1 Features

- a. **High-Level language:** GDScript is a high-level programming language, which provides a strong abstraction from the underlying hardware. This simplifies the process of writing and comprehending code.
- b. **Object Oriented:** GDScript supports object oriented programming (OOP), which allows programmers to create classes and objects. This helps to organize code into parts that can be reused and easily managed.
- c. **Imperative paradigm:** GDScript follows the imperative programming paradigm, which focuses on describing how a program operates through statements that change the program's state.
- d. **Gradually typed:** GDScript is gradually typed, which uses both static and dynamic typing. This helps us to define variables and functions flexibly, which ensures type safety when required while keeping the ease of dynamic typing.

- e. **Python-Like syntax:** GDScript uses an indentation-based syntax similar to Python. This makes the code more readable and reduces the need for brackets and semicolons.
- f. **Signal system:** It includes a robust signal system for event-driven programming. Signals can be emitted and connected to functions, facilitating communication between different parts of your game.
- g. **Built-in documentation:** It supports inline documentation comments, which can be used to generate auto-documentation. This helps in maintaining well-documented code that is easier to understand and collaborate on.

3.2 Middle-Tier Technologies

In this section, algorithms and AI techniques used in our Procedural Dungeon Generation (PDG) system were discussed. These algorithms and AI techniques helped in improving game-play and content generation for our system.

3.2.1 Procedural Content Generation Algorithms

Our PDG system is based on Procedural Content Generation (PCG) algorithms, which enables the automatic generation of a variety of engaging dungeon environments. Based on user-specified criteria, these algorithms create interactive elements, rooms, corridors, and dungeon layouts, offering gameplay variety and complexity.

3.2.1.1 Graph Rewriting Algorithm

Graph rewriting algorithms play a crucial role in establishing the logical flow and structure of dungeons within our PDG system. By defining rules and transformations on graph structures, these algorithms determine how dungeon elements connect and interact, laying the groundwork for coherent and engaging gameplay experiences.

3.2.2 AI for Gameplay Testing

AI techniques are employed for gameplay testing and refinement within our PDG framework. Through procedural personas and algorithms such as Monte Carlo Tree Search (MCTS), our system evaluates generated dungeons based on predefined utility functions and player preferences. This approach provides feedback and metrics, guiding iterative improvements in dungeon design and gameplay dynamics.

3.3 Data-Tier Technologies

This section explores the data-tier technologies used in our Procedural Dungeon Generation (PDG) system, which focuses on the structures that manage and organize the data required for efficient dungeon creation.

3.3.1 Data Structures

3.3.1.1 Graphs and Trees

Graphs and trees are fundamental in representing the relationships and hierarchies within dungeon layouts. Graphs are used to model the connections between different dungeon elements, while trees help in structuring hierarchical data such as room sequences and nested environments.

3.3.1.2 Tile-based Systems

Tile-based systems form the foundation of our dungeon layouts, allowing us to divide the environment into manageable, reusable units. Tiles provide a modular approach to building dungeons, enabling easy customization and variation in the generated environments.

Chapter 4. Project Design

4.1 Introduction

The project design section details the methodologies and strategies employed in developing the Procedural Dungeon Generation (PDG) system. This encompasses the design approach, system component design, map smoothing and refinement techniques, user interface design, and feedback collection mechanism.

4.2 Design Methodology

The PDG system adopts an iterative and modular design methodology. Iterative development allows for continuous refinement of the dungeon generation algorithms based on user feedback, while modular design ensures that different components of the system can be developed, tested, and maintained independently. This approach promotes flexibility, scalability, and ease of integration.

4.3 System Components Design

4.3.1 Map Design

The map design process is driven by user input, allowing for the customization of dungeon parameters such as size, complexity, and themes. The generation engine module employs a graph rewriting algorithm to establish the logical flow and structure of the dungeon, ensuring coherence and adherence to user specifications.

Steps:

- **User Input:** Users provide parameters including dungeon size, room number, and survival chance.

- **Graph Generation:** A graph rewriting algorithm generates the logical flow of the dungeon. A topological graph of the dungeon has subgraphs of itself iteratively replaced with subgraphs following certain design patterns representing common design tropes: locked doors with a key hidden elsewhere, forks in the road, and so on.
- **Room Placement:** Rooms are strategically placed based on the generated graph, ensuring connectivity and navigability.
- **Corridor Generation:** Corridors are created to connect rooms, forming a coherent dungeon layout.

4.3.2 Map Smoothing and Refinement

To enhance the visual and structural quality of the generated maps, various smoothing and refinement techniques are applied. These techniques include cellular automata for terrain smoothing and heuristic adjustments to ensure playability and aesthetic appeal.

Steps:

- **Terrain Smoothing:** Cellular automata are used to smooth the terrain, reducing irregularities.
- **Structural Refinement:** Heuristic-based adjustments ensure all areas are accessible and gameplay is engaging.
- **Validation:** The refined map is validated against design criteria to ensure quality and playability.

4.3.3 User Interface Design

The user interface (UI) is designed to be intuitive and user-friendly, facilitating easy input of dungeon parameters and visualization of generated dungeons. Key components of the UI include:

- Input Fields: For specifying dungeon parameters like room size, complexity.
- Generate Button: To initiate the dungeon generation process.
- Visualization Panel: To display the generated dungeon map.

4.3.4 Feedback Collection Mechanism

After a map has been generated, we have the option to allow an AI agent to play through the map using Monte Carlo Tree Search Algorithm. For every node(step), it rolls out different possible actions through branches and builds a tree. The branches are updated through back-propagation for an adjustable number of iterations (currently 1000); it can be adjusted depending on the size of the map and the processing time tolerance. Depending on the complexity of interaction during the agent runs, we offer few metrics that capture the essence of the generated maps.

Chapter 5. Project Implementation

5.1 Introduction

This chapter delves into the implementation details of the Procedural Dungeon Generation (PDG) system, outlining the steps taken to bring the project from design to a functional state. It covers algorithm implementation, visualization, user interface, feedback module, and integration and testing processes.

5.2 Algorithm Implementation

The core of the PDG system lies in its algorithm for dungeon generation. The main script handles this by using a combination of graph rewriting, cellular automata, and heuristic techniques to generate coherent and engaging dungeon layouts.

Key Components:

- Graph Rewriting: Used to establish the logical flow and structure of the dungeon.
- Cellular Automata: Applied for terrain smoothing to enhance visual quality.
- Heuristic Adjustments: Ensure that all dungeon areas are accessible and the layout promotes engaging gameplay.

The main functions in `dun_gen.gd` include:

- `visualize_border()`: Draws the borders of the dungeon.
- `generate()`: Initiates the dungeon generation process, placing rooms and creating corridors.
- `make_room()`: Recursively attempts to place rooms within the dungeon while avoiding overlaps and ensuring connectivity.

5.3 Visualization Implementation

Visualization in the PDG system is achieved using the Godot Engine's GridMap node. This allows for the rendering of the dungeon's layout, providing a visual representation of the generated rooms and corridors.

Steps:

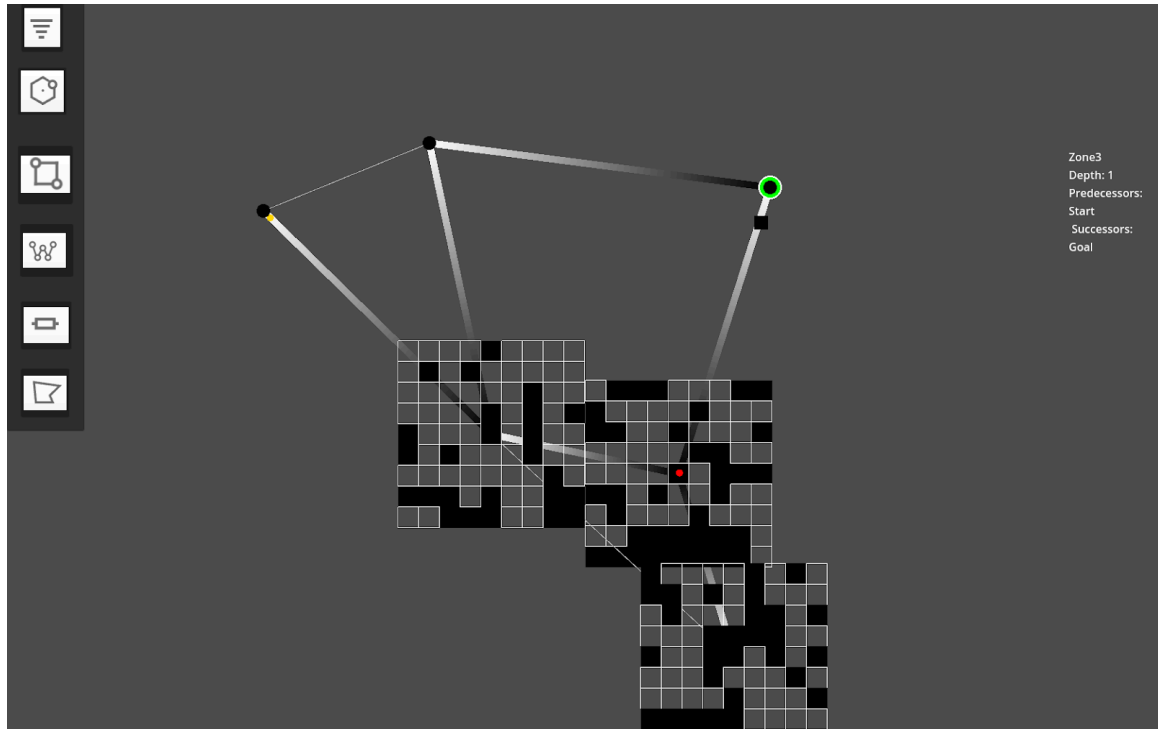
- Initialize GridMap: The `grid_map` variable is linked to the GridMap node in the scene.
- Draw Borders: The `visualize_border()` function draws the dungeon borders.
- Render Rooms and Corridors: The `generate()` and `make_room()` functions handle the placement and rendering of rooms and corridors.

5.4 User Interface Implementation

The user interface is designed to facilitate user interaction with the dungeon generation system, allowing users to specify parameters and visualize the generated dungeons.

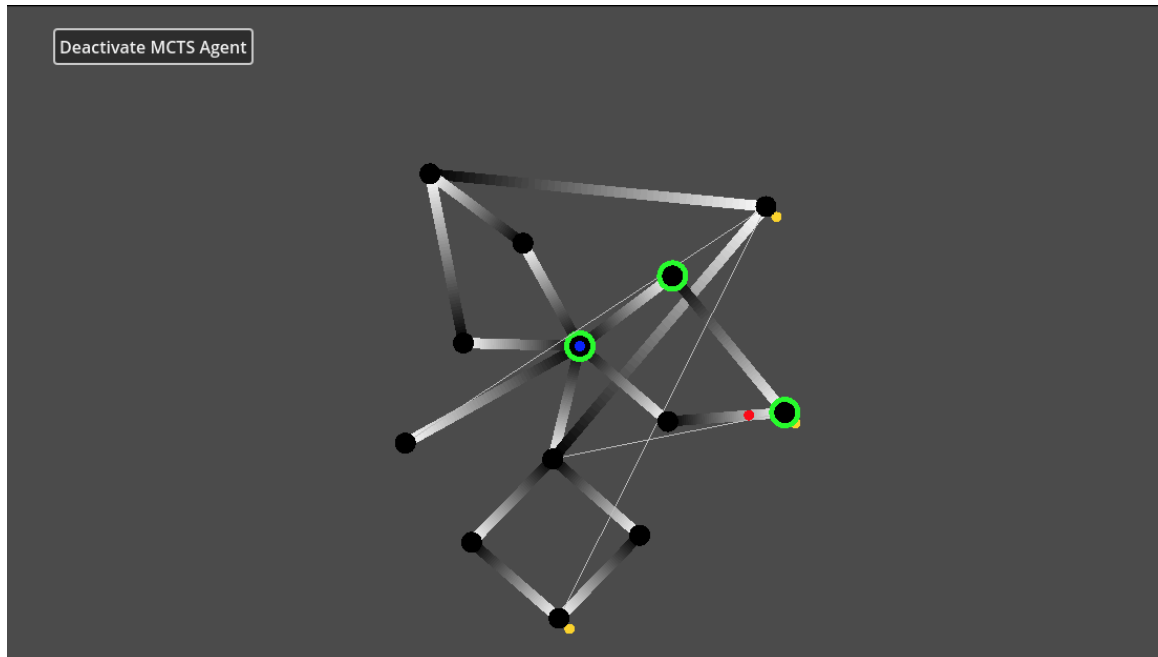
Components:

- Input Fields: Allow users to input parameters such as dungeon size, room number, and complexity.
- Generate Button: Triggers the dungeon generation process.
- Visualization Panel: Displays the generated dungeon layout.



Example Setup:

- Scene Setup: The scene contains a Control node with child nodes for LineEdit, Button, and GridMap.
- Script Attachment: The `dun_gen.gd` script is attached to handle user inputs and generation logic.



5.5 Feedback Module Implementation

- Button: Activate MCTS Agent: starts the agent play-testing the map and outputs feed-back metric.
- Script: MCTSAdeventurer.gd contains the algorithm.

Chapter 6. Testing and Verification

6.1 Testing Strategy

The testing strategy for the PDG system is designed to ensure that each component functions correctly, integrates seamlessly with other components, and delivers a satisfactory user experience. Our approach includes unit testing, integration testing, each targeting specific aspects of the system to identify and resolve issues early in the development process.

6.2 Unit Testing

Unit testing involves testing individual components of the PDG system in isolation to ensure that each part performs as intended. This phase is crucial for identifying and fixing bugs at the module level before they propagate to the system level.

6.3 Integration Testing

Integration testing ensures that individual modules interact correctly and that the system as a whole operates smoothly. This phase helps uncover issues related to module interfaces, data flow, and overall system behavior.

6.4 Verification Results

The verification results summarize the outcomes of the testing phases, highlighting key findings, resolved issues, and any remaining challenges. This section provides a comprehensive overview of the system's readiness for deployment and identifies areas that may require further attention.

Unit Testing: Most modules passed unit tests with minor issues resolved. A few complex modules required additional iterations for stability.

Integration Testing: Successful integration of all modules with smooth data flow and interface interactions. Some edge cases identified and addressed.

Chapter 7. Performance and Benchmarks

7.1 Performance Metrics

Performance metrics provide quantitative data on system efficiency and effectiveness. For our PDG system, the primary performance metrics are scalability and resource utilization. These metrics help in identifying performance bottlenecks and support optimization efforts.

7.1.1 Scalability

Scalability is the ability of a system to efficiently manage and perform under increasing loads by using additional resources. In our project, scalability is evaluated through Frames Per Second (FPS), which measures the game's performance as the scene complexity, or the number of active objects increases. Godot provides a built-in FPS counter that can be enabled to monitor the game's frame rate. A consistent FPS close to the target (60 FPS) indicates good performance.

7.1.2 Resource Utilization

Resource utilization is the effective use of system resources such as CPU, memory, disk I/O, and network bandwidth. It ensures that the system runs smoothly and efficiently, preventing any single component from becoming a bottleneck.

Godot's built-in profiler provides detailed information about CPU and memory usage that allows developers to identify and address performance issues. Key features of the profiler include:

- **CPU Usage:** Monitoring which processes or scripts consume the most CPU resources.
- **Memory Usage:** Tracking memory allocation to identify potential leaks or inefficient usage.

7.2 Analysis of Results

7.2.1 Performance Data

The performance data collected from our testing and benchmarking phases provides a comprehensive view of the PDG system's efficiency and effectiveness. This data includes metrics like average FPS, peak CPU usage, memory consumption patterns, and load times.

Data Collection Methods:

- **Automated Tools:** Utilize Godot's built-in profiling and monitoring tools.
- **Manual Observation:** Conduct manual tests for scenarios that automated tools might not cover.

7.2.2 Interpretation of Results

Interpreting the performance data involves analyzing the collected metrics to identify trends, bottlenecks, and areas for improvement. This analysis helps in making informed decisions about optimizations and enhancements needed for the PDG system.

Chapter 8. Deployment, Operations, Maintenance

8.1 Deployment Strategy

A well-defined deployment plan is crucial for ensuring that the Procedural Dungeon Generation (PDG) system is seamlessly integrated into the production environment. This section provides steps required to deploy a project in Godot.

Steps to Deploy in Godot

1. **Preparation:**
 - **Project Cleanup:** Remove any unnecessary files, assets, and debug information to ensure a clean project directory.
 - **Scene Organization:** Ensure all scenes, scripts, and resources are organized and correctly named for easy management and maintenance.
 - **Version Control:** Commit the latest changes to version control (e.g., Git) to keep track of the deployment version and maintain a history of changes.
2. **Testing:**
 - **Run Tests:** Conduct thorough testing within the Godot editor to ensure all functionalities work as expected, including gameplay mechanics and UI elements.
3. **Export Project:**
 - **Build for Each Platform:** Use the defined export presets to build the project for each target platform, ensuring compatibility and performance.
 - **Verify Builds:** Test the exported builds on their respective platforms to verify functionality, performance, and user experience.

8.2 Operational Considerations

Effective operations are crucial for the smooth functioning and maintenance of the Procedural Dungeon Generation (PDG) system. This section outlines the key

considerations for operational efficiency, ensuring that the system runs optimally and users have a positive experience.

8.2.1 User Documentation

User Documentation for our PDG project helps users get started, customize, and troubleshoot the software. It includes a getting started guide with system requirements, installation instructions, and a quick start tutorial. The user manual provides an overview of features, customization options, and examples of use cases. A troubleshooting guide addresses common issues and solutions, while the reference documentation covers API details and command references. Training materials, such as video tutorials and workshops, are available for further learning. Additionally, the updates and release notes section keeps users informed about the latest versions and improvements.

8.3 Maintenance Plan

Our maintenance strategy for the PDG project ensures the system remains up-to-date, reliable, and responsive to user needs. This strategy includes regular updates and efficient handling of user feedback and bug reports.

8.3.1 Regular updates

We will provide regular updates to the PDG system to improve performance, add new features, and ensure compatibility with the latest versions of Godot Engine.

8.3.2 Handling user feedback and Bug reports

User feedback and bug reports are crucial for the ongoing improvement of the PDG system. We will establish a dedicated feedback and support channel where users can report issues and suggest enhancements. Each report will be logged, prioritized, and addressed in a timely manner. Critical bugs will be patched immediately, while other feedback will be reviewed and incorporated into future updates as appropriate.

Chapter 9. Summary, Conclusions, and Recommendations

9.1 Summary of Work

The Procedural Dungeon Generation (PDG) system successfully leverages advanced algorithms and AI techniques to create engaging and varied dungeon environments. Through iterative development and user feedback, the system continuously improves, offering a robust solution for game developers seeking dynamic dungeon generation. Key

Successful Implementation: The PDG system meets its design goals and performs effectively across various scenarios.

User-Centric Design: The inclusion of user feedback mechanisms ensures the system remains responsive to player needs.

Performance Optimization: Ongoing efforts to identify and address performance bottlenecks have resulted in a stable and efficient system.

9.2 Conclusions

The Procedural Dungeon Generation (PDG) system represents a significant advancement in procedural content generation for video games. By combining user input, advanced algorithms, the system offers a powerful tool for creating dynamic and engaging dungeon environments. Continued development and optimization will ensure the system remains at the forefront of procedural content generation technology, providing game developers with the tools they need to create unique and compelling gameplay experiences.

9.3 Recommendations for Further Research

Future work will focus on expanding the capabilities of the PDG system, including exploring new algorithms, enhancing user interfaces, and integrating additional feedback mechanisms.

Algorithm Expansion: Investigating new procedural generation algorithms to further diversify dungeon layouts.

UI Enhancements: Improving the user interface to provide more customization options and a better user experience.

Feedback Integration: Developing more advanced feedback mechanisms to refine dungeon generation based on detailed player interactions.

Glossary

1. **Dungeon:** We use this term in the context of game world design to refer to enclosed spaces that present players with a challenge to navigate from start to finish through a complex and difficult environment.
2. **Graph Rewriting Algorithm:** A class of generative algorithm where pieces of topological graphs are repeatedly replaced with other patterns according to curated rules. They are essentially generative grammars adapted for topological graphs.
3. **Procedural Content Generation / Procedurally Generated Content:** In the context of game design, refers to the use of content such as environments, objects, characters, et cetera that has been created algorithmically as opposed to created by a human designer.
4. **Random Generation:** Colloquial misnomer for procedural content generation, which includes a degree of controlled randomness for the sake of variety but follows curated rules to output logically coherent content.

References

- [1] Donjon. Random Dungeon Generator [Web tool]. Retrieved from <https://donjon.bin.sh/fantasy/dungeon/>
- [2] Torres, P., and Gustavsson, P. (2017). L-system Application to Procedural Generation of Room Shapes for 3D Dungeon Creation in Computer Games. Chalmers University of Technology, Department of Computer Science and Engineering.
- [3] Pereira, L. T., Prado, P. V., Lopes, R. M., and Toledo, C. F. M. (2021). "Procedural generation of dungeons' maps and locked-door missions through an evolutionary algorithm validated with players," Expert Systems with Applications, vol. 180, p. 115009.
- [4] Baldwin, A., Dahlskog, S., Font, J. M., and Holmberg, J. (2017). Evolving Roguelike Dungeons With Deluged. Proceedings of the 12th International Conference on the Foundations of Digital Games.
- [5] Fernandez-Vara, C. (2017). Procedural Generation of Dungeons. MIT.

Appendices

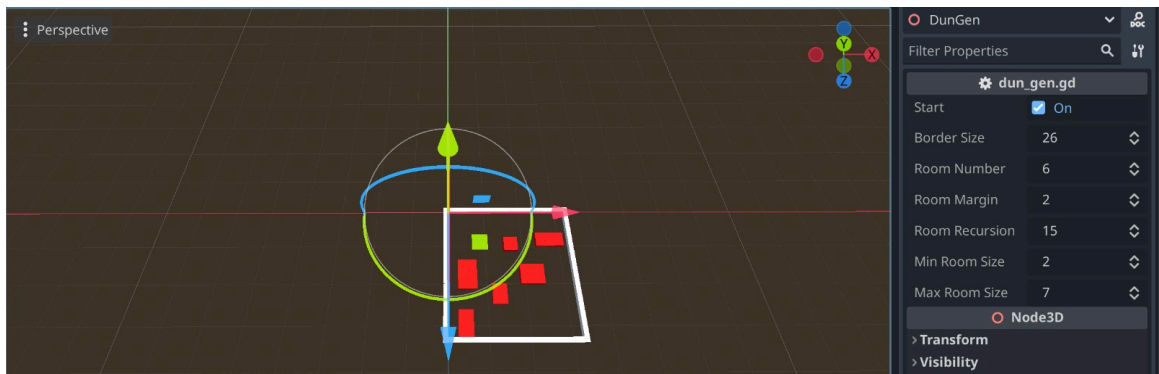
Appendix A: Additional Code Snippets

```
func make_room(rec:int):
>| if !rec>0:
>| >| return
>| var width : int = (rng.randi() % (max_room_size - min_room_size)) + min_room_size
>| var height : int = (rng.randi() % (max_room_size - min_room_size)) + min_room_size
>| var start_pos : Vector3i
>| start_pos.x = rng.randi() % (border_size - width + 1)
>| start_pos.z = rng.randi() % (border_size - height + 1)
>| #check for overlapping:
>| for r in range(-room_margin, height+room_margin):
>| >| for c in range(-room_margin,width+room_margin):
>| >| >| var pos : Vector3i = start_pos + Vector3i(c,0,r)
>| >| >| if grid_map.get_cell_item(pos) == 0:
>| >| >| >| make_room(rec-1)
>| >| >| >| return
>| var room : PackedVector3Array = [] >#For saving rooms for further use
>| #Draws the room
>| for r in height:
>| >| for c in width:
>| >| >| var pos : Vector3i = start_pos + Vector3i(c,0,r)
>| >| >| grid_map.set_cell_item(pos,0)
>| >| >| room.append(pos)
>| room_tiles.append(room)
>| var avg_x : float = start_pos.x + (float(width)/2)
>| var avg_z : float = start_pos.z + (float(height)/2)
>| var pos : Vector3 = Vector3(avg_x,0,avg_z)
>| room_positions.append(pos)
```

```
func designate_special_rooms():
>| if room_tiles.size() < 3:
>| >| print("Not enough rooms for special designations")
>| >| return
>| # Designate start room (preferably one with fewer connections)
>| start_room_index = rng.randi() % room_tiles.size()
>| # Designate end room (preferably far from start)
>| var max_distance = 0
>| for i in range(room_tiles.size()):
>| >| if i != start_room_index:
>| >| >| var distance = room_positions[i].distance_to(room_positions[start_room_index])
>| >| >| if distance > max_distance:
>| >| >| >| max_distance = distance
>| >| >| >| end_room_index = i
>| # Designate key room (not start or end)
>| var available_rooms = range(room_tiles.size())
>| available_rooms.erase(start_room_index)
>| available_rooms.erase(end_room_index)
>| key_room_index = available_rooms[rng.randi() % available_rooms.size()]
```

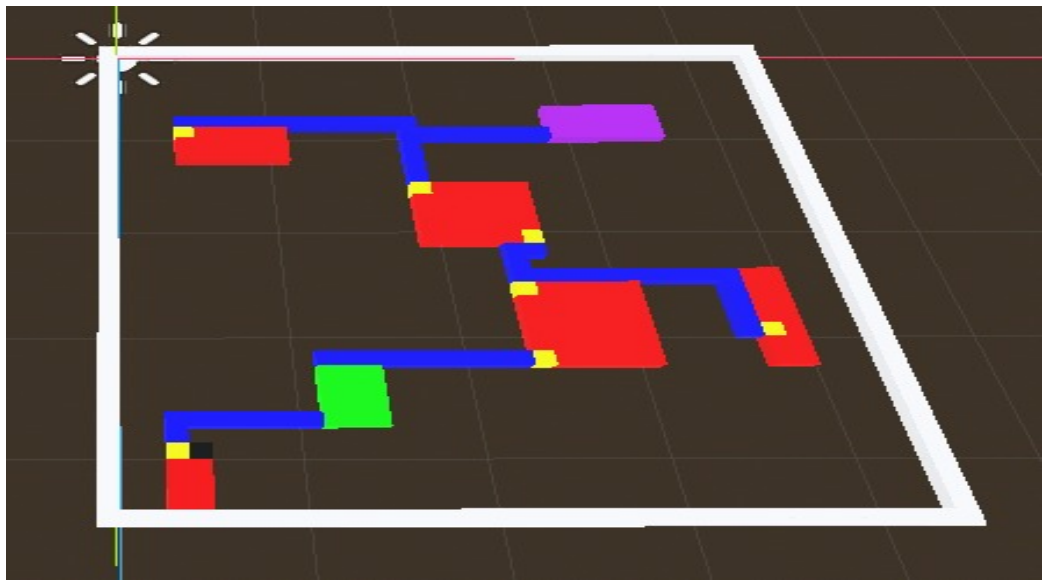
Appendix B: Detailed Data and Results

Dungeon Room Layout Generation:



The generated dungeon layout is represented by various colored rectangles within a bordered area. Each rectangle signifies a room in the dungeon.

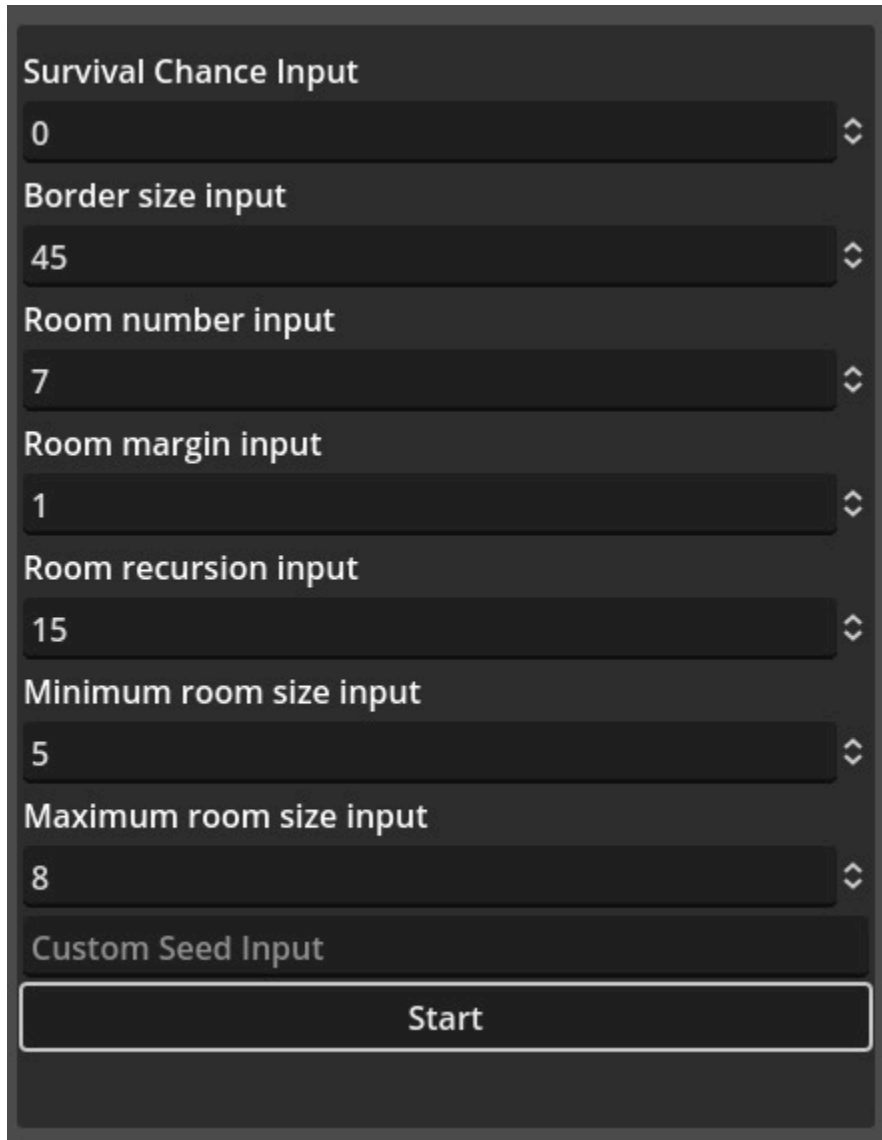
Color-Coded Dungeon Pathway Schematic:



The design includes a green start room (bottom left), blue hallways, red standard room tiles, yellow doors, and a purple end room (top right). This structure promotes exploration and strategic navigation by offering multiple route options. Yellow doors act as

checkpoints, while hallways connect various rooms, leading to the final goal in the upper right corner.

Configuration Panel:



Survival Chance Input
0

Border size input
45

Room number input
7

Room margin input
1

Room recursion input
15

Minimum room size input
5

Maximum room size input
8

Custom Seed Input

Start

The configuration panel allows users to input specific parameters to customize the generation of procedural dungeons.

1. **Survival Chance Input:** Sets the likelihood of certain features or elements surviving during the generation process.
2. **Border Size Input:** Defines the size of the border around the generated dungeon.
3. **Room Number Input:** Specifies the number of rooms to be generated within the dungeon.
4. **Room Margin Input:** Sets the margin or spacing between generated rooms.
5. **Room Recursion Input:** Determines the depth or complexity of room connections and layouts.
6. **Minimum Room Size Input:** Defines the smallest possible size for any room within the dungeon.
7. **Maximum Room Size Input:** Sets the largest possible size for any room within the dungeon.
8. **Custom Seed Input:** Allows users to input a custom seed value for generating repeatable dungeon layouts.

The **Start** button initiates the dungeon generation process based on the provided parameters.

Procedural Dungeon Generator Interface:

