International Journal of Creative Multimedia

From Algorithm to Playable Space: A Technical Note on BSP-Based Dungeon Design

Jing Ming Yap
yapjm-wm22@student.tarc.edu.my
Tunku Abdul Rahman University of Management & Technology, Malaysia

Nian Chein Woo
yapjm-wm22@student.tarc.edu.my
Tunku Abdul Rahman University of Management & Technology, Malaysia

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

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

Abstract

Procedural content generation (PCG) is widely used in game development to enhance replayability and reduce manual design effort. This paper presents a practice-oriented case study of a procedural dungeon generator implemented in Unity, using Binary Space Partitioning (BSP) as the core algorithm. The generator exposes adjustable parameters through the Unity Inspector, enabling designers to control dungeon dimensions, room density, offsets, corridor width, and decorative elements without altering code. To ensure reliable connectivity, a depth-first search (DFS)-inspired method is applied to corridor generation, overcoming instability found in standard BSP approaches. Additional features, such as decorations, torches, shrines, and enemy placement, illustrate how the generator can be extended to enrich atmosphere and gameplay. Importantly, the system separates generation logic from gameplay logic by working with prefabs rather than embedded scripts, providing developers with flexibility to adapt the tool to their own mechanics and project requirements. Demonstrations show how small parameter adjustments, particularly in corner modifiers and offsets, produce distinct dungeon layouts, ranging from dense and compact to spacious and exploration-oriented. The generator has also been applied in tutorial-level and regular gameplay scenarios, highlighting its adaptability. Overall, the

International Journal of Creative Multimedia (2025) 6, 2, 287-308
doi: https://doi.org/10.33093/ijcm.2025.6.2.16

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

system contributes as a practical prototyping tool for developers and an instructional resource for educators and students. By combining algorithmic clarity with implementation detail, it offers an accessible foundation for applying PCG in both development practice and teaching contexts.

Keywords Procedural Content Generation (PCG), Binary Space Partitioning (BSP), Unity Game Development, Dungeon Generation, Game Prototyping
Received: 4 May 2025, **Accepted:** 26 September 2025, **Published:** 30 September 2025

Introduction

Procedural generation is widely used in game development to reduce the workload of manual level design while enhancing replayability and variety. In particular, roguelike games have popularised the approach by creating new dungeons, enemies, and items each time a player begins a session. Early titles such as Beneath Apple Manor (1978) and Rogue (1980) demonstrated the value of algorithmically generated levels, and today the practice remains integral in both independent and large-scale game production. For educators and students, procedural content generation (PCG) offers a way to understand algorithmic design while developing flexible and reusable tools.

Among the many techniques available, the Binary Space Partitioning (BSP) algorithm has become a common method for dungeon generation. It works by recursively splitting a space into smaller subspaces, which can then be shaped into rooms and corridors. While the algorithm itself is well established, there is a practical need for accessible documentation that explains how BSP can be applied within modern development environments such as Unity, and how additional techniques such as depth-first search (DFS) and breadth-first search (BFS) can extend the results.

This paper presents a practice-oriented case study of BSP-based dungeon generation in Unity. Rather than proposing new algorithms, it focuses on clear pseudocode, implementation steps, and example outcomes. The contribution is intended for developers, educators, and students who seek to adapt procedural generation techniques for their own design projects, classroom exercises, or prototyping tasks.

Literature Review

Binary Space Partitioning (BSP) is a well-established approach to level and dungeon generation in games. Foundational studies, such as Clairbois (2006) on optimal partitioning, and Shaker et al. (2016) in Procedural Content Generation in Games, describe how BSP recursively divides spaces into subregions that can be adapted into rooms. More recent implementations, including Putra et al. (2023), demonstrate BSP in two-dimensional dungeon layouts and highlight its compatibility with other generative methods such as L-systems. These references provide the basis for applying BSP in contemporary development environments.

Beyond BSP, other approaches to procedural generation have been widely examined. Baron (2017) compared multiple dungeon generation techniques, noting the strengths and limitations of different corridor-connection methods. Silva et al. (2023) and Oyhenard (2023) explored more recent machine-learning and design pattern applications, showing how PCG techniques can vary in complexity and creative scope. While these works broaden the theoretical landscape, they also underscore the

continuing relevance of straightforward partitioning methods when clarity and reproducibility are prioritised.

Supporting algorithms play an important role in enhancing BSP-based generation. Elkari et al. (2024) and Jain (2023) provided comparative discussions of depth-first search (DFS) and breadth-first search (BFS), which are integrated here to manage corridor generation and boss-room identification. Complementary perspectives, such as Baldwin et al. (2017) on design patterns, also suggest how generative tools can be aligned with gameplay logic.

Taken together, these studies demonstrate both the maturity and flexibility of procedural generation research. This paper builds on these foundations by offering a practice-oriented case study of BSP in Unity, focusing on implementation clarity and reproducibility for developers, educators, and students.

Implementation

Binary Space Partitioning

The first stage in generating a dungeon layout is dividing the entire map area into smaller, manageable spaces. The Binary Space Partitioning (BSP) algorithm provides a straightforward way to achieve this. Starting from a single rectangular space, the algorithm recursively splits the area along randomly selected horizontal or vertical lines. Each split creates two subspaces, which may themselves be further divided until user-defined limits are reached.

In this implementation, the splitting process is controlled by parameters such as maximum iterations, minimum space width, and minimum space length. These values allow designers to influence the number of partitions and the eventual size of the rooms. For example, increasing the maximum iterations leads to a denser dungeon with more potential rooms, while larger minimum sizes prevent overly small spaces that cannot hold meaningful content.

The algorithm terminates when no further splits are possible, either because the iteration count is exhausted or the remaining space is below the defined threshold. Each final partition is stored in a list of potential room spaces. Figure 1 illustrates the conditions under which a space can be split, while Figure 2 presents pseudocode for the procedure.

```
node.RoomWidth >= minRoomWidth * 2 || node.RoomLength >= minRoomLength * 2
```

Figure 1. Conditions for Space Partition Eligibility

The partitioning algorithm will stop splitting the space when it runs out of iteration that is defined by the user or the space for splitting is too small to be split. When a space can no longer be split, it will be given a partitioned space number, and this space will be stored in a list of partitioned spaces for future references. Below is the pseudocode for reference.

```
START
        Initialize the space width, length, max iterations and minimum space width and length
        Pass the space to a loop/recursive function
        IF space eligible to split AND iteration is larger than 0
        Check the length and width to determine orientation, get a random coordinate for line
        Split the space based on the line, save to a list, pick one space to repeat
        ELSE
        Add the space node to a list, pick another space to split
        RETURN when no iteration is left/no space left in list, add any remaining space to return list
END
```

Figure 2. Pseudocode for Binary Space Partitioning

By exposing these parameters to Unity's inspector, developers and educators can experiment with values in real time, observing how the partitioning directly affects dungeon structure. This makes BSP not only a practical generation method, but also an effective teaching tool for understanding recursive algorithms in game design.

Spawn Room

Before generating the remaining dungeon rooms, a dedicated spawn room is created to serve as the player's starting point. Randomisation is used to vary its location and size, ensuring that each run of the generator provides a different opening scenario. In Unity, this is achieved by applying the built-in Random.Range function, which selects values for both size and placement within predefined limits.

In the example implementation, the spawn room size is constrained between 8 and 12 units. Once these values are set, the system determines the exact position by selecting one of four possible quadrants around the dungeon area (top, bottom, left, or right). Figure 3 shows how this placement is determined, while Figure 4 provides sample Unity code for the calculation.

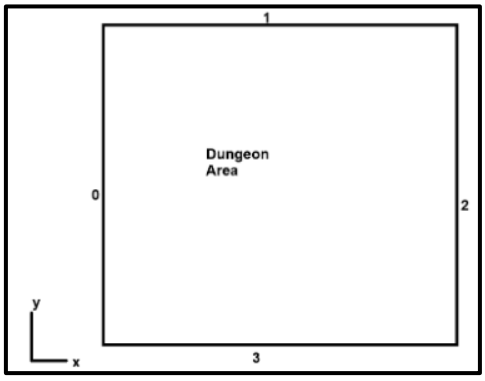

Figure 3. Visual of Spawn Room Position Relative to Dungeon Area

```
spawnRoomPosition = new Vector2(
Random.Range(transform.position.x + 8, dungeonAreaLength − 8),
    Random.Range(dungeonAreaWidth + 10, dungeonAreaWidth + 20));
```

Figure 4. Code for Spawn Room Position Calculation

By enforcing a controlled randomisation process, the generator guarantees that the spawn room always has sufficient space to connect to corridors, even in the smallest case. The logic is summarised in Figure 5 as pseudocode.

```
START
        Initialize the size, location and position of the spawn room
        Randomize the size and location
        Calculate the actual position
END
```

Figure 5. Pseudocode for Spawn Room Creation

This approach highlights a practical balance between variety and reliability: developers can achieve a different spawn layout with each execution while avoiding disconnected or unplayable starting points. For educators, the spawn room logic also illustrates how parameter ranges and conditional placement can be combined to achieve predictable yet flexible outcomes in procedural generation.

Generate Rooms

Once the dungeon area has been partitioned, the next step is to define playable rooms within those spaces. The partitioned regions created by BSP are often tightly packed and need adjustment before they can function as distinct rooms. The room generation algorithm addresses this by applying two key parameters: the corner modifier and the room offset.

The corner modifier introduces controlled randomness into the room's size and placement. By adjusting this value, designers can influence whether rooms appear larger, smaller, or shifted within their allocated partition. The room offset, on the other hand, determines the gap between the room and the edge of its partition, ensuring that adjacent rooms do not overlap and that corridors can be added later. Together, these parameters allow the overall dungeon layout to range from compact and clustered to more open and widely spaced.

The algorithm begins by inserting the spawn room into a list of room nodes. It then iterates through each partitioned space, calculating the coordinates of the bottom-left and top-right corners. Using these boundaries, the offset and modifier are applied to define the final dimensions of the room.

Figure 6 provides an example of the calculation in Unity, where Random.Range is used to vary room size within the permitted bounds.

```
var minX = boundaryLeftPoint.x + roomOffset;
var maxX = boundaryRightPoint.x - roomOffset;
Random.Range(minX, (int) (minX + (maxX - minX) * roomBottomCornerModifier))
```

Figure 6. Code to Generate Room Space

After processing all partitions, the list of room nodes is returned, with the spawn room always preserved as the first entry for consistency. Figure 7 presents the pseudocode summarising this process.

```
START
        Get the room spaces from Binary Space Partitioning.
        Calculate coordinates for bottom left and top right with modifier and offset
        Repeat for all the room spaces.
        Return the list of room.
END
```

Figure 7. Pseudocode for Room Generator

From a practice perspective, this approach gives developers and students a clear way to balance structure and variation in dungeon design. By adjusting just two parameters, it becomes possible to generate layouts that range from highly compact to widely spaced, offering a flexible tool for both experimentation and teaching demonstrations.

Generate Corridor

Corridors provide the primary means of navigating between rooms, and their generation is essential to making a dungeon playable. In BSP-based layouts, a common approach is to connect sibling nodes from the partition tree. However, this often produces unstable results, such as corridors that pass through existing rooms or leave parts of the dungeon disconnected.

To improve reliability, this implementation applies a depth-first search (DFS)-inspired method for corridor generation. Instead of directly linking child nodes, the algorithm progressively searches for valid connections from one room to the next. Each room is connected in turn, and when a dead end is encountered, the algorithm backtracks to a previous room to continue the search. This ensures that all rooms are connected, avoiding isolated spaces and creating a more consistent dungeon structure.

Practically, this is managed with two lists:

- nodesToGenerateCorridor - tracks the current room attempting to connect.
- nodesToCheckCorridorConnection - holds rooms still available for linking.

The algorithm selects a random room from the first list, chooses a direction, and searches for the nearest eligible neighbour. If found, a corridor is generated, and the new room becomes the current node. If no valid connections are identified in all four directions, the process backtracks until another viable link can be created.

Figure 8 shows pseudocode for this approach. By adapting the standard BSP process with DFS-style traversal, designers gain greater control over dungeon connectivity. For educators, this also illustrates how classic search algorithms can be repurposed for practical level design challenges.

```
START
        Select a random room from the room list
        Select a random direction to search for nearest room
        If a room is found, corridor is generated, and the connected room is used to repeat
        If no room is found, it will repeat the process, the direction into a list to avoid
        If no room is found in four directions, it will backtrack to the last room and repeat step 2
        If all room is connected, it ends the recursion
END
```

Figure 8. Pseudocode for Corridor Generator

Generate Floor

After all the rooms and corridors is generated, but they only exist as data within their respective structures. The first element to bring to life is the floor, as it forms the foundation of the dungeon and gives a clear outline of its structure. While instantiating prefabs for each floor tile is common, this approach can be inefficient due to varying room and corridor sizes. A better method is to generate the dungeon floor as a single mesh directly through a C# script. By generating a mesh, it can handle floors of varying sizes more easily and improve performance by reducing the number of instantiated objects. Before diving into the script, it's important to understand how meshes are defined in Unity. Refer to the figure below:

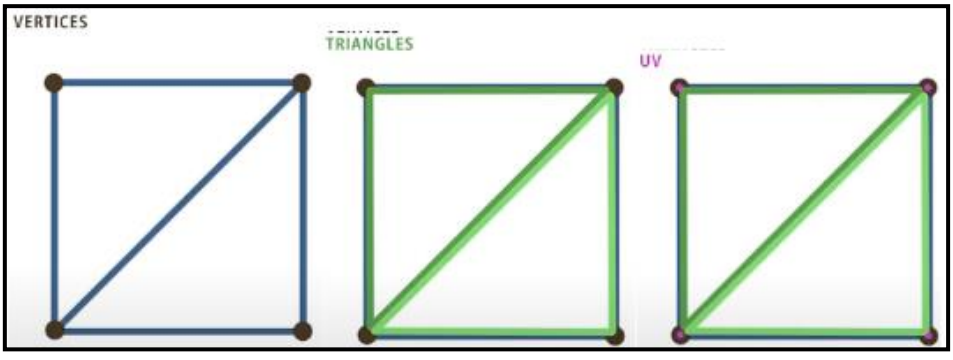

Figure 9. Three Key Element of Meshes

A mesh in Unity is composed of three key elements: vertices, triangles, and UVs. To break it down simply, vertices are vectors representing the positions of points in 3D space, triangles are integers

that define how these vertices are connected to form surfaces, and UVs are vectors that dictate how textures are applied to the mesh. In Unity's mesh system, the vertices, triangles, and UVs are stored in arrays. It's crucial to remember that the order of these elements is important, incorrect sequencing can result in the mesh being misaligned or not displayed properly. To ensure the mesh is correctly oriented towards the camera and properly textured, start by initializing the vertices array by reading the Vector2Int values of each corner from the node data and mapping the y-coordinate of the Vector2Int to the z-component of the Vector3. In this setup, index 0 of the array corresponds to the top-left corner, index 1 to the top-right, index 2 to the bottom-left, and index 3 to the bottom-right. This setup is depicted in the diagram below:

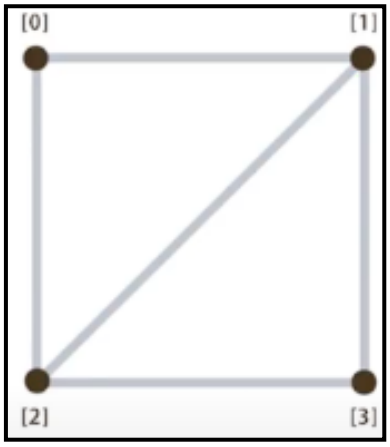

Figure 10. Index of Vertices

Next, define the array of triangles, which is an integer array. The order of the elements in this array is critical, as they must follow a clockwise sequence to ensure the mesh faces the camera. In this case, the array values should be 0, 1, 2, 2, 1, 3, which represents two triangles that form the quad. This arrangement ensures the mesh is correctly oriented towards the camera. Lastly, define the UVs, which are stored in an array of Vector2 values. This array must have the same number of elements as the vertices array. The UV coordinates map the texture onto the mesh, so the order of the UVs should match the corresponding indices in the vertices array. In a UV mapping system, the bottom-left corner is represented by (0,0), and the top-right corner is represented by (1,1). For this specific scenario, the UVs should be assigned as follows: index 0 is (1,0), index 1 is (1,1), index 2 is (0,0), and index 3 is (0,1).

Now that the mesh is defined, it still doesn't exist in the game world as a visible object. To bring it into the scene, create a new *GameObject* with three components, *MeshFilter, MeshRendere* and *MeshCollider*. The mesh created earlier will be assigned to *MeshFilter.mesh* and *MeshCollider.sharedMesh*, while the floor material will be assigned to *MeshRenderer.material*. Figure 11 shows pseudocode for this approach.

```
START
        Create three array vertices (Vector3), triangles (Int) and uv (Vector2)
        Assign value to the arrays (refer to section above)
        Create a new Mesh and assign the arrays to component vertices, triangles and uv respectively
        Create a new GameObject with MeshFilter, MeshRenderer and MeshCollider
        Assign the new Mesh to MeshFilter.mesh and MeshCollider.sharedMesh,
        Floor material to MeshRenderer.material
END
```

Figure 11. Pseudocode for Floor Generator

Placing Walls and Doors

Walls are a crucial structure in a dungeon, providing boundaries for rooms and corridors, so creating a basic empty dungeon layout must include placing walls accurately. Implementing walls is straightforward since there is list of RoomNode and CorridorNode structures that provide the coordinates of their corners, using a loop to iterate through it to get the positions, and save them to two list representing different orientation is an option. For example, the top is handled with a loop that iterates from the top-left corner's x component to the top-right corner's x component, effectively covering the top boundary of the room. This process is repeated for other sides with respective corners and components.

However, this approach did not consider future door placement. To determine the location of a door, cross check the list of wall edge and corridor edge. As it loops through a room's walls, if a point on the wall already exists in the list, that indicates a door should be placed there. In such cases, the point is removed from wall list and added to door list.

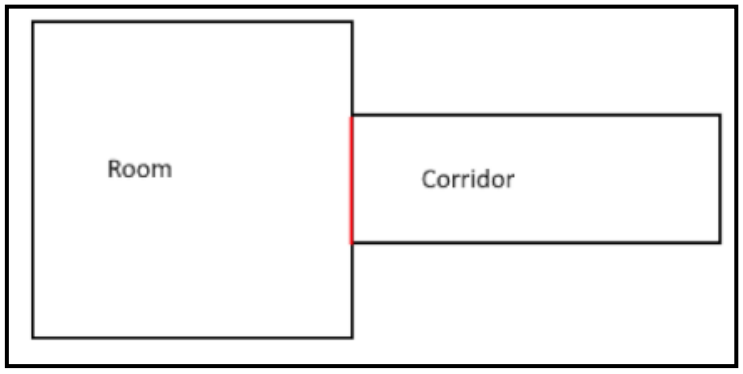

Figure 12. Connection of Room and Corridor

With the four lists of vertical and horizontal positions for walls and doors ready, simply use a loop to iterate through these lists and instantiate the corresponding wall and door prefabs in the dungeon. This basic implementation works well, but it's important to keep in mind that adjustments may be needed depending on the specific prefabs used for walls and doors, as their size and pivot points can affect placement. The code will need to be adjusted based on specific scenario. Below is the pseudocode.

```
START
        Create four lists of Vector3 to store wall position and door position for both orientation
        Go through the top, bottom, left and right of the room, and add them to the respective list.
        If there is no same Vector stored in the list, it was added to list
        If there is a same Vector stored, it is removed from the wall list and stored in the door list
        Go through the four lists and Instantiate respective prefab in different rotations.
END
```

Figure 13. Pseudocode for Walls and Doors Generator

Placing Decorations

Decorations may be optional from a gameplay perspective, but they play a crucial role in making the dungeon feel more dynamic and immersive. To implement decoration placement during procedural generation, it could randomly select a position to place it. It does create two issues, how to ensure that no overlap happen, and what is the appropriate number to place? To address the first issue, the room is mapped to a 2D Boolean array, which resets for each new room. As for the quantity to place, there is a function that identifies the smallest room in the dungeon, which serves as a reference for calculating how many decorations each room should receive.

If the current room is smaller than the reference size, it's likely a corridor, so it will only place 0 or 1 decoration at most. For larger rooms, calculate a size modifier based on the square root of the total dungeon area divided by the smallest room's area. If the room's size exceeds this modified threshold, place the maximum number of decorations allowed. Otherwise, select a random number between the minimum and maximum values for the room. Once the number is set, the actual placement begins. The room or corridor is represented by a 2D boolean array, initially set to all false. This array is two units smaller in both dimensions to ensure decorations aren't placed on walls or doors. Then, select a random position in the array for each decoration, mapping these positions back to their respective spots in the room. If a selected position is already true (occupied by another decoration), continue looping until a valid position is found. Below is the pseudocode:

```
START
        Get the smallestRoomSize
        Get the room to place decorations
        Initialize a 2D boolean array to false with the room dimension
        Calculate room size
        Calculate a modifier to determine which room is big or small
        Determine decoration quantity based on result
        Use a loop to place decoration
        Select a random position in the array that is false
        Calculate the actual position based on the array, instantiate the decoration
        Set the position to true
        Repeat
END
```

Figure 14. Pseudocode for Decorations Generator

Special Decorations

When a special type of decoration serves a specific gameplay purpose, it should be handled separately from regular decorations. For example, if you want to place a shrine or statue exactly in the centre of each room, the placement function for such decorations should be independent, directly calculating the centre point of the room for instantiation. If exactly one of them in the middle of each room:

```
START
        Get the prefab
        Calculate the middle point of current room
        Instantiate the prefab
        Proceed to next room and repeat
END
```

Figure 15. Pseudocode Example for A Special Decoration in the Center

Placing Torches

Even if real-time lighting isn't applied, present of torch can make the dungeon feel more atmospheric. Unlike other decorations, torches are placed directly on walls and have four possible orientations—left, right, top, or bottom—depending on the wall's position. To implement this, a function is built upon previous method for checking wall positions by adding a function that determines the orientation of the wall relative to the room node (left, right, top, or bottom). This can be easily achieved by creating four new lists that represent each side of the room and use a function to write the position in when checking the respective side of the room. Additionally, cross-check these lists with the existing door lists to avoid placing torches where doors exist.

Once the position is confirmed, the placement algorithm could be implemented, which basically just iterates through all four lists and instantiates the torch prefabs. When instantiating the torch prefab, adjust its rotation based on the wall it's placed on. For example, a torch placed on the top wall should face opposite direction. To add further variety and make the dungeon feel more dynamic, variants of torch could be added, such as unlit torches. Additionally, when iterating through the list of positions, varying the spacing between torch might result in more natural appearance.

```
START
        For each node, go through all the position of top, bottom, left and right and save to list
        Go through each position list and place the torch.
        Select a random prefab in the torch list
        If the position is occupied in door list, it will skip this iteration
        Based on which side of wall is placed, set the rotation accordingly
        Instantiate the torch with the prefab, position, rotation
END
```

Figure 16. Pseudocode for Torch Placement

Determine Boss Room

Usually, there will be a boss fight in most of the dungeon games and the boss room usually will be in the deepest of the dungeon. To achieve this, a path finding algorithm is implemented to find out the cost of each room by using breadth-first search. The room with the largest cost will be the boss room.

First, the algorithm will loop through the corridor list to calculate the cost of each corridor which is the same as the length of each corridor. Next, build a graph which contains all the dungeon rooms that are connected by corridors. The corridor will be the edge of the graph as each of them will connect with 2 rooms. After creating the graph, a breadth-first search will be carried out to search for the room with greatest cost as the example in figure below.

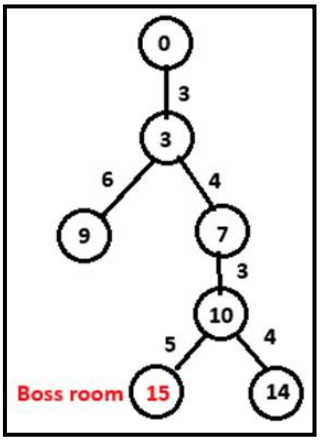

Figure 17. Searching of Boss Room

After the boss room is found out, it will be stored in a boss room variable for the usage in spawning enemies. Below is the pseudocode:

```
START
        Calculate the length of corridor and assign it to corridor nodeValue
        Build the graph and add path node and edge
        Calculate the total path cost for each room node
        Get the BossRoom which is the room with the highest path cost.
END
```

Figure 18. Pseudocode for Finding Boss Room

Spawning Enemies

Enemies are the primary challenges in a dungeon, they can be roughly categorized into three types: weak but common normal enemies, fewer but stronger elite enemies, and powerful, unique bosses. It shares similarities with the placement of decorations, particularly in ensuring that enemies do not overlap with one another and that a specified number of enemies are spawned in each room. Thus, mapping the room to a 2D Boolean array and use it as a reference will prevent overlaps. The number of

enemies can also be determined based on the room's size. From the Enemy class, an enum is used to classify enemies into three types: Normal, Elite, and Boss. Before spawning begins, the room's size is used to calculate a "room value." When an enemy is spawned, its respective integer value is subtracted from this room value, except in the case of bosses, which follow a different spawning logic. Once the room value reaches 0 or below, enemy spawning for that room is complete, and the next room can be processed.

Just like the decoration placement, first, map the room to a 2D boolean array, initializing all positions to false. The room value is calculated, and a while loop is used to spawn enemies while the room value is greater than 0. In each iteration, a random enemy is selected from a predefined list, and a random location is chosen for placement. If the location is already occupied, as indicated by a true value in the boolean array, a new location is selected. Once an enemy is successfully placed, the room value is reduced by the enemy's associated value. Boss rooms, however, are handled separately by simply placing a single boss in the center of the room. Refer to the pseudocode in Figure 19.

```
START
        Loop through list of room to place enemy
        If it is the boss room, spawn only the boss and skip this iteration
        Initialize a 2D boolean array based on the room dimension to false
        Calculate the room value
        While room value greater than 0
        Select a random position in the array that is false
        Calculate the actual position based on the array, instantiate the enemy
        Set the position to true
        Subtract the room value based on enemy type
END
```

Figure 19. Pseudocode for Spawning Enemies

Result

The procedural dungeon generator was implemented in Unity, with all key parameters exposed in the Inspector interface to make the system flexible and transparent for use in different projects. As shown in Figure 20, designers can directly adjust values such as dungeon dimensions, minimum room size, maximum BSP iterations, corner modifiers, room offset, and corridor width.

In addition, the interface provides slots for assigning materials, wall and door prefabs, and optional features such as shrines, torches, decorations, and enemies. This arrangement not only facilitates experimentation but also makes the system suitable for teaching purposes, as students can adjust parameters and immediately observe their effects (Figure 21).

In practice, the most influential parameters for shaping dungeon layouts were found to be the corner modifiers and offset values. While parameters such as dungeon size, corridor width, and prefab

selection have predictable outcomes, corner and offset values introduce meaningful variation to the spatial structure. To illustrate this, several demonstrations were carried out using different parameter sets. These are not controlled experiments, but practice-oriented examples intended to show how the generator behaves under different conditions.

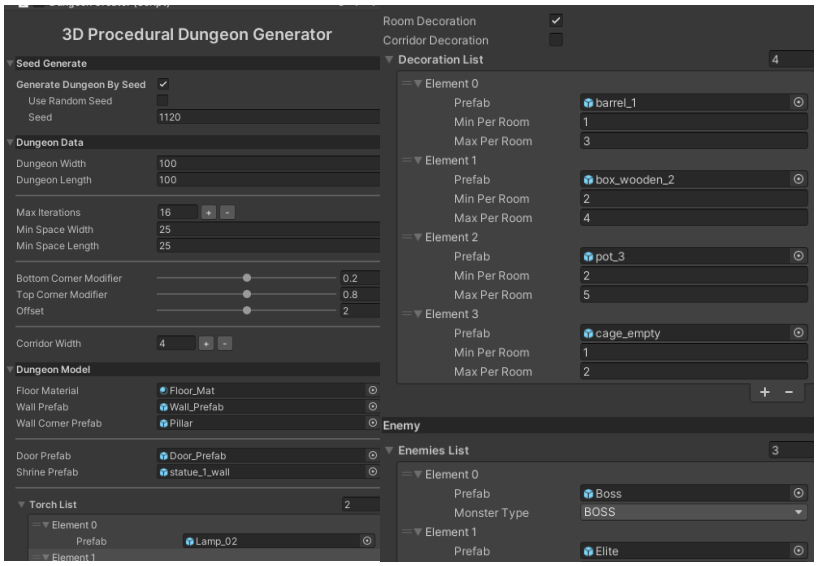

Figure 20. Unity Inspector Interface for Dungeon Generator

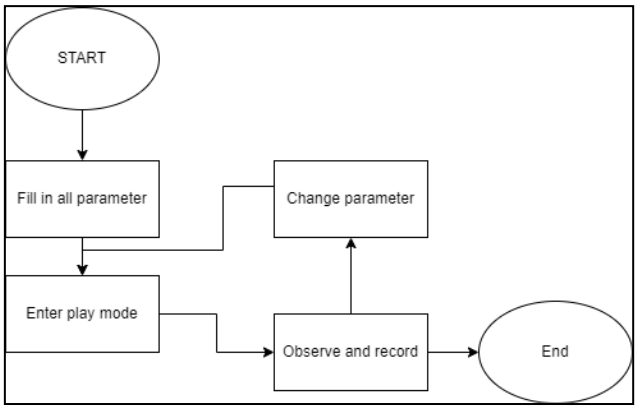

Figure 21. Flow of Testing

Table 1. Test data for generator

Parameter Name	Data
Bottom Corner Modifier	0.2
Top Corner Modifier	0.8
Offset	2
Corridor Width	4

In the first demonstration, values were set at 0.2 for the bottom corner modifier, 0.8 for the top corner modifier, an offset of 2, and a corridor width of 4 (Table 1). With these inputs, the generator produced a dungeon containing 12 rooms in addition to the spawn room, distributed across the layout

with moderate spacing. Figure 22 illustrates this configuration. The outcome shows a balanced layout where rooms have clear separation yet remain easily connected by corridors.

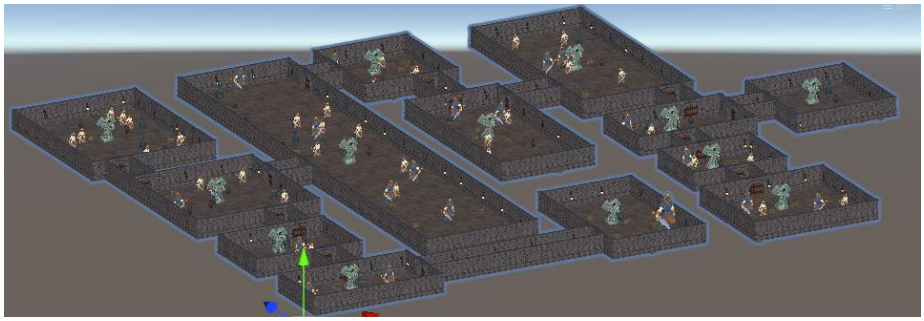

Figure 22. Result of First Test

A second demonstration used more extreme corner modifier values, set to 0.1 and 0.9, combined with a smaller offset value of 1 (Table 2). This resulted in a compact dungeon where rooms were tightly grouped, leaving minimal gaps between them (Figure 23).

Table 2. Test data for generator: More extreme corner modifiers, smaller offset

Parameter Name	Data
Bottom Corner Modifier	0.1
Top Corner Modifier	0.9
Offset	1
Corridor Width	4

From a design perspective, this produces a denser navigational experience, with players moving quickly between adjacent rooms. Such a layout may be useful for fast-paced gameplay or scenarios where frequent encounters are intended.

In a third case, the corner modifiers were adjusted to 0.3 and 0.7, with a larger offset value of 3 (Table 3). This produced a more open dungeon in which some rooms appeared smaller and separated by wider gaps (Figure 24).

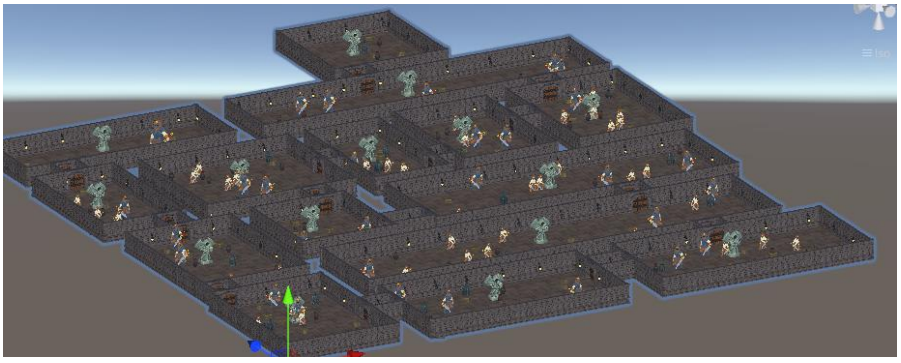

Figure 23. Result: More Extreme Corner Modifiers, Smaller Offset

To further understand how those parameters affect the generation, another extreme of data is prepared, refer to Table 3:

Table 3. Test Data for Generator: Less Extreme Corner Modifiers, Larger Offset

Parameter Name	Data
Bottom Corner Modifier	0.3
Top Corner Modifier	0.7
Offset	3
Corridor Width	4

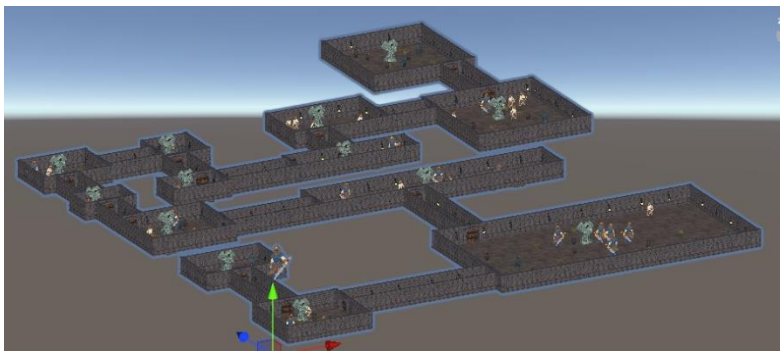

Figure 24. Result: Less Extreme Corner Modifiers, Larger Offset

This variation demonstrates how offset values directly affect spacing: increasing the offset enlarges the buffer between rooms, thereby reducing density and giving corridors more prominence. From a practical standpoint, this configuration could support exploration-focused gameplay, where travel between rooms contributes to pacing and atmosphere.

These three demonstrations highlight how small adjustments to a handful of parameters can lead to noticeably different dungeon layouts. For developers, this provides a simple but powerful mechanism to tailor level structure without rewriting code. For educators, it offers a way to show students how algorithmic parameters function as design levers, linking computational rules to visual and spatial outcomes.

Beyond these parameter-focused examples, the generator was also tested in different gameplay scenarios to illustrate its adaptability. By limiting BSP iterations, the generator can create very small layouts suitable for tutorial levels. For instance, constraining the system to only two rooms with a fixed random seed produces a repeatable configuration where explanatory text or scripted events can be placed in known positions (Figures 25–27). This demonstrates that procedural generation can support not only replayable dungeons but also controlled learning environments.

Figure 25. The Spawn Room in the Tutorial Level

Figure 26. A Regular Room in the Tutorial Level

Figure 27. The Boss Room in the Tutorial Level

In a regular gameplay scene, the generator produced a standard dungeon crawl, beginning from the spawn room and progressing through a sequence of connected rooms. Corridors generated with the DFS-based method ensured that no rooms were isolated, and BFS logic successfully identified the boss room at the farthest point from the starting location. Figures 28 and 29 show typical examples of this progression. From a practice standpoint, the outcome demonstrates how the generator can be incorporated into conventional game loops without requiring additional manual level design.

Figure 28. The Spawn Room in Regular Gameplay

Figure 29. A Regular Room in Regular Gameplay

It is important to note that these results are presented as demonstrations of capability rather than formal evaluations. The intention is not to measure performance or novelty against other algorithms but to show how the system can be configured, adapted, and applied in practice. For developers, this means the generator can serve as a flexible prototyping tool, accelerating early-stage design work. For educators, it can function as a teaching resource that illustrates how algorithmic design decisions translate into playable spaces.

Overall, the generator demonstrates three key points of value. First, the exposed parameters make it easy to adjust dungeon density, spacing, and structure, allowing different gameplay experiences to be created from the same core algorithm. Second, the separation of generation logic from gameplay logic ensures that the tool can be integrated into a variety of projects without enforcing specific mechanics. Finally, the visual clarity of the results, supported by Unity's real-time interface, makes the generator especially suitable for instructional use in courses on game development, algorithms, or procedural design.

Conclusions

This paper has presented a practice-oriented case study of a procedural dungeon generator implemented in Unity, using the Binary Space Partitioning (BSP) algorithm in combination with supporting search

methods. By extending BSP with a depth-first search approach for corridor creation, the generator ensures reliable room connectivity and avoids isolated spaces. Additional features, such as decoration placement, torch lighting, and enemy spawning, illustrate how the generator can be expanded to support richer and more engaging dungeon environments. Importantly, the system separates generation logic from gameplay logic, relying on prefabs rather than embedded scripts, which makes it adaptable to different projects and development contexts. From a practical perspective, the generator demonstrates how algorithmic design can be exposed through adjustable parameters, giving developers the ability to tailor dungeon density, room spacing, and structural variety without altering the underlying code. This same flexibility makes the tool suitable for educational use: instructors can demonstrate how small parameter changes produce significant differences in layout, while students can experiment with algorithm-driven design in a visual, interactive environment. There are some caveats to consider in applying the generator. For example, the use of pivot points in walls and doors can occasionally produce alignment issues, and large-scale prefabs may overlap if not properly calibrated. These are not limitations of the concept but practical details that can be addressed by refining prefabs or adjusting loop logic.

Practical Considerations and Refinements

While the current generator already provides a flexible and reproducible workflow, there remain opportunities for further enhancement. Possible refinements include supporting irregular room shapes, improving prefab alignment to reduce overlap, and integrating additional procedural techniques such as cellular automata or noise functions. These extensions would broaden the system's applicability, offering developers and educators even more options when adapting procedural generation to their projects. Overall, the generator offers a clear, accessible foundation for procedural dungeon creation. It contributes as both a prototyping tool for developers and a teaching resource for classrooms, showing how procedural content generation can be meaningfully applied in practice.

References

[1] Baldwin. et al. (2017). *Mixed-initiative procedural generation of dungeons using game design patterns*. 2017 IEEE Conference on Computational Intelligence and Games (CIG), 25-32. https://doi.org/10.1109/CIG.2017.8080411

[2] Baron, J. R. (2017). *Procedural Dungeon Generation Analysis and Adaptation*. ACMSE '17: Proceedings of the 2017 ACM Southeast Conference, 168–171. https://doi.org/10.1145/3077286.3077566

[3] Elkari. et al. (2024). *Exploring Maze Navigation: A comparative study of DFS, BFS, and A* search algorithms.* Statistics Optimization & Information Computing, 12(3), 761–781. https://doi.org/10.19139/soic-2310-5070-1939

[4] Galaxykate. (2016, February 22). *So you want to build a generator.* Tumblr. https://www.tumblr.com/galaxykate0/139774965871/so-you-want-to-build-a-generator

[5] Jain, R. (2023). *A comparative study of breadth first search and depth first search algorithms in solving the water jug problem on Google Colab*. SSRN Electronic Journal. https://doi.org/10.2139/ssrn.4402567

[6] Oyhenard Vazquez, M., & Lincke, O. (2023). *How the level design of the game Lost in Random uses staging, shape language, and scale to signify combat areas.* https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-504904

[7] P. A. Putra. et al. (2023, October 4). *Procedural 2D dungeon generation using binary space partition algorithm and L-Systems*. IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/10285811

[8] Shaker, N. et al. (2016). *Procedural content generation in games*. https://doi.org/10.1007/978-3-319-42716-4

[9] Silva. Et al. (2023). *Dungeon level generation using generative adversarial network: an experimental study for top-down view games*. Seminário Integrado De Software E Hardware, 95–106. https://doi.org/10.5753/semish.2023.229905

[10] X.J.L. Clairbois. (2006, August 31). *Optimal Binary Space Partitions* Eindhoven University of Technology https://research.tue.nl/en/studentTheses/optimal-binary-space-partitions

Acknowledgment

The authors express gratitude to Dr. Tan Bee Sian for providing help and guidance in writing this article. The authors also want to thank Tunku Abdul Rahman University of Management and Technology (TARUMT) for the facilities provided to complete the project.

Funding Information

The author received no funding from any party for the research and publication of this article.

Authors' Bio

Jing Ming Yap is a graduate student of Tunku Abdul Rahman University of Management and Technology (TARUMT), pursuing a Bachelor of Computer Science (Honours) in Interactive Software Technology. His research interests focus on game design, algorithmic generation, and interactive system development. He has worked on Unity-based implementation of generative systems, aiming to make algorithmic design more accessible for game developers and educators.

Nian Chein Woo is a graduate student of Tunku Abdul Rahman University of Management and Technology (TARUMT), currently undertaking a Bachelor of Computer Science (Honours) in Interactive Software Technology. His academic interests include procedural content generation, game development, and software prototyping. He has contributed to the development of Unity-based dungeon generators that demonstrate how parameter-driven approaches can influence gameplay structure and pacing.

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

Ir. Ts. Dr. Chong Kim Soonis 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.