International Journal of Creative Multimedia

Extending Recursive Backtracking for Procedural Generation of Interconnected Rooms and Staircases in Multi-Level 3D Dungeon Layouts

Zhen Shern Soh
sohzs-wp21@student.tarc.edu.my
Tunku Abdul Rahman University of Management & Technology, Malaysia
ORCID iD: 0009-0006-8629-4012

Kah Chun Chong
chongkc-wm22@student.tarc.edu.my
Tunku Abdul Rahman University of Management & Technology, Malaysia
ORCID iD: 0009-0001-6849-3660

Bee Sian Tan
tanbs@tarc.edu.my
Tunku Abdul Rahman University of Management & Technology, Malaysia
ORCID iD:0000-0001-7088-7864
(Corresponding Author)

Jia Hui Ong
ongjh@tarc.edu.my
Tunku Abdul Rahman University of Management & Technology, Malaysia
ORCID iD:0009-0006-4746-7638

Chin Hui Ooi
ooich-wm20@student.tarc.edu.my
Tunku Abdul Rahman University of Management & Technology, Malaysia
ORCID iD:0009-0003-7058-3360

Kim Soon Chong
chongks@ucsiuniversity.edu.my
UCSI University
ORCID iD:0000-0002-0584-3010

Abstract

Procedural Content Generation (PCG) is a powerful tool in video game development, enabling the automated creation of diverse and replayable game environments. This paper presents a novel extension of the recursive backtracking algorithm, which adapted to procedurally generate interconnected rooms

International Journal of Creative Multimedia (2025) 6, 2, 23-47
doi: https://doi.org/10.33093/ijcm.2025.6.2.2

Published by MMU PRESS. URL: https://journals.mmupress.com/index.php/ijcm

and staircases across multiple vertical floors in 3D dungeon layouts. By using parameters such and maximum room chain lengths, staircase probability and directional constraints, the system generates coherent, non-repetitive and fully traversable environments. Through 13 test samples, the API demonstrates its ability to produce varied and scalable dungeons, including multi-level environments, showcasing its versatility and adaptability for diverse game design needs. Performance testing further reveals the API's efficiency, with low CPU and GPU demands, optimized memory usage, and high frame rates ensuring smooth and visually responsive gameplay. These results highlight the API's ability to deliver high-quality content generation while maintaining system stability, making it suitable for use on mid-range hardware. The implementation offers game developers a robust and flexible tool for creating dynamic and engaging game experiences with minimal resource overhead.

Keywords: Application programming interface; Dungeon generation; Procedural Content Generation; Recursive backtracking; Video game development
Received: 24 January 2025, **Accepted:** 1 July 2025, **Published:** 30 September 2025

Introduction

Procedural content generation (PCG) is a technique used to dynamically create complex and varied dungeon layouts by applying algorithmic rules and randomized factors (de Pontes, 2022; Volz, 2023). This approach ensures that every playthrough offers a unique experience, enhancing replayability and player engagement.

As the demand for unique, expansive, and replayable content grows in the gaming industry, an increasing number of developers are adopting PCG techniques to enrich their games. By utilizing PCG, developers can craft intricate dungeons that challenge players through the unpredictability and dynamism of procedurally generated content.

PCG is often implemented using application programming interfaces (APIs), which allow developers to customize parameters such as layout complexity, room types, and overall structure to align with specific gameplay objectives. This paper introduces a novel API that harnesses the power of PCG to generate 3D room-based dungeons using the Recursive Backtracking algorithm. The proposed solution provides game developers with a flexible and efficient tool for creating procedurally generated dungeons tailored to their design goals.

Related Work and Background

The concept of procedural content generation (PCG) has been utilized for several decades in gaming, tracing back to early text-based adventures and roguelike games (de Pontes et al., 2022; Volz et al., 2023). A notable example is "Rogue" (Viana et al., 2022), which employed random algorithms to generate distinct dungeon layouts for each playthrough. This innovative approach gave rise to the “Roguelike” genre, characterized by procedural level design and permanent death mechanics (Viana et al., 2022). However, earlier PCG techniques, relying solely on simple random number generators, often produced poorly structured dungeons, frustrating players and diminishing their gaming experience.

Advancements in technology have progressively addressed these limitations. Instead of depending on basic randomization, procedural generation has shifted toward more structured methods. Developers began experimenting with various algorithms to create balanced and coherent dungeon layouts, moving away from fully randomized, disjointed designs.

The introduction of sophisticated algorithms has marked a significant evolution in procedural generation, transforming it into a refined technique. As the industry matured, structured approaches such as Recursive Backtracking, Binary Space Partitioning (BSP), and the Random Walk Algorithm

were adopted to create dungeons that are not only coherent but also engaging for players (Bellot et al., 2021; Lan et al., 2023; Li et al., 2023).

Recursive Backtracking offers several advantages, primarily its ability to guarantee a traversable dungeon layout where every room is accessible from the starting point (Bellot et al., 2021). This makes it ideal for maze-like structures, as the interconnected spaces provide players with a sense of exploration and adventure. Additionally, this algorithm allows developers to define varying degrees of complexity in dungeon structures, enabling customization to suit different playstyles.

Binary Space Partitioning (BSP), while also a recursive algorithm, takes a distinct approach to generating complex environments. Originally developed for 3D graphics, BSP can be adapted for 2D dungeon generation through recursive division. It divides a space into two halves, repeatedly subdividing these areas until the desired number of rooms is created, forming a binary tree structure of rooms and corridors (Lan et al., 2023). Rooms are then placed within the resulting areas, and corridors are established to connect them. This approach produces structured and visually appealing dungeons, granting developers precise control over room sizes and shapes to serve specific gameplay purposes. However, BSP is less suitable for games requiring multiple floors, levels, or vertical traversal unless combined with other algorithms or techniques.

Random Walk Algorithm involves randomly traversing a grid from a starting point and continuing until a path is generated (Li et al., 2023). While this algorithm is simple to implement and can produce irregular maze-like results, it has limitations. Without proper guidance, the generated layout may become disconnected or form loops, potentially compromising the overall design quality.

Application of PCG in Games

The usability of procedural content generation (PCG) in video games is extensive and impactful, offering the ability to create expansive worlds without the need for manual level design, a significant advantage in modern gaming where players expect abundant and diverse content (de Pontes et al., 2022; Volz et al., 2023). PCG is widely employed across various game genres, each leveraging its unique benefits. Role-Playing Games (RPGs) frequently use procedural dungeons to deliver open-world experiences, ensuring engaging and fresh gameplay across multiple playthroughs (da Rocha Franco et al., 2024). Roguelike games heavily rely on PCG to generate unpredictable and challenging dungeons, enhancing their signature elements of permadeath and emergent gameplay. Adventure games also benefit from PCG, enabling the creation of dynamic and responsive environments that offer players exciting and unpredictable opportunities, ensuring novel and immersive experiences with each session (de Pontes et al., 2022; Volz et al., 2023). By generating diverse, adaptive, and content-rich

environments, PCG has become a cornerstone of modern game design, underscoring its value in the evolving gaming landscape.

Examples of Games using PCG

Several popular games have successfully integrated procedural dungeon generation to enhance gameplay variety and replayability. Spelunky (2008) features procedurally generated levels with traps, treasures, and enemies, ensuring challenging and diverse experiences. The Binding of Isaac: Rebirth (2014) uses randomized layouts and encounters to provide unique gameplay for each player. Darkest Dungeon (2016) leverages procedurally generated environments to create a tense, immersive atmosphere filled with relentless challenges. Similarly, Enter the Gungeon (2016) offers dynamic gameplay through its procedurally generated dungeons packed with enemies, weapons, and historical items.

Application of PCG in Games

A novel API has been developed as a room-based procedural dungeon generator, empowering developers to create diverse and replayable 3D dungeons by leveraging a predefined set of room prefabs. The API generates a room-based layout and assembles the dungeon by strategically placing these pre-designed rooms into the layout. To achieve this, the Recursive Backtracking algorithm is employed, ensuring the creation of coherent and traversable dungeon structures.

Dungeons are a vital element in many game genres, including RPGs, Roguelikes, and Horror games, where procedurally generated environments significantly enhance replayability and player engagement. Unlike static, pre-designed dungeons, procedural generation introduces adaptability and unpredictability, ensuring players experience fresh and challenging layouts with each playthrough.

This API is structured to give developers flexibility by allowing them to:

- **Design and save custom room prefabs:** The API dynamically arranges these prefabs into varied dungeon layouts, ensuring flexibility and uniqueness.
- **Adjust parameters:** Customize aspects such as the number of rooms, connection pathways, and room types to control the complexity and structure of the generated dungeons.
- **Implement predefined room types:** Include specific room types, such as end rooms, with placement criteria to ensure logical dungeon flow and accessibility.

Recursive Backtracking for Layout Creation

The process begins with a grid-based framework, where each cell represents a potential room. The Recursive Backtracking algorithm ensures that every room connects to the layout's starting point, eliminating isolated spaces and creating a fully navigable dungeon.

Room Initialization: The API begins by defining the starting room. Using the Recursive Backtracking algorithm, it incrementally places rooms by selecting available doorways and connecting neighbouring rooms.

Room Placement and Stair Generation: The API generates a dungeon consisting of interconnected rooms, each capable of having up to four doors—one on each side. These doors connect to other rooms, ensuring a coherent and traversable layout. The Recursive Backtracking algorithm is employed to generate the dungeon layout using a grid-based system, establishing the foundational structure.

Once the layout is created, the API iterates through the grid to place rooms and stairs, filling the layout with randomly selected pre-designed rooms. This process connects and completes the dungeon, resulting in a dynamic and interconnected environment.

Condition-Based Room Assignment: Room placement adheres to specific conditions based on room type. For instance, end rooms must have only one entrance with no additional doors. The API allows users to integrate their own pre-designed rooms into the layout, ensuring these rooms meet the specified conditions. Developers can also customize parameters such as the maximum number of rooms to tailor the dungeon to their desired specifications.

Replayability and Customization: Using this API, random 3D dungeons are generated with each run, providing a unique experience every time and enhancing the game’s replayability. Players are challenged to adapt to new layouts, preventing the dungeon from becoming predictable.

The API also offers developers the flexibility to adjust parameters, enabling them to create dungeons of varying scales and complexity to fit their games. Additionally, developers can integrate their pre-designed rooms with custom objects, introducing further variation and diversity into the dungeon design.

Algorithm Description

The algorithm employed in this API is Recursive Backtracking, a powerful technique for solving complex problems through systematic exploration. Recursive Backtracking is particularly effective for maze generation, producing a unique layout each time it is used, making it ideal for creating dungeon layouts. The recursive nature of the algorithm enables a trial-and-error approach, where it explores potential paths and backtracks when a dead end is encountered, ultimately ensuring the generation of a coherent and traversable dungeon.

Recursive Backtracking Implementation

Recursive Backtracking is an algorithmic paradigm that incrementally solves problems through recursion. The algorithm makes a series of choices, each leading to a new state. If a choice results in a dead end, the algorithm backtracks to the previous state and explores alternative paths. This process continues recursively until a solution is found or all possibilities are exhausted (Bellot et al., 2021)

As illustrated in Figure 1, a 4x4 matrix is provided to represent a path, where a value of 1 indicates an accessible tile, and 0 indicates an inaccessible tile. The algorithm starts at (0, 0) and aims to reach the destination at (3, 3). A recursive function is employed to check adjacent tiles of the current position. If an adjacent tile has a value of 1, the function is called recursively for that tile. The algorithm starts by randomly selecting an initial cell, explores its neighbours, and continues until no valid moves remain. When all neighbouring cells have been visited, the algorithm backtracks to previous cells to resume exploration. Once all cells in the grid are visited, the maze is complete. If no solution is found on the left node, the algorithm backtracks to the previous tile and examines other options. This process repeats until a valid path is identified. In this example, the solution is discovered on the second path. If no solution were found, the algorithm would continue backtracking to previous states, recursively exploring other possible paths until a solution is reached or all options are exhausted.

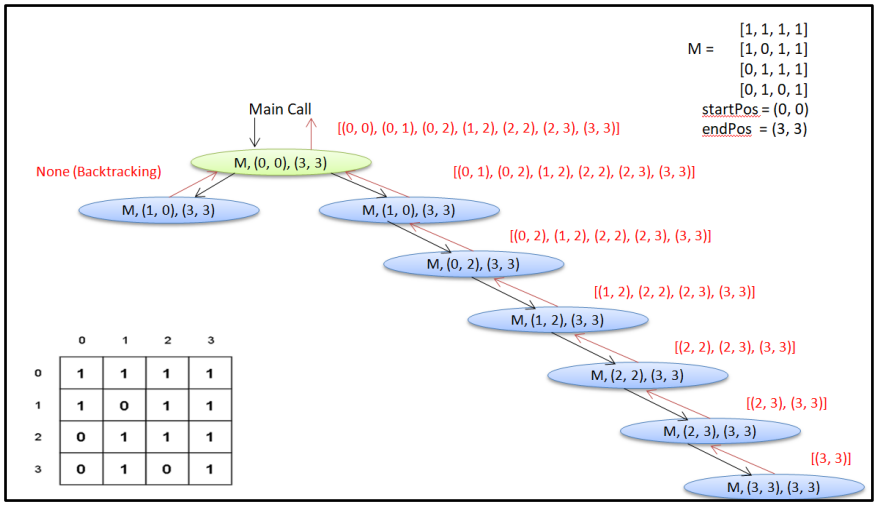


Figure 1. Visualization of Recursive Backtracking

Common use of the Recursive Backtracking Algorithm

Recursive Backtracking is widely used for maze generation, making it an ideal choice for creating dungeon layouts in video games. Dungeons in games often function similarly to mazes, requiring players to navigate through interconnected pathways to find exits, items, or other objectives. By procedurally generating a maze, it can serve as the layout for a dungeon, ensuring each layout is unique and provides players with a fresh experience every time they start a new game.

The process of maze generation using Recursive Backtracking begins with a grid of cells, where each cell represents a potential pathway. An initial cell is selected to begin the exploration, and neighbouring cells are examined to construct the maze. The key steps involved in maze generation are as follows:

1. **Starting Point:** A predefined or algorithm-selected starting cell is marked as visited and serves as the initial point for maze construction.
2. **Exploration:** From the current cell, neighbouring cells are examined to determine if they have been visited.
3. **Recursive Choice:** Among the unvisited neighbouring cells, one is randomly selected as the next cell. The algorithm then recursively calls itself, setting the chosen cell as the new current cell.
4. **Backtracking:** If all neighbouring cells are visited or no valid moves are available, the algorithm backtracks to the previous cell and continues the exploration from there.
5. **Completion:** The process repeats until all cells in the grid have been visited, resulting in a fully generated maze.

However, Recursive Backtracking comes with a limitation, where it is possible for a single path to continuously extend itself, leaving other paths idling for an extended period of time. To address such an issue, it is possible to implement conditions/restrictions that can limit the extension of a single path, forcing it to stop and backtrack once certain conditions are met.

Application of Recursive Backtracking to API Design

The concept of Recursive Backtracking is utilized to generate a maze, which serves as the foundation for creating the dungeon layout in the API. Using Unity with C# coding, a GameObject array can be implemented to represent the grid of cells. One element of the array is predefined as the starting point, from which exploration begins. To mark an element as visited, it is replaced with a placeholder cube GameObject, while null elements are treated as unvisited. To add further complexity to the dungeon, a 3D GameObject array is used, allowing exploration to extend vertically. This approach expands the layout across multiple layers, resulting in a dungeon that is both randomized and multi-levelled.

Since the API generates a dungeon without a fixed shape, several modifications are made to the standard Recursive Backtracking algorithm:

1. **Recursive Choice:** Instead of randomly selecting an element to create a new cube and continuing exploration from there, the API first determines which of the unvisited elements will be generated as a new cube. Exploration then proceeds from this newly created cube. This ensures that the generated dungeon adheres to specified constraints, such as limiting the number of rooms to a maximum value.

2. **Backtracking:** The backtracking process has been enhanced with additional restrictions and conditions to refine the layout:
    a. **End Cubes:** A cube is considered an end cube if it cannot generate additional cubes. This may occur due to randomization or when the maximum number of rooms has been reached. In such cases, the algorithm backtracks to the previous cube.
    b. **Preventing Premature Backtracking:** If randomization prevents additional cube generation but the maximum room count has not been reached, and the minimum chain length of rooms has not been achieved, the algorithm selects an unvisited element to generate a new cube. This ensures the chain continues, avoiding incomplete dungeon layouts.

3. **Completion Criteria:** The layout generation is deemed complete when either:
    a. The maximum number of rooms has been reached, or
    b. No unvisited elements remain to generate additional cubes.

Once the layout generation is finished, the placeholder cubes in the grid are replaced with actual room designs. The resulting dungeon layout is randomized, multi-layered, and ready for use in the game.

Pseudocode and Workflow Visualisation

Pseudocode for Recursive Backtracking Function

The pseudocode for the solve_backtracks() algorithm is presented in Table 1. This algorithm operates as follows:

- **Solution Check:** The function first checks if the solution is already found. If the solution is found, the function immediately returns true.

- **Iterating Through Candidates:** The algorithm iterates over each candidate in the list of potential solutions. A candidate represents a possible choice or moves for the current step in solving the problem.
- **Candidate Validation:** Within the loop, the algorithm checks whether the current candidate is valid. If the candidate is invalid, the loop skips to the next candidate using the continue statement.
- **Accepting a Candidate:** If a candidate is deemed suitable, it is accepted as part of the current solution. The algorithm then recursively calls solve_backtrack() to attempt solving the problem with the updated state. If this recursive call returns true, the solution has been found, and the function exits successfully.
- **Failure to Find a Solution:** If no candidate meets the requirements, or if all possibilities are exhausted, the function returns false.

This process ensures that the algorithm systematically explores and validates all possible solutions, backtracking when necessary, to find a viable path to the solution (Bellot et al., 2021).

Table 1. The solve_backtrack() pseudocode.

Algorithm 1 BackTracking Algorithm

```
function solve_backtrack():
  IF solutionIsFound
    return TRUE
  FOREACH candidate in candidateList:
    IF candidate is not valid
      continue
    SELECT candidate for the current position
    IF solve_backtrack():
      return TRUE
    ELSE
      remove candidate
  return FALSE
```

For the API's implementation, the concept of Recursive Backtracking is applied to generate rooms dynamically. The process begins with the creation of a startRoom using the GenerateRoom function. This function then performs a recursive call by invoking GenerateRoom(startRoom).

The GenerateRoom(Room currentRoom) function generates new rooms based on the input currentRoom (Table 2). Each room is assigned random door states (true or false) to determine which doors will lead to new rooms. New rooms are created by recursively calling the GenerateRoom function for each open door in the current room.

If the current room is designated as a stairs room, a temporary room is generated above or below it after all neighbouring rooms have been processed. Additionally, each room has specific variables to

assist with condition checking. For instance, a boolean variable, isEndRoom, determines whether a room qualifies as an End Room.

These conditions influence the generation of subsequent rooms, allowing for diverse and dynamic dungeon layouts. The recursion process concludes when there are no additional rooms to generate, at which point the algorithm backtracks to the previous room in the sequence.

Table 2. Recursive GenerateRoom() pseudocode

Algorithm 1 Initialize the GenerateRoom function
Create startRoom as a new Room object Set chainIndex of startRoom to 0 Save room to arrayOfRooms Increment the noOfRooms count Call GenerateRoom(startRoom)to begin room generation

Algorithm 2 Recursive Room Generation
IF currentRoom chainIndex is 0 AND noOfRooms greater than 1 set noOfDoors to 4 ELSE set noOfDoors to 3 IF currentRoom greater than maxChain OR noOfRooms has reached maxRoomNo): Set all doors to false currentRoom.isEndRoom = true RETURN to end this recursive call IF room with chainIndex greater than 0 Generate RandomNumber IF RandomNumber greater than or equals to stairs probability set stairNumber to 1 OR -1 to indicate upward or downward connection ELSE Set stairNumber to 0

Algorithm 3 Door Assignment
FOREACH noOfDoors representing directions like west, north, east and south IF neighbourRoom is occupied in that direction doorIsClosed ELSE Randomly assign the door to be open or close IF all doors are closed currentRoom.isEndRoom is TRUE

Algorithm 4 Recursive Room Placement
IF currentRoom is not an EndRoom FOREACH door that is opened Create a newRoom Set newRoom's chainIndex to currentRoom.chainIndex+1 Save room to array Increment of noOfRooms Call GenerateRoom(newRoom) to continue to recursive generation process

Algorithm 5 Stairs Generation
IF stairNumber is not 0 Create a tempRoom to connect above or below the currentRoom Set tempRoom.chainIndex to 0 Save tempRoom in the room array Call GenerateRoom(tempRoom)to add it to the layout

Figure 2 provides a visualisation of the working for the algorithms; the details of each step are as shown below:

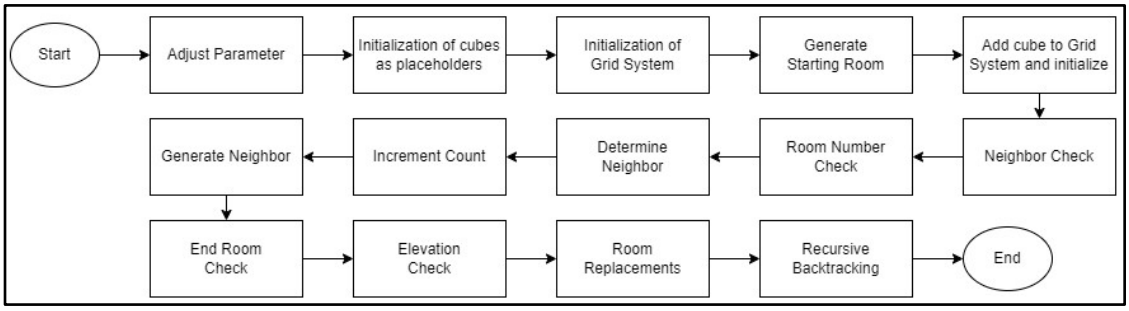


Figure 2. Recursive Backtracking API Flowchart

1. Users can first tweak the public parameters shown in Table 3.
2. Initialize 4 GameObjects cubes to act as placeholders that represent different types of rooms, **startRoom**, **normalRoom**, **stairsRoom**, **endRoom** and **wall**. An additional dummy cube will be initialized to act as placeholder for the room above or below the stairRoom.
3. **startRoom** - The starting/entrance room of the dungeon, essentially the first room the player will see.
4. **normalRoom** - Rooms generated without any special condition
5. **stairsRoom** - Rooms that allow players to climb to the next floor.
6. **endRoom** - Rooms without any door other than the entrance.
7. **wall** - Wall prefab used to block directions without doors. Users should also specify the thickness of the wall and the height of the wall for calculation purposes.
8. Each room will store variables as shown below:
    a. An integer to indicate its number in the chain
    b. A boolean to indicate if it's an end room
    c. An integer value to indicate type of stairsRoom (0 = non, 1 = Up, -1 = Down)
9. Initialize a **3D array** to use as our grid system.
10. First generate a starting room on [maxRoom/2][0][0]
11. Every time a cube is generated, add it to the gameObject array and create an array of bool to represent the west, north, east of the block respectively
12. Check if the neighbouring block is occupied, if it is, set false for the respective index.
13. If the room chain number is the same as the maximum chain number, set all to false. Else if not all entries are false, and the chain number is less than the minimum chain number, randomly assign a null entry to be true and increment room count.
14. Using a loop, for every null entry, check if maximum room is reached, if it is set to false, else randomly set true or false to entry.
15. Increment room count every time when true is set
16. Based on the array result, generate room; prioritize west, north then east room.
17. If the room does not have any doors, set true as an end room

18. Before recursing back to the previous room, check the upper block and if it is occupied, if it is not, based on stairsProb, randomly assign stairs number to 1, if it is assigned to 1, generate a dummy block with 4 directions instead of 3. If the upper block is occupied, check the bottom block, and repeat the same steps, but assign to -1 instead. Set the chain index of the room as 0.
19. Repeat the steps until the number of rooms reaches the limit.

After layout generation, loop through the 3D array, if the entry is not null, based on the type of placeholder cube, and randomly replace it with the respective room type prefab, then add doors based on the array in the cube. If it is a dummy cube, skip it, and if it is a stairs room, check the block above or below based on the stair index, and generate doors based on the results and stairs index

Table 3 shows the parameter used in the algorithm and the description for each parameter.

Table 3. Public and adjustable parameters

Parameter	Description
int maxRooms	The maximum number of rooms the dungeon will have. If users wish to have more rooms in their dungeon, they should increase this parameter, however, do note that this parameter only LIMITS the number of rooms, hence it is not guaranteed that the number of rooms generated will be or close to the specified number.
int minChain	The minimum number of rooms in a single chain of connected rooms. On a chain of connected rooms, an end room (Room without any other door other than the entrance) will not appear until the specified Min Chain is reached, for example if Min Chain is 3, the end room will only appear starting on the 3rd room of the chain.
int maxChain	The maximum number of rooms in a single chain of connected rooms. Once the specified Max Chain number of rooms is reached in a single chain, the nth room will be guaranteed to be an end room.
float roomSize	The size of the prefab rooms assigned by the user, do note each room should be shaped as a cube, with the same width, length and height.
float stairsProb	The probability of a room generated to be replaced by a stair room, which allows the user to climb to another floor.

Design of the Recursive Backtracking API

In this example, the dungeon is generated using a recursive function to create rooms. The process begins with the generation of a starting room and its doors. For this case, the west and north doors are initialized. The algorithm prioritizes the west door first, invoking the recursive function to generate a room on the west side.

As new rooms are generated, their chain index is incremented from the previous room. If the chain index is less than the minimum chain length (minChain, set to 3) and the total number of rooms has not exceeded the maximum room limit (maxRooms, set to 20), the room will always have at least one door. Once a room reaches the minimum chain length (minChain of 5), it will no longer generate additional doors.

If a room is designated as a stair room, the algorithm diverges slightly. Instead of immediately backtracking to the previous room, it first generates a new room directly above the stair room. This newly generated room follows the same rules as other rooms but is initialized with four doors instead of three and begins with a chain index of 0. Once this new room's generation is complete, the algorithm returns to the stair room and then backtracks to the preceding room from which it originated.

When a room runs out of available doors for further generation, the algorithm backtracks to the previous room and checks for any remaining unprocessed doors. If any doors are available, the function is called again to generate a room for that door. This process repeats until either no more doors are available (including in the starting room) or the maximum number of rooms (maxRooms) is reached.

API Tool Implementation

The Recursive Backtracking API is divided into three main sections:

- **Dungeon Layout Editor:** This section manages the layout generation process, allowing users to create the foundational structure of the dungeon.
- **Room Replacement Editor:** This section handles replacing the placeholder rooms in the layout with pre-designed rooms, enabling customization of the dungeon's appearance and functionality.
- **Save Dungeon:** This section allows users to save the generated dungeon, preserving its layout and room configurations for future use.
- For user convenience, tooltips are provided. If users are unsure about specific parameters or buttons, they can hover over them to view descriptive tooltips explaining their functionality.

Dungeon Layout Editor

In the Dungeon Layout Editor, users can adjust parameters to customize the dungeon's structure to their preferences. Once satisfied, they can click the Generate Layout button to initiate the layout generation process.

During generation, blocks are placed to visually represent the layout. Different block colours indicate room types:

- White blocks: Normal rooms
- Green blocks: Starting room
- Red blocks: End rooms
- Blue blocks: Stair rooms

If the generated layout is not satisfactory, users can click the Generate Layout button again (Figure 3), either with the same or updated parameters. This will destroy the current layout and replace it with a newly generated one. The process can be repeated as many times as needed until the desired layout is achieved.

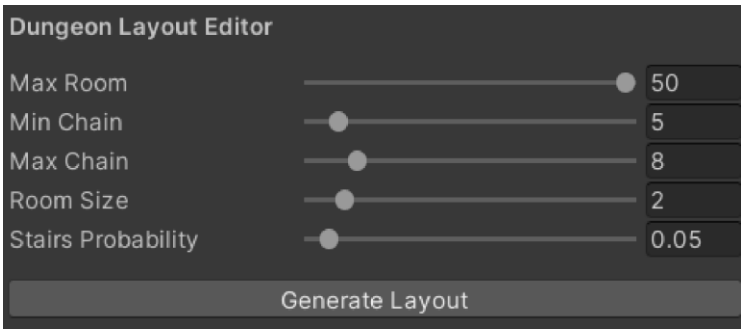


Figure 3. Parameter Used in Dungeon Layout Editor

Jagged-textured blocks indicate placeholder blocks generated to store data for the room replacement process. These placeholder blocks ensure that rooms generated during replacement will not overlap, as they are ignored when placing final room assets.

Room Replacement

The API (Figure 4) enables users to assign their pre-designed rooms, which are then randomly placed according to the layout generated by the API. However, stress testing revealed that each user-designed room should have a maximum of 120,000 triangles, particularly for dungeons with a high number of rooms. This limitation is essential because the total number of triangles is effectively multiplied by the number of rooms in the layout. Considering additional resource demands, such as physics colliders, adhering to this limit helps prevent high computational usage, which could otherwise lead to performance issues.

Using the API, pre-designed rooms are assigned to replace each cube in the generated layout. The API supports assigning multiple variations of rooms to a single room type, allowing for randomized replacement based on the room type. This process can be repeated with the same layout to produce different results, enabling users to refine the dungeon until the desired configuration is achieved.

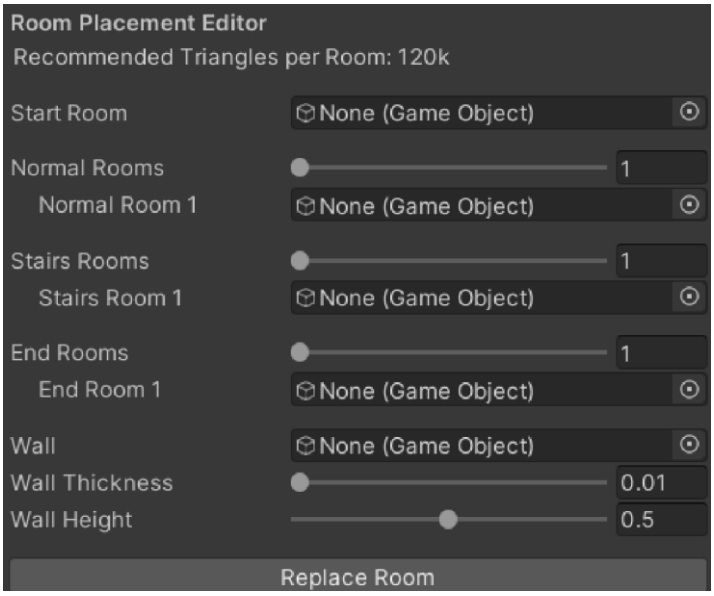


Figure 4. Parameter Used in Room Placement Editor

Saving the Dungeon

The generated dungeon can be saved as a prefab by clicking the Save Dungeon button. This action saves the current dungeon to the Dungeon folder, located at the path: Assets/Dungeon. Once saved, users can export the dungeon or incorporate it into a new scene within their project. This functionality allows seamless integration of the generated dungeon into any custom project or scene.

Results

Dungeon Layout Configuration

Table 4 shows the configuration for the dungeon layout, and Figure 5 displays the output under the parameter input in Table 4.

Table 4. Configuration for dungeon layout.

Test case	Dungeon Layout Type	maxRooms	minChain	maxChain	stairsProb
1	Large Dungeon	50	6	12	0.05
2	Small Dungeon	10	2	3	0.05
3	Vertical Dungeon	50	1	2	0.9
4	Single-Floor Dungeon	50	5	8	0
5	One-Way Dungeon	50	50	10	0
6	Consistent Shape Dungeon	50	5	6	0
7	Diverse Shape Dungeon	50	5	8	0

The dungeon layouts generated using the proposed API are visually represented with colour-coded blocks to distinguish room types: white blocks for normal rooms, green for starting rooms, red for end rooms, and blue for stairs. This intuitive representation provides a clear overview of the dungeon structure, aiding in the identification of room types within the layout. The Large Dungeon Configuration

(Figures 5a and 5b) demonstrates a high-density layout with diverse room connections across multiple floors, achieved using a moderate stairsProb setting to enable vertical progression. For the Multi-Floor Small Dungeon Layout (Figures 7c and 7d), a stairsProb of 0.05 generates staircases for multi-floor layouts, whereas a stairsProb of 0 restricts the layout to a single floor. The Vertical Dungeon Configuration (Figures 5e, 5f, and 5g) emphasizes intricate room connections across multiple floors with a high stairsProb value, promoting vertical exploration. In contrast, the Single-Floor Dungeon Configuration (Figures 5h and 5i) features a densely connected layout on a single floor due to a stairsProb of 0. The One-Way Dungeon Configuration (Figures 5j and 5k) limits backtracking by setting the minChain parameter higher than maxChain, resulting in linear room connections. The API is implemented in a way where minChain takes precedence over maxChain, and since users have the ability to input values for both minChain and maxChain, the API would essentially generate the layout that ignores maxChain completely if the minChain has a higher value than maxChain. Combined with setting the maxRoom to be the same or lower than minChain, it would guarantee the maximum number of rooms is reached once the first chain of rooms are generated, resulting in a one way dungeon. The Consistent-Shaped Chain Dungeon Configuration (Figures 5l and 7m) achieves minimal variation in room connections by maintaining a small gap between minChain and maxChain, creating a high-density layout. Lastly, the Diverse-Shaped Chain Dungeon Configuration (Figures 5o and 5p) produces varied layouts with unique dungeon shapes by setting significant differences between minChain and maxChain, resulting in diverse room connection patterns.

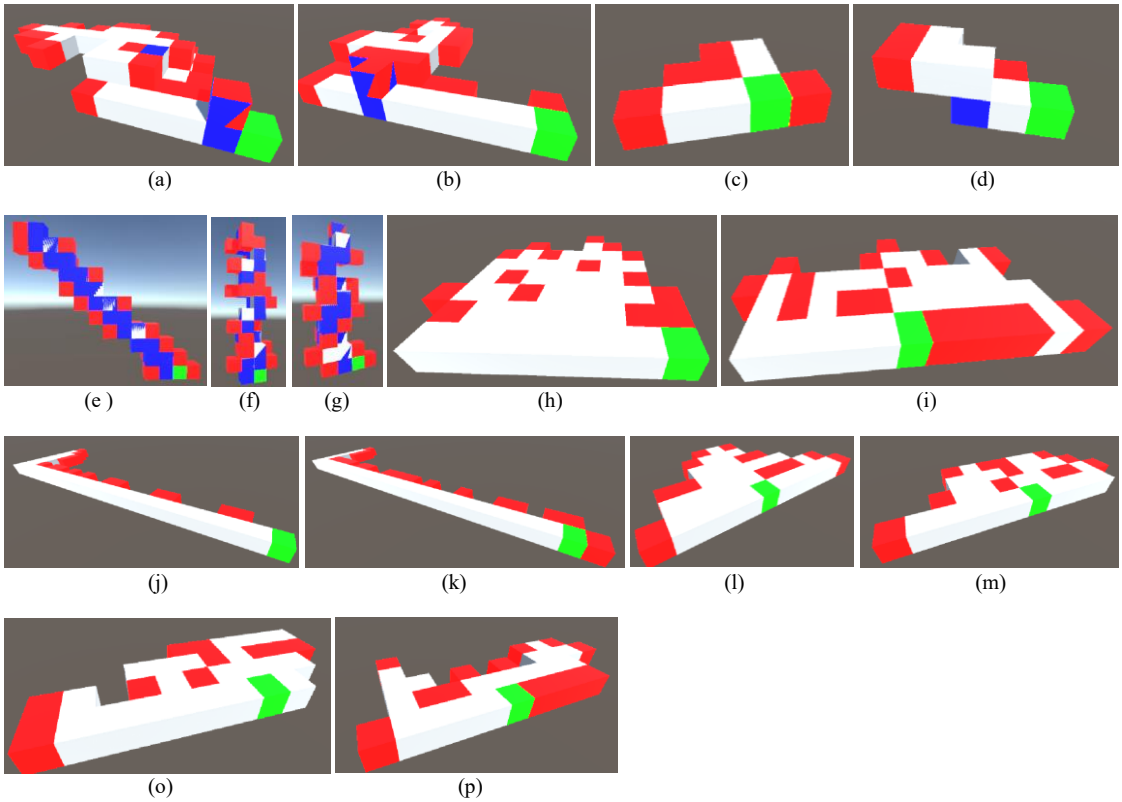


Figure 5. The 15 Screenshots from 7 Test Cases of Dungeon Output Generated using the API

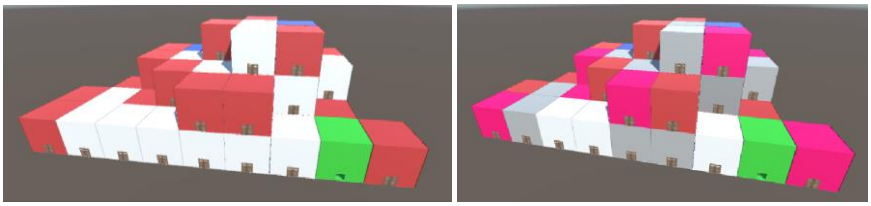
Room Placement Configuration

Room placement allows the customization of the room prefab according to the preference of the game developer. From Table 5, Singular type room placements are the room that assigns only one room prefab for each type of room, while multiple type room placements.

Table 5. Room placement configuration

Test case	Room Placement Type	Normal Room	Stair Room	End Room
8	Singular Type Room Placements	1	1	1
9	Multiple Type Room Placements	3	1	2

Singular Type Room Placements (Figure 6a) assign only one room prefab for each type of room. Multiple Type Room Placements (Figure 6b) assign multiple rooms for each type of room.



(a) *(b)*

Figure 6. Room Placement Configuration from 3 Test Cases

Chain Placement Configuration

Table 6 chain placement configuration. Min chains represent the minimum number of rooms to be placed in a chain while max chain represents the maximum number of rooms that can be placed in a chain.

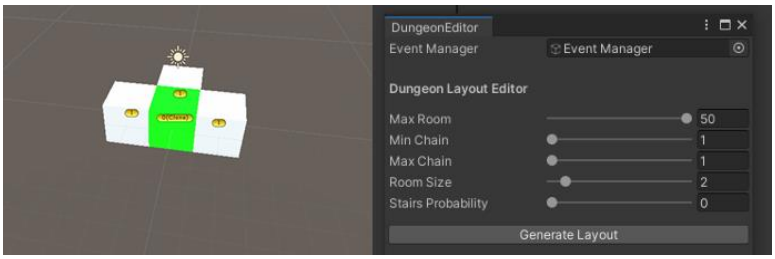
Table 6. Chain Placement Configuration from 3 test cases

Test case	Room Placement Type	Max Room	Min Chain	Max Chain	Room Size	Stairs Probability
10	Min and Max Chain Room Placements	50	1	1	2	0
11	Same Max Chain and Min Chain Placements	50	20	20	2	0
12	Same Max Chain and Min Chain Placements	50	5	5	2	0
13	All Max Placements	50	50	50	20	1

In the Min and Max Chain Room Placements configuration (Figure 7a), when both the MinChain and MaxChain parameters are set to 1, the dungeon will generate only one additional room, even if the MaxRoom parameter is set to 50. Occasionally, an end room may appear in this extra room due to the random placement logic.

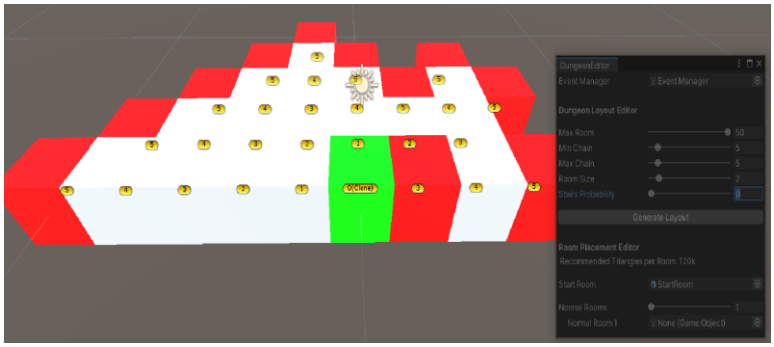
For the Same Max Chain and Min Chain Placements configuration (Figures 7b and 7c), when the MaxChain and MinChain parameters are equal, the resulting dungeon layout is relatively linear. For instance, setting both MaxChain and MinChain to 20 produces a straight dungeon with minimal branching.

In the All Max Placements configuration (Figure 7d), setting all parameters to their maximum values creates a significantly larger and more complex dungeon layout. The entire model scales up, and the layout includes additional staircases, leading to a more intricate multi-floor structure.



(a)


(b)


(c)

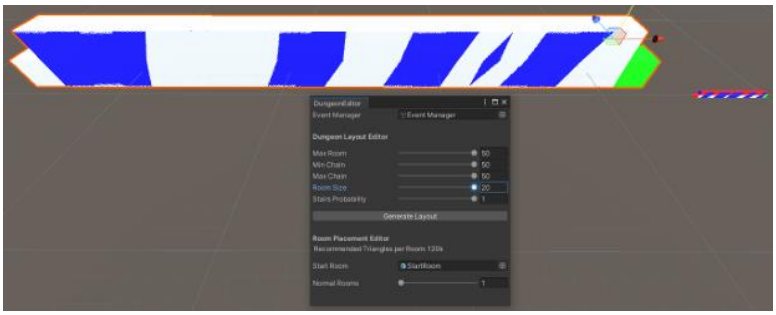
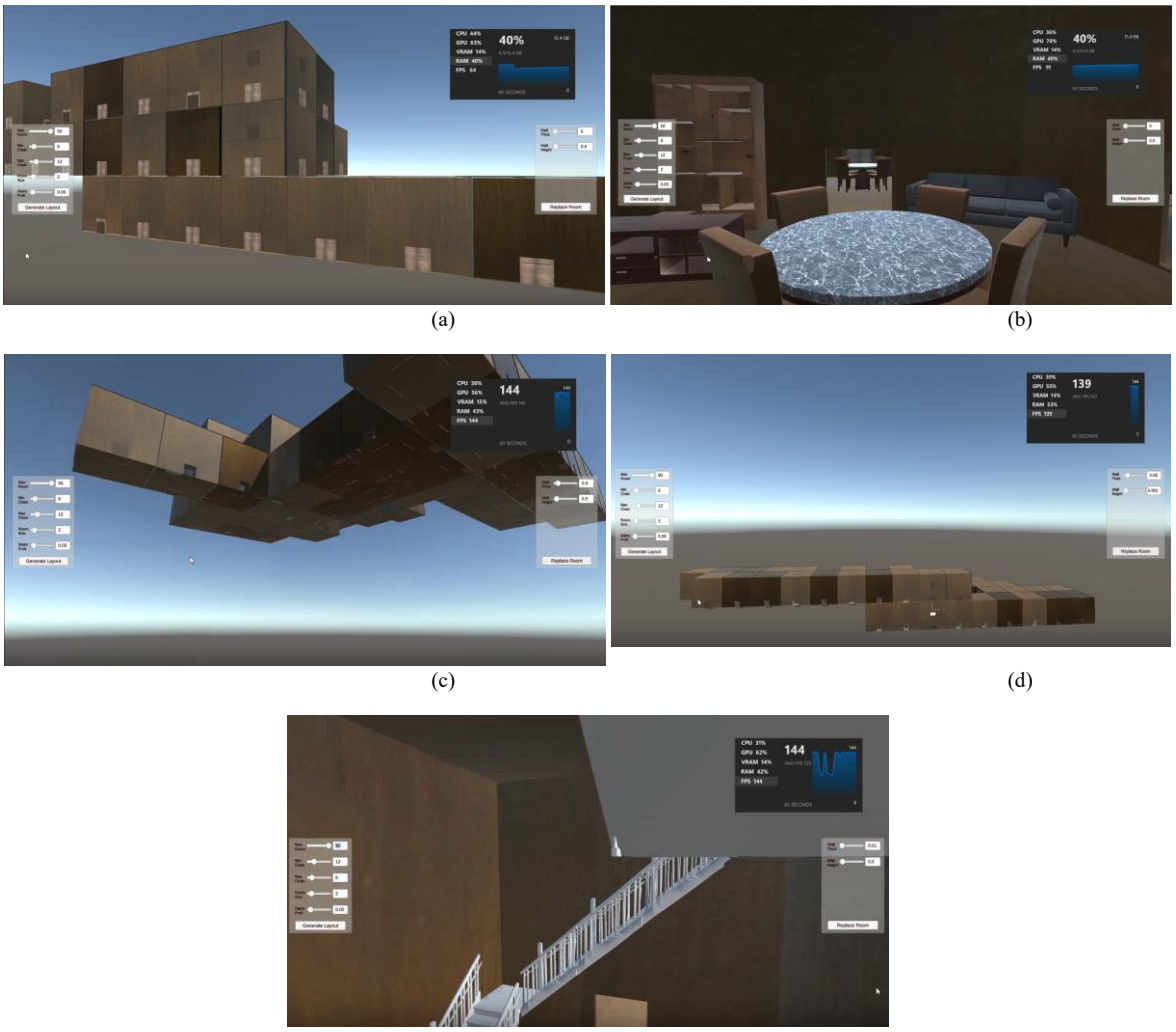

(d)

Figure 7. Min and Max Chain Room Placement Configuration

After replacing the placeholder blocks with pre-designed room prefabs, the dungeon generation is complete. Figure 8 showcases screenshots of the completed dungeon, including both indoor and outdoor environments, illustrating the transformation after room replacement.



(a) (b)

(c) (d)

(e)

Figure 8. Completed Dungeon

Performance Test Results

A performance test is conducted with the PC requirements below to identify the benchmark of the API. Below are the screenshots of the generated environment, indoor and outdoor, after being replaced with prefabs. The performance test is conducted using the gaming laptop with the following specifications.

- CPU -AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz Base Speed
- Memory 16.0GB with 3200 MHz
- GPU -NVIDIA GeForce GTX 1650Ti
- AMD Radeon ™ Graphics

Large dungeon's generation performance is being captured using Microsoft Game Performance software. The duration of the recorded footage was 1 minute. Table 7 shows the summary of the performance metric together with average value.

Table 7. Summary of the performance metric

Metric	Average Value
CPU usage (%)	27.642
GPU usage (%)	35.9189
Memory usage (%)	47.5926
VRAM (%)	14.1111
Framerate (FPS)	135.7407 (Peak value 144)

The average value of CPU usage at around 28% indicates that the API is not demanding and stable on the processor. Users can run additional background processes when using this API. Meanwhile, GPU usage around 36% indicates a moderate graphics load, with some peak values. This shows that certain parts of the room graphics are more high-consumption in GPU. Overall, the moderate graphic load indicates that the system can remain cool with the low risk of thermal throttling. As for the RAM usage at 47.59% and VRAM suggests that the API is moderate but below the threshold of graphic bottleneck. This indicates that the algorithm implementation is optimized for memory efficiency. High average fps at 135.74 with the peak value 144 suggests that the API is highly performant, rendering frames above the common standard of 60 fps, indicating smooth visuals and responsiveness. The FPS values are consistent over the time, suggesting that the application runs smoothly without significant frame drops. Users may expect a visually smooth gaming experience with minimum stuttering. In overall, the API performs efficiently across all metrics. However, it is suggested that the application runs on a mid-range hardware PC so that it leaves room for other applications to run concurrently.

Figure 9 to figure 11 shows the performance of CPU usage capture in 60 seconds of interval over the time.

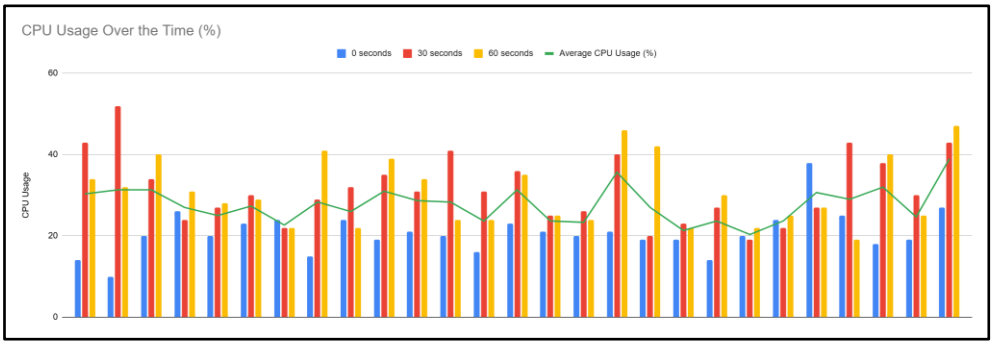


Figure 9. Average CPU Usage over Time

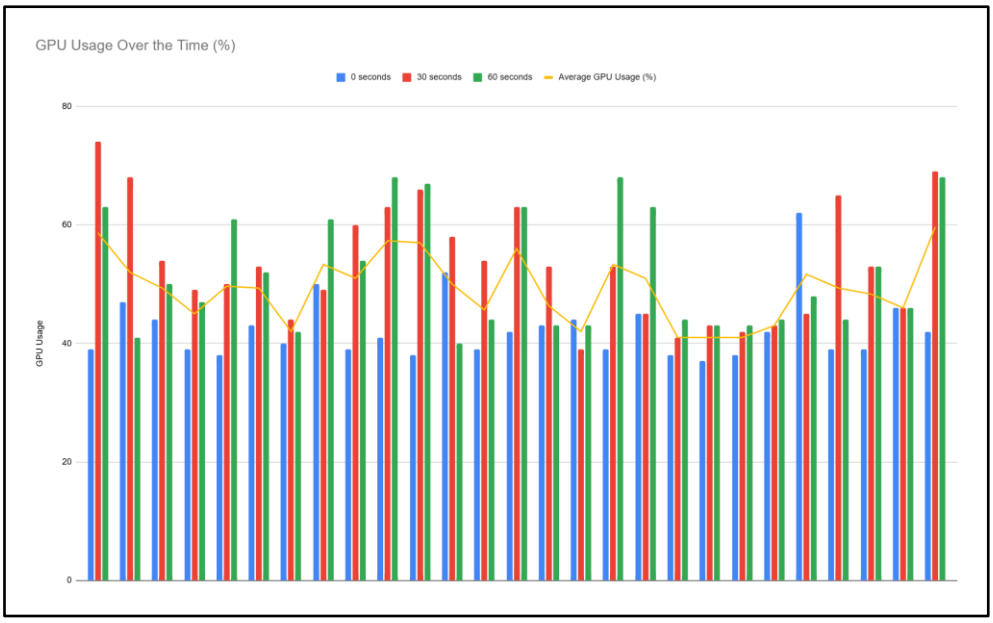


Figure 10. Average GPU Usage over Time

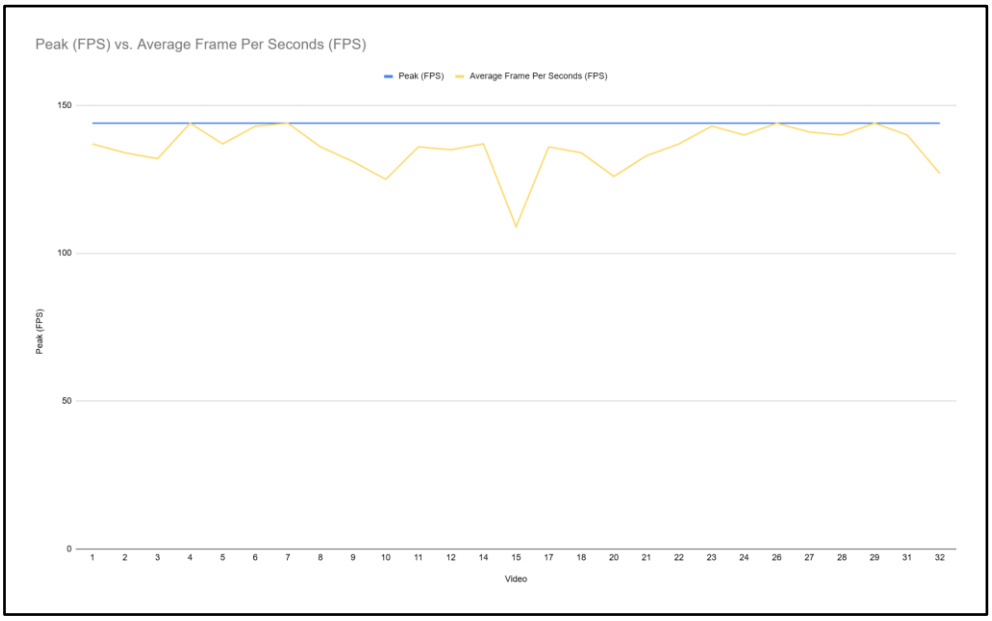


Figure 11. Peak Frame Per Seconds (FPS) vs Average Frame Per Seconds (FPS)

Conclusion

Recursive Backtracking has been thoroughly explored in this research to determine its implementation in an API designed to procedurally generate 3D room-based dungeons. This approach allows game developers to create dynamic, replayable content that enhances player engagement. By integrating customizable parameters, the API provides developers with the flexibility to tailor dungeon layouts to suit diverse game genres, ensuring that each dungeon is functional, distinct, and engaging. This level of adaptability empowers developers to optimize gameplay experiences while maintaining creative control over design aesthetics.

However, several limitations were identified during testing. While recursive backtracking ensures connectivity between rooms, it can lead to uniform layouts, particularly when using narrow minChain and maxChain ranges. This uniformity may limit aesthetic variability, potentially affecting the uniqueness of dungeon designs. Additionally, with complex configurations or high maxRoom settings, computational load increases significantly. To maintain performance, it is recommended that the triangle count for the 3D models remains below 120,000. Furthermore, random room placements can sometimes result in illogical layouts that detract from the overall coherence of the level design. To address this, additional constraints, such as thematic grouping or logical path progression, should be applied to ensure that layouts are both coherent and purposeful.

Looking forward, as technology continues to advance; the scale and complexity of video games will expand beyond what was previously imaginable. Procedural generation will play an increasingly vital role in enabling developers to create expansive, dynamic, and engaging content efficiently. By automating repetitive design tasks, developers can allocate more time to refining gameplay mechanics, narrative depth, and artistic elements.

To further enhance procedural generation, the integration of machine learning and artificial intelligence holds immense potential. These technologies can enable adaptive dungeon designs that learn from player behaviour and preferences, creating more personalized and immersive experiences. For instance, AI-driven algorithms could analyse gameplay data to dynamically adjust dungeon difficulty or suggest thematic elements that resonate with specific audiences. Developers are encouraged to explore hybrid approaches that combine recursive backtracking with other algorithms, such as cellular automata or graph-based methods, to diversify procedural outcomes and overcome current limitations.

In conclusion, while recursive backtracking offers a robust foundation for procedural dungeon generation, continued research and iterative refinement are essential to unlock its full potential. By addressing identified limitations and leveraging emerging technologies, game developers can create richer, more versatile procedural systems that elevate player experiences and set new standards in game design.

References

[1] de Pontes, R. G., Gomes, H. M., & Ritta Seabra, I. S. (2022). Particle Swarm Optimization for Procedural Content Generation in an Endless Platform Game. *Entertainment Computing, 43*, 100496. https://doi.org/10.1016/j.entcom.2022.100496

[2] Volz, V., Naujoks, B., Kerschke, P., & Tušar, T. (2023). Tools for Landscape Analysis of Optimisation Problems in Procedural Content Generation for Games. *Applied Soft Computing, 136*, 110121. https://doi.org/10.1016/j.asoc.2023.110121

[3] Viana, B. M. F., Pereira, L. T., Toledo, C. F. M., dos Santos, S. R., & Maia, S. M. D. M. (2022). Feasible–Infeasible Two-Population Genetic Algorithm to evolve dungeon levels with dependencies in barrier mechanics. *Applied Soft Computing, 119*, 108586. https://doi.org/10.1016/j.asoc.2022.108586

[4] Bellot, V., Cautrès, M., Favreau, J-M., Gonzalez-Thauvin, M., Lafourcade, P., Le Cornec, K., Mosnier, B., & Rivière-Wekstein, S. (2021). How to generate perfect mazes? *Information Sciences, 572*, 444–459. https://doi.org/10.1016/j.ins.2021.03.022

[5] Lan, X., Zhou, L., Lin, B., Li, J., & Lv, G. (2023). Rapid survey method for large-scale outdoor surveillance cameras using binary space partitioning. *ISPRS Journal of Photogrammetry and Remote Sensing, 207*, 57–73. https://doi.org/10.1016/j.isprsjprs.2023.11.017

[6] Li, Y., Liang, J., Yue, C., Yu, K., & Guo, H. (2023). An incremental random walk algorithm for sampling continuous fitness landscapes. *Neurocomputing, 553*, 126549. https://doi.org/10.1016/j.neucom.2023.126549

[7] da Rocha Franco, A. de O., de Carvalho, W. V., da Silva, J. W. F., Maia, J. G. R., & de Castro, M. F. (2024). Managing and controlling digital role-playing game elements: *A current state of affairs. Entertainment Computing, 51*, 100708. https://doi.org/10.1016/j.entcom.2024.100708

Acknowledgment

The authors gratefully acknowledge the use of the services and facilities of Tunku Abdul Rahman University of Management and Technology (TARUMT).

Funding Information

The research was done without funding.

Authors' Bio

Soh Zhen Shern is a graduate student of Tunku Abdul Rahman University of Management and Technology (TARUMT), with a Bachelor of Computer Science (Honours) in Interactive Software Technology. Has a passion for technology, and enjoys diving deep into the digital world, consuming media and discovering new ideas.

Dr. Tan Bee Sian is an Assistant Professor at Tunku Abdul Rahman University of Management and Technology, where she teaches game-based learning and game technology. She is a Unity Certified VR Developer and Unity Certified Programmer, with expertise in immersive technologies, game design, serious games, and game artificial intelligence (AI). Her work focuses on virtual and augmented reality for education and training, game algorithms and technologies, software development. She actively publishes in related fields and promotes interdisciplinary approaches in game-based and technology-enhanced learning.

Chong Kah Chun is a graduate in Computer Science (Interactive Software Technology) from Tunku Abdul Rahman University of Management and Technology (TAR UMT), Malaysia. His academic background and freelance experience cover areas of software development, programming, video editing, and digital design. His technical expertise includes C, C#, C++, Python, video editing, and Photoshop.

Ts. Ong Jia Hui is a lecturer in Computer Science at Tunku Abdul Rahman University of Management and Technology (TARUMT). He is passionate about exploring how AI and blockchain can enhance mobile application security. With a background in mobile development and fintech ecosystems, his work bridges academic research and real-world applications, with a focus on building trusted digital systems for Southeast Asia.

Ir. Ts. Dr. Chong Kim Soon is an Assistant Professor, Head of Department from the Faculty of Engineering, Technology and Built Environment, University College Sedaya International (UCSI). He has a Bachelor's degree in Electrical and Electronic Engineering, a Master degree of Science majoring in Electrical, Electronic and System Engineering and a PhD in Electrical, Electronic and System Engineering from Universiti Kebangsaan Malaysia. He leads numerous industry grants and serves as a principal investigator for industry consultation projects. His research area focuses on IoT, electronic engineering, games for health and biomedical devices. He is also a registered Professional Engineer from the Board of Engineers Malaysia (BEM), Professional Technologist from the Malaysia Board of Technologists (MBOT) and Certified HRDF Trainer.