Uhunoma Osemwengie

Supervisor: james Banton

Second supervisor: Bradley DAvis

Procedural Dungeon Generation

GDEV60001 Games development project

Contents

[Abstract 2](#_Toc191420232)

[Introduction 2](#_Toc191420233)

[Aims and Objectives 3](#_Toc191420234)

[Literature Review 4](#_Toc191420235)

[Procedural Content Generation 4](#_Toc191420236)

[Procedural Dungeon Generation 5](#_Toc191420237)

[Binary Space Partitioning 6](#_Toc191420238)

[Random Point Connection 7](#_Toc191420239)

[Drunkards Walk 8](#_Toc191420240)

[Cellular Automata 10](#_Toc191420241)

[L-Systems 10](#_Toc191420242)

[A\* Pathfinding Algorithm 12](#_Toc191420243)

[Dijkstra's algorithm 13](#_Toc191420244)

[Perlin Noise 13](#_Toc191420245)

[Research Methodologies 14](#_Toc191420246)

[Testing and Evaluation 15](#_Toc191420247)

[Results and Findings 16](#_Toc191420248)

[Discussion and Analysis 20](#_Toc191420249)

[Conclusion 22](#_Toc191420250)

[Recommendations 25](#_Toc191420251)

[References 26](#_Toc191420252)

# Abstract

The paper investigates the use of procedural content generation (PCG) in modern game development, with a particular emphasis on its function in improving replayability and player involvement in roguelikes through procedural dungeon generation. The major goal is to investigate how procedural systems, such as binary space partitioning (BSP) and cellular automata, may be successfully used to build dynamic and compelling game dungeons.

The research presented here uses systematic evaluation and testing to assess the benefits and limits of several procedural techniques, showing their potential to address fundamental game design difficulties such as scalability and replay value. The findings are intended to provide actionable insights for developers, pushing for hybrid techniques that mix multiple algorithms to provide specialised and flexible game material. Finally, this research aims to contribute to a better understanding of procedural generation's practical applications by providing a critical viewpoint on its use in creating interesting and replayable gaming experiences.

# Introduction

Procedural level generation has become a cornerstone of modern game development, offering solutions to challenges that arise from the increasing complexity and scale of digital games. PCG is an integral part of game development since it allows many different games to run. For example, without procedural generation, everything, including terrain, levels, audio, and storylines, would have to be developed manually. In most gaming genres, improving replay value is beneficial since it allows each player of a game to have multiple unique experiences; for example, roguelikes. Rogue-like games benefit greatly from content generation because the genre is based on the original game "Rogue" which employed randomly generated levels to enhance its dungeon crawling mechanisms such as grid-based mobility and turn-based gameplay. One of the most appealing aspects of procedural creation is its ability to increase replay value. In classic linear games, after completing a level or solving a puzzle, the experience becomes predictable in subsequent playthroughs. Procedural systems disturb this predictability, ensuring that no two sessions are the same. For example, in roguelikes like Dead Cells or Hades, randomised level layouts, opponent compositions, and power-up distributions push players to constantly adjust their strategy. This variety encourages long-term play, as players return to uncover new difficulties and combinations.

Complex algorithms are used to alter data at the procedural level by applying rules and constraints. The algorithms range from simple random placement to systems that incorporate gameplay and aesthetics. More importantly, this will discuss how dungeon generation is valuable and may be applied in games. At its core, procedural generation relies on algorithms that strike a balance between randomness and intentional design. Simple techniques, such as random tile placement or noise-based height maps, can generate simple structures or terrains. However, to ensure functional and visually consistent outputs, developers frequently add rules and limits to these systems. For example, dungeon-generating algorithms may enforce room connectivity, ensure vital path accessibility, or distribute rewards appropriately to difficulty. PCG incorporates a variety of algorithmic processes, each adapted to certain design objectives and aesthetic consequences. Randomised seeds, for example, act as the foundational inputs that determine an algorithm's output, providing reproducibility while retaining variability, an important aspect for debugging or sharing unique level designs. Cellular automata, inspired by biological cell behaviour, imitate growth and decay principles to create organic, irregular structures like caverns or natural terrain, making them excellent for producing realistically chaotic landscapes. In contrast, binary space partitioning (BSP) takes a more methodical approach, recursively subdividing a space into smaller parts to construct structured layouts such as grid-based dungeons with clearly defined rooms and corridors, balancing randomisation with navigational logic. Wave function collapse is a modern method based on quantum mechanics concepts that iteratively arranges tiles or assets based on proximity, maintaining visual and functional coherence, which is particularly beneficial for creating intricate, interconnected settings such as landscapes or puzzle rooms. These strategies represent the entire range of procedural generation, from randomness-driven simplicity to rule-bound complexity, and each contributes to the construction of dynamic, replayable settings while adhering to gameplay and aesthetic limitations.

Dungeon procedural generation is the creation of dungeon layouts, structures, or levels using algorithms. This is useful since dungeons will no longer need to be set up manually but can instead be generated randomly while remaining unique each time if the technique is properly implemented. Effective procedural dungeon generation goes beyond chance, requiring an organised integration of design principles that account for pacing, difficulty development, and player psychology. This design philosophy is centred on the existence of a crucial path which is a navigable pathway that guides players from start to conclusion, frequently surrounded by optional branches that encourage exploration without compromising essential progression. The risk-reward balance is also important, as high-stakes zones, such as guarded treasure vaults or lore-rich chambers, encourage players to engage in difficulties equal to the possible benefit. Thematic consistency strengthens the experience by ensuring that visual patterns, enemy types, and environmental hazards all work together cohesively, for example, a necromancer's lair may combine skeletal enemies, cursed relics, and caverns to enhance its grim image.

Lindenmayer systems, or L-systems, are a powerful class of formal grammars that have their roots in theoretical biology. Aristid Lindenmayer created them in 1968 with the intention of modelling plant and other biological entities' growth patterns. However, their capacity to produce complex structures from basic rules has resulted in widespread use in computer graphics, notably procedural content generation. L-systems offer a distinct and elegant method of constructing different and sophisticated forms, ranging from organic plant-like structures to geometric patterns and even architectural designs. Their versatility and creativity have made them an invaluable tool for artists, designers, and developers looking to use the potential of generative algorithms.

# Aims and Objectives

The aim of this project is to design a dungeon-generating algorithm that generates rooms before compilation time using a pre-built engine such as Unity or Unreal Engine. This dungeon-generating method should allow the user to tweak specific features of the generation to give users more choices and examine the code against tests to determine how robust and useful the algorithm is at building dungeons for games. To ensure that the rooms are formed correctly, the generator should be able to create corridors that connect numerous rooms, as well as a starting and ending room.

Objectives:

* Design and implement a procedural generation algorithm capable of creating dungeon layouts where all rooms remain interconnected
* Ensure that corridors start from the edges of rooms and maintain proper sizing to avoid unnecessary overlaps or redundant paths.
* Allow flexibility in generation by supporting both random and sibling nodes corridor connecting methods to provide varied layouts
* Implement a customisable system where developers can modify parameters such as room size, corridor width, and dungeon density.
* Maintain a structured project workflow to iteratively refine and improve the algorithm based on testing feedback, ensuring stability and scalability.
* Provide a visual representation to track dungeon generation steps, helping developers identify and resolve issues more efficiently like gizmos

By meeting these goals, this project hopes to provide a useful tool for game creators, allowing them to construct different and entertaining dungeon experiences with ease and efficiency. The emphasis on pre-compilation generation, user customisation, and rigorous testing means that the algorithm is both strong and adaptable, allowing developers to explore a wide range of level design options.

# Literature Review

## Procedural Content Generation

PCG is the process of creating data or content by using an algorithm rather than manually. PCG is widely utilised in a number of media, including video games, art, music, and literature. It is beneficial to all of these mediums since it enables the efficient and diverse generation of content. PCG relies significantly on artificial intelligence and algorithms to generate anything from a small auditory sound to a large open environment. The basic idea behind PCG is to provide a set of rules and parameters, and then let the computer generate the material based on those instructions. This enables developers to efficiently generate massive volumes of diverse material, potentially leading to larger, more dynamic and replayable game experiences. There are various benefits to using PCG in game creation. Perhaps the most obvious is the sheer amount of content it can generate. Instead of carefully developing each level or item by hand, developers can use PCG to generate a large number of versions quickly. This not only saves time and resources but also enables the development of much larger and more extended game environments. Furthermore, PCG can greatly improve replayability. Because the content is produced algorithmically, each replay can provide a distinct experience, with varying levels, difficulties, and rewards. This can keep players involved for far longer, extending the life of a game. PCG is capable of creating dynamic and personalised experiences in addition to content generation. Games can be made more immersive and engaging by adjusting generated content to the player's preferences or playstyle. For example, a game may create stages that are more difficult for experienced players or provide different types of treasure depending on the player's character class.

However, PCG does provide some significant issues. One of the most difficult challenges is maintaining quality and coherence. While algorithms can produce a large amount of information, ensuring that it is continuously entertaining, balanced, and visually appealing can be tough. Randomness can occasionally result in confusing or irritating gameplay experiences. Another problem is managing the output of PCG algorithms. Developers must discover ways to guide the generation process so that it delivers content that is consistent with the game's overall concept and goal. This frequently requires a precise balance between randomness and designer control. Finally, incorporating PCG neatly into the game's design can be challenging. It takes careful preparation and consideration of how randomly generated content will interact with other game elements including gaming mechanics, narrative, and art style. Despite these issues, PCG remains an important tool for game developers, and as technology advances, we should expect to see even more imaginative and engaging use of procedural generation in the future.

## Procedural Dungeon Generation

Dungeon procedural generation is a subset of PCG in game development that focusses on generating the creation of dungeon-like environments. Such environments often comprise interconnected rooms, corridors, traps, puzzles, monsters, and loot, all of which are algorithmically built to provide a new experience with each playthrough. Handcrafted levels are rigid and require significant manual effort to design, whereas procedurally produced dungeons are dynamic and expandable. This implies they can be built in real time, reacting to player actions or game situations while maintaining basic gameplay needs like navigability, balance, and thematic coherence. This method has become a defining feature of genres such as roguelikes, action RPGs, and survival games, where replayability and unpredictability are essential for player engagement. Developers can employ algorithms to create huge, detailed worlds without the need for extensive manual design, allowing for more efficient resource consumption and the capacity to continually offer new material to gamers.

One of the main purposes of procedural dungeon generation is to ensure that no two dungeon layouts are the same. This is accomplished by including unpredictability into the generation process, whether through randomised room placements, different corridor configurations, or unpredictable monster spawns. The goal is to give players a sense of freshness and discovery, prompting them to return to the game several times. However, this randomisation must be carefully managed to avoid creating levels that appear chaotic or incomprehensible. By introducing rules and limits, developers may ensure that each dungeon is unique while yet sticking to a logical structure that facilitates gameplay.

While variety is desirable, it should not come at the expense of playability. A randomly generated dungeon must be walkable, with obvious paths for players to take and no inadvertent dead ends or inaccessible parts. Furthermore, the dungeon's challenges, such as enemy encounters, traps, and puzzles, must be balanced to ensure a fair but entertaining experience. Algorithms must evaluate things like as difficulty progression, resource allocation, and player skill levels. For example, a dungeon may begin with simpler adversaries and fewer traps, gradually rising in complexity as the player goes. This guarantees that the dungeon is hard but not frustrating or overpowering.

To provide an immersive experience, a dungeon's visual and mechanical features must be consistent with its story or aesthetic concept. For example, a dungeon set in a haunted castle would have dimly lighted hallways, crumbling walls, and ghostly opponents, whereas a futuristic sci-fi dungeon might have slick metallic surfaces, laser traps, and robotic enemies. Procedural generation systems must incorporate these thematic components into their algorithms to ensure that the created material is consistent and credible. This sometimes entails leveraging modular assets, such as pre-designed room templates or enemy types, that may be merged in a variety of ways while retaining a consistent visual and narrative tone.

One of the most notable benefits of procedural generation is its potential to reduce the amount of handwork needed in level creation. Handcrafting levels may be time-consuming and resource-intensive, particularly in large or complicated games. Procedural systems automate much of this process, allowing developers to create levels rapidly and efficiently. However, efficiency should not come at the expense of quality. Algorithms must be carefully built to guarantee that produced material satisfies the same criteria as handcrafted levels, with a focus on detail, balance, and player experience. This frequently entails iterative testing and refinement to fine-tune the creation process and address any problems that develop.

## Binary Space Partitioning

Binary Space Partitioning (BSP) is an algorithmic approach that recursively subdivides a space either horizontally or vertically until a specified minimum size is achieved. This process is facilitated using a binary tree data structure (​Putra et al., 2023)​. Understanding BSP requires familiarity with binary trees, which are characterised by nodes having a maximum of two connected nodes, known as children. The topmost node in the tree is referred to as the root node. In cases where a node has two children, the left node represents the lower value, while the right node represents the higher value, simplifying the search process within the tree.

One of the primary reasons why BSP is employed in dungeon generation is that its rooms are less likely to overlap, which is further explored in many different BSP ​tests (Shaker et al., 2016)​. This is because the algorithm only assigns rooms to their own dedicated partitions. Secondly, this technique is highly effective, which is especially helpful in procedural generation because generation must be as fast as possible so that it does not slow down other portions of the gameplay. This algorithm is particularly efficient due to its recursive nature of just dividing until a maximum number of divisions are completed, which means it will take a predictable amount of time and memory resources. The predictability of this algorithm makes it an excellent alternative for developers working with limited resources or wishing to create a dungeon creator in real-time. Furthermore, the fact that BSP utilises a binary tree with a hierarchical structure is quite useful since it may make corridor placement easier. Connecting corridors is also an important stage in dungeon construction. Hence, the fact that our technique improves corridor effectiveness is a big gain.

Despite its many benefits and widespread use, BSP's recursive partitioning algorithm often produces rectangular divisions and rooms. While rectangles are simple to work with and can be useful for level creation, they can reduce the visual variety and spatial complexity of created dungeons. Many games benefit from rooms with more organic and varied shapes, such as circular, triangular, or irregular ones. These shapes can create a more natural and intriguing environment, increasing visual appeal and providing potential for novel gameplay experiences(Niemann & Preuß, 2015). BSP's rectangular limitation can also limit room functionality. Rectangular rooms may not be suitable for certain forms of gameplay, such as combat engagements that necessitate open space or strategic positioning. They may also limit the positioning of objects and features in the room, thus resulting in a less dynamic and engaging environment. While BSP is great for separating area and establishing different chambers, it might be difficult to provide long-range communication within the dungeon. The recursive partitioning procedure can result in isolated portions or clusters of chambers that are insufficiently connected to the rest of the dungeon. This can result in dead ends, confusing paths, or a lack of overall flow across the level design. To solve this issue, developers frequently use extra methods to maintain connection. One strategy is to employ pathfinding algorithms like A\* to connect distant chambers or sections of the dungeon. These algorithms can find the shortest path between two points, ensuring that all regions of the dungeon are accessible and that the layout is consistent. Another option is to change the BSP algorithm to prioritise connectivity. This can be accomplished by adding heuristics or limitations that encourage the algorithm to generate partitions that connect to existing rooms or corridors. For example, the algorithm may be biased towards constructing partitions that align with the dungeon's primary direction, or it could be prevented from creating partitions that isolate a piece of the dungeon.

## Random Point Connection

Random point connection is an algorithmic approach that involves creating corridors by randomly selecting vertices in two rooms and connecting them using a pathfinding algorithm, such as the Manhattan distance algorithm, which is more explored in this report ​(Baron, 2017)​.

The key advantage of Random Point Connection is its ability to generate different and intriguing corridor layouts. By choosing different spots within each room, the algorithm can create corridors that weave around the dungeon space, surprisingly, avoiding tedious straight lines or known patterns. This can create a more fascinating and immersive player experience, as players face diverse setups with each playthrough.

However, this strategy is not without limitations. One worry is the possibility of overlapping or poorly designed corridors. If rooms are built too close together or without adequate attention for corridor location, the resulting corridors may overlap or form awkward junctions. This can disrupt the flow of the dungeon and cause player disorientation. Careful room arrangement and corridor routing algorithms are required to address this issue.

Another problem is making sure the corridors are passable and contribute to a clear feeling of direction across the dungeon. Without sufficient limits, the random nature of the connections might result in complicated paths that confuse the player. Techniques such as requiring a minimum corridor length can help to keep the created corridors organised and purposeful.

## Drunkards Walk

Drunkard's Walk is an alternate technique for simulating corridor generation. This popular technique generates irregular corridors by simulating a random walk through a two-dimensional grid. The procedure starts by putting a walker in a random spot on the grid, usually at a random vertex of a corridor. The walker then moves randomly up, down, left, or right, one cell at a time. This operation will continue until a predetermined stopping condition is satisfied, such as a certain number of steps or a certain percentage of grid coverage. Because of its proclivity for producing irregular routes, Drunkard's Walk method is sometimes used to create cave-like maps or surprising corridors (​Baron, 2017).

At its foundation, the Drunkard's Walk algorithm is surprisingly simple. It is based on the fundamental concept of a random walk, in which a "walker" goes through a grid, choosing a direction with equal chance at each step. This fundamental reasoning results in code that is simple to create and understand, even for those who are new to procedural generation techniques. The algorithm's simplicity provides it an easy entrance point into the field of procedural content creation, allowing developers to quickly comprehend its dynamics and explore with its potential. The Drunkard's Walk's simplicity also helps it be efficient. The algorithm's main operations are merely basic arithmetic and random number generation, making it computationally light. This performance is especially significant in procedural generation, where algorithms frequently need to generate large amounts of content in real time or under tight time limitations. Because of its low computational overhead, the Drunkard's Walk can generate complicated corridor layouts without negatively hurting performance, resulting in a smooth and responsive gameplay experience. Despite its basic simplicity, the Drunkard's Walk is quite versatile. By modifying numerous parameters and limitations, developers can dramatically influence the characteristics of the created corridors, customising them to specific design aims and aesthetic preferences. One important parameter is the halting condition, which defines when the walker's random stroll comes to a conclusion. The requirement can be stated in a variety of ways, including a specific number of steps, a percentage of grid cells visited, or a combination of criteria. By adjusting the stopping condition, developers can regulate the length and density of the created corridors, resulting in anything from tiny, winding tunnels to massive, complex networks.

The inherent randomisation of the Drunkard's Walk ensures that each produced dungeon is distinct. This diversity is important for increasing replayability because players face different corridor layouts and spatial arrangements with each session. The element of surprise keeps the gameplay experience fresh and fascinating, encouraging players to return to the game repeatedly to face new and unexpected challenges. In contrast to handcrafted levels, which provide a fixed and predictable experience, randomly generated dungeons like the Drunkard's Walk add an element of surprise. Players can't rely on memorised paths or methods because each playthrough introduces a new and unfamiliar setting. This promotes adaptation and improvisation, forcing players to think quickly and alter their approach to solve the problems that emerge.

One of the key difficulties associated with the Drunkard's Walk is the possibility of creating disconnected paths or dead ends. The walker's erratic motions might cause passages to abruptly cease or loop back on themselves, leaving isolated portions of the dungeon inaccessible to the player. This can be irritating for players since they may become stuck or unable to access particular portions of the dungeon, impeding their progress and decreasing their sense of exploration. To address this issue, developers frequently use additional algorithms or strategies to maintain communication. One typical method is to utilise backtracking algorithms to analyse the created corridors and find any disconnected sections. These algorithms then seek to connect these parts by either extending existing corridors or building new ones, ensuring that all areas of the dungeon are accessible. Another method is to include connectivity checks in the generation process. These checks ensure that each newly formed corridor connects to an existing corridor or room, avoiding the generation of separate sections. This can be accomplished by maintaining a data structure that tracks the dungeon's connectivity, allowing the program to detect and address any disconnections as they occur.

The Drunkard's Walk's reliance on randomisation makes it difficult to accurately regulate the design and structure of the created dungeon. While limitations and biases might have an impact on the walker's movement, it can be difficult to achieve precise layouts or ensure the existence of certain features. The algorithm's inherent unpredictability may result in dungeons that differ from the developer's original design, harming the desired gaming experience or aesthetic vision. Given the lack of fine-grained control, developers must strike a balance between randomness and design purpose. While embracing the organic nature of the Drunkard's Walk can produce interesting and unexpected results, it is critical to maintain some control over the generation process to ensure that the dungeon meets certain criteria, such as having a clear path from the starting point to the exit or incorporating specific types of rooms or features.

## Cellular Automata

Cellular Automata is a popular room generation technique, notably in dungeon generation. It involves creating a two-dimensional grid with randomly assigned cells as either walls or floors. Following then, a set of rules is followed to gradually transform the grid into a well-organised dungeon. These rules often include assessing neighbouring cells' statuses to determine whether each cell should be classified as a wall or a floor in the future iteration. For example, if a cell is surrounded by many neighbouring wall cells, it may transition into a wall, whereas cells with fewer neighbouring walls may continue as floors or even transform into floors. This iterative procedure, like natural processes, is repeated several times to create smoother and more coherent room and corridor layouts.

Cellular Automata's popularity stems from its capacity to create environments that feel organic and lifelike, improving the player experience. This is incredibly useful for creating irregular chambers, tunnels, and caves that appear more natural than organised grid-like dungeons (​Niemann & Preuß, 2015)​. Furthermore, Cellular Automata allows for greater versatility because the algorithm's designer can alter some parameters to make the generation accomplish elements that the user sees, such as what the initial walls should be. The flexibility allows for more experimentation, which helps the designer by providing more control and modification of the dungeon. Another advantage is that the algorithm will offer distinct results each time, increasing the game's replay value.

Despite its usefulness, Cellular Automata has certain drawbacks. For example, ensuring that each room is interconnected can be difficult due to the algorithm's random and organic nature, resulting in isolated or inaccessible areas. To solve this, it is customary practice to include additional algorithms that increase effectiveness while reducing the risk of connectivity problems. However, this approach's unstructured and organic growth-like behaviour may cause confusion for players throughout exploration. Despite these challenges, Cellular Automata remains an important technique to procedural dungeon generation, providing distinctive and natural-looking environments that can be quite useful in a variety of video games.

## L-Systems

Lindenmayer systems, or L-systems, are a powerful class of formal grammars that have their roots in theoretical biology. Aristid Lindenmayer created them in 1968 with the intention of modelling plant and other biological entities' growth patterns. However, their capacity to produce complex structures from basic rules has resulted in widespread use in computer graphics, notably procedural content generation. L-systems offer a distinct and elegant method of constructing different and sophisticated forms, ranging from organic plant-like structures to geometric patterns and even architectural designs. Their versatility and creativity have made them an invaluable tool for artists, designers, and developers looking to use the potential of generative algorithms. At their essence, L-systems are string rewriting systems. They work with a collection of symbols known as the alphabet and a set of production rules that define how these symbols change over time. The method begins with an initial string known as the axiom, which serves as the basis for the generating process. The production rules are then successively applied to the string, replacing each symbol with a sequence of symbols determined by the rules. This repeating process of symbol replacement produces increasingly complicated strings, which can subsequently be read as instructions for generating geometric structures or other types of data.

L-systems are fundamentally string-rewriting systems that use iterative symbol replacement to produce increasingly complex structures over time. To explain this process, consider a simple L-system with an alphabet of two symbols, 'A' and 'B'. We start with the starting string, or axiom, 'A', and construct two production rules: 'A' transforms into 'AB', and 'B' transforms into 'A'. Beginning with the axiom, each iteration applies these rules to all symbols in the string. In the first iteration, the rule A to AB substitutes 'A' with 'AB', creating the string 'AB'. In the second iteration, the rules are applied again, transforming the 'A' in 'AB' into 'AB' and the 'B' into 'A', resulting in 'ABA'. This symbol replacement process continues, resulting in ever longer sequences such as 'ABAAB', 'ABAABABA', and so on. The beauty of L-systems is that these strings can be read as drawing instructions. For example, interpreting 'A' as 'draw a line forward' and 'B' as 'turn right by 90 degrees' yields the intriguing geometric pattern known as the Cantor set. This small example exemplifies the core notion of L-systems: creating complexity from a few basic rules and an initial starting point (Guo et al., 2020).

L-systems can be broadly categorised into two types: deterministic and stochastic. In deterministic L-systems, each symbol has a single, preset production rule, hence the generation process is completely predictable. As seen in the preceding example, the output will always be similar when the same axiom and rules are applied. In contrast, stochastic L-systems include a chance component. Each symbol may have numerous production rules, each with a likelihood of being applied. During generation, the algorithm chooses a rule at random based on these probabilities, resulting in a wide range of results despite the same starting conditions. This unpredictability is especially useful for designing organic structures such as plants and trees, where natural variation and irregularity are required for visual reality.

Another key feature of L-systems is context sensitivity. In context-sensitive L-systems, creation rules can consider the surrounding symbols while deciding how to replace a symbol. This enables for more complicated and nuanced patterns because the generation process is influenced by each symbol's local context. For example, a rule may state that a symbol should be substituted only if it is before or followed by a specified symbol or sequence of symbols.

L-systems have a wide range of applications in procedural generation, particularly when it comes to constructing realistic structures and patterns. They specialise in creating realistic plants and trees, capturing the complicated branching patterns and organic growth processes that occur in nature. L-systems may produce a wide range of plant morphologies by altering parameters such as branch length, angle, and thickness. Furthermore, L-systems can be utilised to build dungeon layouts, resulting in interconnected rooms and corridors with organic shapes that are enhanced by combining randomness and context-sensitivity for unexpected designs. L-systems help to generate realistic terrain characteristics such as mountains and valleys, and when paired with techniques such as Perlin noise, they produce varied and visually appealing landscapes. Their capacity to form fractal patterns, leads to complicated and captivating designs. Beyond the natural world, L-systems are used in architectural design to experiment with diverse space configurations to create imaginative and practical building layouts.

L-systems provide several compelling advantages in the area of procedural generation. For starters, their core ideas are extremely simple to understand and apply, making them accessible to a wide spectrum of users, from newbie developers to experienced programmers. Despite their simplicity, L-systems are exceptionally expressive, capable of producing a wide range of sophisticated and complicated structures, from organic shapes such as plants and trees to exact geometric patterns. This adaptability arises from their capacity to define a set of rules and an initial starting point, where complexity grows through iterative development. Furthermore, L-systems provide great control over the generation process. Parametric L-systems, in example, enable smooth modifications by introducing characteristics such as length, angle, and width that may be changed by rules made by the programmer. This allows for the construction of customised forms and features, adjusting the output to match specific design needs. Furthermore, context-sensitive L-systems improve control by considering the surrounding symbols when applying rules, resulting in more nuanced and context-aware creation. Another advantage of L-systems is their capacity for growth. They can be used to create constructions of varying sizes, from minute details to large patterns. This makes them useful for a wide range of applications, from creating fine textures to planning huge landscapes or complex architectural plans.

However, L-systems have limitations. Creating highly complicated structures can be computationally expensive, especially when working with a large number of iterations or a long alphabet of symbols. L-systems are iterative, which implies that each step builds on the preceding one, and as complexity increases, so does computational weight. This is a concern for real-time applications or for producing large amounts of content. Another problem is maintaining precise control over the created shapes. While parametric and context-sensitive L-systems provide tools for fine-tuning, mastering this control may necessitate a thorough grasp of the interactions between rules, parameters, and the generation process. To achieve specific outputs for very complex structures, the L-system's components may need to be carefully experimented with and refined iteratively. Furthermore, L-systems are primarily intended for creating static structures. While they are skilled at constructing sophisticated and visually appealing shapes, including dynamic components or real-time interactivity might be more difficult. L-systems often produce a fixed output based on the initial conditions and rules, and modifying them to respond to human input or environmental changes requires additional techniques and considerations.

## A\* Pathfinding Algorithm

A\* is a powerful and efficient pathfinding method that identifies the shortest route between two points in a graph. In the context of a randomly generated dungeon, the graph represents the layout of the dungeon, with nodes symbolising rooms and edges denoting connections between these locations. A\* employs a heuristic function to evaluate the distance between a specific node and the intended destination, enabling it to prioritise the exploration of likely routes. By combining this heuristic with the actual cost of traversing the graph, A\* effectively identifies the optimal path, minimising unnecessary exploration of dead ends or inefficient routes.

A\* has various applications in procedurally constructed dungeons. Perhaps the most common application is controlling NPC movement. In several games, enemies and other NPCs must travel the dungeon to locate the player, patrol assigned regions, or complete specific tasks. A\* allows these figures to make informed decisions regarding their journey, avoiding obstacles and finding the shortest path to their destination. Without a strong pathfinding system like A\*, NPCs may become imprisoned, roam aimlessly, or react in unnatural ways, reducing the player's immersion.

The importance of A\* stems from its efficiency and ability to find optimal or near-optimal paths. In complicated dungeons with several rooms and branching passages, the search area for alternative paths can be enormous. A\* intelligently navigates this area, considerably lowering the computational cost of pathfinding when compared to less advanced algorithms. This efficiency is critical for keeping the game running smoothly, especially when numerous NPCs are navigating the dungeon at once (Cui & Shi, 2010).

## Dijkstra's algorithm

Dijkstra's algorithm, developed by Edsger W. Dijkstra, solves the shortest path from one source problem for graphs with non-negative edge weights. Given a beginning node, the algorithm calculates the shortest path between that node and all other nodes in the graph. It explores the graph iteratively, keeping track of which nodes have been visited and which have not. Initially, all nodes except the source are marked as unvisited, and the distance between the source and itself is set to zero, while the distances between all other nodes are set to infinity.

The algorithm selects the unvisited node with the shortest distance from the source. This node is then flagged as visited, and its neighbours are investigated. The algorithm determines the distance from the source to each neighbour by adding the distance to the current node and the weight of the edge that connects them. If the computed distance is less than the current reported distance from the neighbour, the recorded distance is updated. This process is repeated until all nodes have been visited, at which time the algorithm determines the shortest path from the source to each other node in the graph.

Dijkstra's algorithm is useful in a variety of procedural generating contexts. While it is not always optimal for real-time NPC navigation because to the computing expense in large networks, it is a useful tool for pre-processing and analysis. For example, in a procedurally created dungeon, Dijkstra's algorithm can be used to calculate the distances between all chamber pairs. This precomputed distance matrix can subsequently be used for a variety of purposes, including AI decision-making, level design analysis, and calculating the dungeon's overall complexity. Knowing the shortest path between critical areas allows the game to dynamically change enemy placement or resource distribution to provide a balanced challenge to the player. Furthermore, comprehending pathfinding principles requires knowledge of Dijkstra's algorithm. It offers a simple and easy way to explore a graph and find the shortest pathways. Its logic serves as the foundation for more complex algorithms such as A\*, which can be considered an optimisation of Dijkstra's algorithm. A\* uses a heuristic function to steer the search, making it more efficient in determining the shortest path to a specific target node. However, the essential ideas of graph traversal and distance calculation in A\* are based on Dijkstra's algorithm.

## Perlin Noise

Perlin noise is a type of gradient noise, which is a mathematical function that generates a continuously fluctuating field of numbers. It was created by Ken Perlin in 1983 and has since become a staple of procedural generation in computer graphics, particularly for creating natural-looking textures, terrains, and patterns. Its versatility arises from its ability to generate a continuous, pseudo-random series of numbers that are both unpredictable and coherent, achieving a balance between unpredictability and organic flow.

Perlin noise is fundamentally based on a grid of points. Each grid point receives a pseudo-random gradient vector, which essentially sets the direction of the steepest slope at that location. To calculate the noise value at a certain grid point, the method first determines the grid points that surround it. It then computes the dot product of the gradient vector at each grid point and the vector connecting that grid point to the place of interest. These dot products are then interpolated with a smooth interpolation function, yielding a single noise value for the specified position. The use of many octaves is what gives Perlin noise its fantastic quality. An octave corresponds to a distinct frequency and loudness of noise. Perlin noise can generate intricate and detailed patterns by merging numerous octaves, each with a unique scale and contribution. Lower octaves define the overall shape and structure, whereas higher octaves add finer details. This multi-octave method generates textures that imitate natural phenomena such as clouds, mountains, or wood grain, combining large-scale patterns with fine details.

The uses of Perlin noise in procedural generation are numerous and diverse. Perlin noise can be utilised in dungeon generation to add organic variation and break up the rigidity of purely geometric layouts. For example, it can be used to alter the shape of rooms, resulting in uneven walls or curved corridors. It can also affect the positioning of objects in the dungeon, resulting in a more natural and less uniform distribution. Furthermore, Perlin noise can be utilised to build height maps to create different terrain within the dungeon, increasing the environment's complexity and realism.

Beyond dungeon production, Perlin noise is often used in other elements of procedural content creation. It is widely used to generate realistic terrains, which can include undulating hills, craggy mountains, and flowing rivers. Perlin noise, when used to generate textures, can create elaborate patterns that resemble stone, wood, or clouds. It is also utilised in animation to provide fluid and organic movements, as well as visual effects to represent fire, smoke, and other natural phenomena.

# Research Methodologies

During the research phase, methodologies, tools, and frameworks for procedural generation in video games will be explored. Procedural content generation (PCG) is extensively employed in games like "Rogue" and "Minecraft," providing substantial advantages such as replay ability and resource efficiency. To obtain a better understanding of the subject, three major subjects must be researched: Understanding Procedural development, Unity Engine Capabilities, and looking at existing work related to dungeon development. For procedural generation, there were other places to go, such as public GitHub repositories, which provide more information on how different people utilise procedural generation approaches.

The goal of this project is to investigate how binary space partitioning can be utilised to arrange rooms in a 2D grid in Unity. Unity was chosen over Unreal Engine since it offers numerous advantages that can be applied in this circumstance. For example, this project will use various algorithms, with a primary focus on performance, which means that an engine that is lightweight and easier to browse would reduce production time, as time is a constraint. Unity also employs C# instead of Unreal Engine's C++, which might be considered as an advantage because C# is simpler and has general garbage collection and memory management tools, making it less prone to memory-related issues that could impair development. Unity's excellent documentation and active community make it easy to implement sophisticated algorithms such as binary space partitioning. The engine's 2D toolset is also ideal for grid-based room placement and spatial division tasks.

The procedural dungeon algorithm will begin by dividing the space with Binary Space Partitioning and randomly placing a room within the partitioned space. The size of the room will be random to ensure that each room is unique. To improve the quality of the created dungeon, this method must ensure that no rooms or corridors overlap. Also, the dungeon can be large or small, making it easier to manage and regulate. This artefact will need developing a splitting space code, which will then allow rooms to be inserted inside each space, implying that each node in the BSP tree will have at least one room, making it easier to manage and debug. To ensure that the rooms and space splitting are done correctly, gizmos are utilised for debugging purposes via the unity engine to check if each space partition has a room that occupies it, if the room prefab covers the gizmo dimensions, and if there are any mistakes in room placement. Rooms will then be linked together using a random connection mechanism or by connecting them via each sibling node. To make the random connection algorithm function, each room is looped through and tested against every other room as well as a random value ranging from 0 to 1 before connecting if a specified number is reached. For example, if the random value is less than 0.1, which indicates there is a 10% chance this happens, the room can link to the current other room being compared; otherwise, it will continue to the next room until the loops are completed. The method will be created via a script with several variables that can be updated on the inspector, such as how big the dungeon is with dungeon width and height, as well as a variable for the maximum depth of the binary tree. This is significant since it allows for greater control of how many rooms the dungeon should have, and because it works by dividing the grid in half, the number of rooms will usually be in multiples of 2.

## Testing and Evaluation

After the script has been built, it is subjected to numerous tests to confirm that it operates as intended, as well as to assess its quality and performance under various scenarios. The first test will be to determine whether all the rooms are placed in the correct partition of the BSP split, with the intended answer being that the rooms are correctly placed inside each partition, resulting in no room overlaps. Another test that will be performed is a room limits test, which will attempt to ensure that the rooms do not exceed the dungeon's limit, allowing the dungeon to be traversable. Additionally, when corridors are generated, it will be made sure that each corridor does not pass through a room, preventing it from being obstructed. The fourth test will look at the room prefabs to ensure they match the expected proportions and do not cause rendering or collision issues. This test is especially critical for ensuring visual consistency and avoiding gameplay disruptions caused by incorrectly sized or situated rooms. The fifth test will involve creating smaller size dungeons to assess the algorithm's effectiveness with minimal dimensions. The generator was intended to provide a suitable dungeon plan with no errors, despite the restricted area available. This test ensures that the algorithm is flexible and adaptable to varying dungeon sizes. The sixth test will measure frames per second (FPS) during runtime to determine how the algorithm affects performance. The desired effect was for the FPS to remain consistent, with no noticeable decreases during or after dungeon generation. This test is critical for confirming that the algorithm does not degrade the game's performance. The seventh test assessed the algorithm's behaviour at the smallest conceivable dimensions, such as creating a single room or an overly simplistic layout. The desired result was that the generator will produce a genuine dungeon with no faults, even under harsh limits. This test validates that the algorithm is resilient and reliable.

# Results and Findings



Fig 1: Showing the rooms (green square) in a partition and not on top of the partition line (partition line)

The initial test was to ensure that the rooms were properly partitioned during the space-splitting stage. This was tested by inserting gizmos every time the dividing process was completed, which has been developed and tested. After running the generator 10 times, the rooms appear to be accurately arranged each time, as shown in the photographs above. This lays a solid foundation for the remainder of the code to function properly because it ensures that the room placements do not contain flaws that could cause other parts of the tests to fail. The above images show 3 out of the 10 tests where rooms are placed correctly.

The second test looked at what occurs if the room sizes are significantly larger or smaller than the dungeon size. This was checked by adjusting the minimum and maximum room sizes with the inspector. To begin, the minimum will be altered to be greater than the maximum room size to see if there is any validation to ensure that the user does not enter a minimum room size greater than the maximum room. As a result, if this occurred at higher amounts, the minimum size of the room was changed to be one smaller than the maximum.



Fig 2: Showing the rooms with a room prefab (Sandahl, 2012) being connected by blue corridors from the test below.

The third test was a corridor check, which assessed whether the corridors would go through numerous rooms after generation. After multiple attempts, it appears that the larger the dungeon, the more probable it is that at least one corridor will pass through a room to reach their destination. This is due to the lack of an actual algorithm in the code that ensures that when identifying the path towards the goal end position of the corridor, which is normally at the border of a room. There are numerous approaches to addressing this issue, but the most common is to utilise a pathfinding algorithm from one edge of a room to another to reduce the likelihood of a room collision occurring. One of the most used pathfinding algorithms is the A\* method, which calculates the shortest path between two points in a graph or grid. A\*'s efficiency stems from its use of heuristics to estimate the cost of reaching the other location. The Manhattan distance is the most popular heuristic to utilise because it is well-suited to grids with no diagonals but only horizontal and vertical pathfinding. However, there is a significant disadvantage to utilising this method, which is that two lists are required to implement it, hence the time to create the dungeon would increase exponentially as the dungeon size increases.



Fig 3: Rooms being generated with different prefabs left using paint and right being from online (Sandahl, 2012)



Fig 4: The left showing the rectangle sprite while the right is showing it implemented

The fourth test is an extremely useful procedure for determining how prefabs work in Unity. Using several different images produced in Paint and then imported into Unity to see if the code will allow different images and sprites to be used as room prefabs. A web image was used first, followed by images created with paint, to see if the code would still be able to render the images effectively while also ensuring that the images were the correct size and location on the grid. The first image utilised came from this website (Sandahl, 2012), and it depicted a room from a top-down dungeon, which was ideal for this test. The next image was created using paint and features a white backdrop with a black square in the centre. When both were added to Unity and converted into sprite prefabs, they could be added to the script via the inspector and display successfully, indicating that the prefab integration is working as expected. Even utilising a rectangle sprite rather than a square it has achieved positive results because it would attempt to realign the sprite into a square, making it suitable for the generator.



Fig 5: Smallest possible dungeon being able to be made with the smallest room in-game view

The fifth test consisted of generating small dungeons to see if making a dungeon as small as possible would still allow it to generate rooms to a sufficient standard or, at the very least, prevent the user from entering values that would result in significant errors. When you enter numbers less than 50 for dungeon height or width, the script will set them to 50, ensuring that no problems develop because of incorrect inputs. This is a very nice result because it prevents any erroneous input from going through and interfering with the creation while yet allowing anything to be generated on the smallest feasible scale.



Fig 6: The average FPS calculated from 2 tests after 1 minute of the code being run

The sixth test includes a dungeon performance test that tracks how much load the dungeon will put on the engine as the FPS were tested after the dungeon was generated for the first minute by testing it on the smallest possible dungeon size and then on the largest possible dungeon size. The frame rate was measured after multiple runs, both with and without corridors. The tests were aimed to stress the engine and discover any performance bottlenecks associated with dungeon development. Specifically, the FPS was measured after the dungeon was built, with the performance tracked for a full minute. This consistency across dungeon complexities, with and without corridors, and with or without A\* pathfinding activity indicates that the engine can efficiently handle dungeon generating load while maintaining a fluid gameplay experience. This is an important finding since it demonstrates the engine's durability and capacity to expand across different levels of dungeon complexity without significantly reducing performance.







Fig 7: Execution time of generating the dungeons from 3 different tests at the smallest possible dungeons while depth is 4.

The next test was to see how long it took for different dungeons to be generated based on the settings added. This test was important in learning how different components of the dungeon generator affect generation time, such as room location, corridor placement, and even A\* pathfinding. One of the early tests was merely having room placement and seeing how long it would take after ten attempts to generate the rooms in a dungeon with four depth. This test produced a positive result because all generations had a consistent time of roughly 0.02 milliseconds, which is a very decent amount of time, indicating that room placements do not take that long to split the space four times and place the rooms in it. Even increasing the dungeon width and height to their limits did not significantly affect the formation time, which remained less than 0.09 seconds. However, raising the depth to a higher number, such as a 6 or 7, will take a much longer time to complete this generation, thus it would probably be best to limit this depth choice to be between 1 and 4/5 to allow the code to work correctly. To stress test this further, the room size is set to the smallest achievable value, which is between 10 and 11, while creating as many rooms as possible, putting a higher strain on the engine. This resulted in a significantly longer time to build these dungeons because having the corridor generation on top of the room placement created a tremendous load that required a lot of CPU time.

The more tests performed, the more difficulties and limits must be overcome, such as assessing how much depth the engine can withstand before crashing. For example, if the computer utilised does not have enough RAM to run the software at a specific setting, the Unity Engine will freeze and not function until it is forcefully terminated. To combat this, the binary partitioning depth was limited to 4, which allowed for many rooms to be made. A depth of 4 means 2^4 number of partitions allowed according to the space available. If the code runs out of space to partition, it will stop, cutting space in half. 16 is very ideal for any dungeon size. At the lowest feasible dungeon size and depth of 10, the dungeon can still generate in under 0.1 milliseconds. This suggests that the depth of 4 will no longer be a bottleneck, implying that increasing or decreasing it further would not produce different outcomes in efficiency. However, when the dungeon was set to the maximum level and the A\* algorithm was not used to prevent corridors from colliding with rooms more securely, there was a significant increase in generation efficiency and allowing for the engine to work without freezing, which increased from 0.1 to almost 3 milliseconds. The use of the A\* pathfinding algorithm in this test is extremely detrimental because applying it and then switching to the maximum size dungeon would cause the engine to crash, implying that there is no way to integrate A\* with maximum capacity, necessitating the use of a dungeon limit or a basic connection method if the dungeon exceeds a specific threshold.

# Discussion and Analysis

After multiple tests, one of the issues with the implementation is that the dungeon grid is recorded as an array that includes the dungeon width and height parameters. Even though this is an effective method of storing the dungeon because it allows walls and corridors to be represented as numbers (for example, in this implementation, 0 was used as empty space, 1 as rooms, and 2 as corridors), the problem with C#.NET arrays is that it limits the grid to a numerical value. This figure is 2^31, which is approximately 2 million. This is significant because it can only hold 20000x20000. This can be problematic since it prevents users from manually adjusting the dungeon height and width to any amount they like. This limitation exists because C# utilises 32-bit unsigned integers to represent arrays, which have a range of 2^-31 to 2^31-1 (2,147,483,648 to 2,147,483,647). This limitation can be handled in several ways. One approach is to modify how the dungeon grid is stored from an array to another form of data structure, such as utilising a dictionary instead. This enables for extremely big dungeons to be constructed as it allows you to go beyond the 2^31 by first, allowing each tile to be stored in a 64-bit integer which would allow for a larger amount to be kept while only storing the tiles that are needed, enhancing efficiency.

The tests and assessments into the artefact provided valuable knowledge on the topic of dungeon generation. These tests demonstrated that creating a dungeon generator will require a significant amount of work and trial and error to understand and execute. Understanding how corridor generation works, for example, differs significantly from understanding how space partitioning works since corridor creation is more critical and more difficult to implement due to the difficulty of debugging the corridors. This is because, while the corridors may generate correctly at times, using random functions may result in very different outcomes, such as corridors on top of each other or corridors that do not fully connect. Although this occurred on certain occasions, most of the time the corridors were effectively generated with the rooms, allowing for a successful BSP deployment. There is also the fact that when creating corridors, you must account for the fact that you must be able to start and end the corridor right before the room sprite, which is more difficult than it appears because the room may take up more space than the dungeon grid suggests, causing the corridor to not connect fully, making it even more difficult to implement.

One of the major challenges with the implementation was attempting to design an A\* algorithm that could efficiently connect the corridor to another location in the dungeon while avoiding obstacles and determining the shortest route to the target. This is a problem because it would take a very long time for each corridor to be implemented. While this may work fine for smaller dungeons, it appears that Unity cannot run this because there is so much work that will need to be done at once, causing the program to freeze and terminate. There are numerous solutions to this problem, including employing a different pathfinding algorithm, such as implementing a breadth-first search, which could reduce the amount of recursion the engine would have to cope with. Another solution for preventing this is to employ multithreading more efficiently. Multithreading is a programming method that allows users to have applications run concurrently. On the Unity engine, all processes are performed on a single thread, known as the main thread, which means that the entire code will be executed one at a time. This is problematic since this code makes extensive use of recursion, which means that each function must be executed sequentially. So, since there are many separate functions that require recursion at the same time, using multithreading would allow each application of recursion to be under various threads, allowing them to be executed concurrently. Even if this is effective, there are some drawbacks, including the fact that it will result in a significantly higher CPU load and that even if the jobs are completed concurrently, the freeze may not be resolved or the time it takes to construct the dungeon to be reduced.

This is because some of the recursion will rely on the other to complete their task first before it can execute, implying that it will not boost performance but rather maintain it. Dijkstra's approach could alternatively be used instead of A\* because it is easier to build and provides more control while still finding the shortest path. However, this may not solve the problem because it is much slower than A\*, which may result in higher demand and a longer time to build corridors.

The use of BSP appears to have been excellent. This is due to its implementation, which has allowed for extremely exact room arrangements. Overall, BSP is an excellent technique for dungeon generation because it is a relatively simple algorithm to implement into most engines; all that is required is knowledge of how to split space and create rooms within those partitions. There is also the fact that BSP is extremely rapid while producing high-quality rooms. As demonstrated by the testing, every time a newly created dungeon is made, the rooms become spread out while also being highly distinct from each other because it will randomly pick how to partition space each time, allowing for increased variation. There is also the fact that the rooms will be structured in a binary tree, which will make it easier to manage many aspects, such as knowing which room was created first and which was created last. This is beneficial because it allows the user to be more customisable; for example, the first room generated may be the starting room, and the last room could be the end room, which is relatively simple to implement because the binary tree is ordered.

However, this cannot be stated in the corridor connection algorithm. This is due to the A\* algorithm, which could not always operate correctly or would require significant tuning to work as intended on the Unity engine, resulting in several crashes, as opposed to BSP, which could do the operation without trouble. Also, because A\* pathfinding does not ensure that all rooms are connected, another algorithm is required to ensure that this works as intended and that no dead ends are created, such as using a minimum spanning tree, which would reduce the likelihood of rooms being isolated or disconnected. There is also the fact that A\* consumes a lot of memory because its implementation requires two lists and recursion, which takes a long time to complete when compared to the BSP space-splitting technique.



Fig 8: Inspector for the dungeon generation script

The artefact achieved many goals, such as the script inspector, which allows anyone with the script to change many things to customise the code to their liking, such as having the dungeon width and height to give them more control over how big the dungeon should be. There is also the ability to modify the minimum and maximum size of the rooms, which is a function that will randomly create a room between those limits, allowing for greater variability between each room and making the dungeons appear more unique with each generation. There is also the option to raise the corridor width to a quantity between 0.5 and 5, as well as change the colour of the line renderer to anything appropriate, which can improve the dungeon's clarity and visual appeal.

The use of Unity in the artefact was acceptable because, while it allowed for prefab usage and easier implementation of various algorithms, it still had many limitations that other engines did not have, such as arrays having a low limit to how much they can handle, making the dungeon grid system more difficult and limited.

# Conclusion

Overall, procedural dungeon generation is an intriguing and challenging field that combines creativity, mathematics, and computer science to generate dynamic and compelling landscapes for games and simulations. This dissertation investigated the design, implementation, and evaluation of a procedural dungeon generation algorithm, with a focus on constructing interconnected dungeon layouts while solving critical difficulties such as room connection, corridor placement, and generation flexibility. The creation of an artefact to test and visualise the algorithm revealed important information about the approach's effectiveness and possible uses in game development. This conclusion summarises the important findings, considers the aims met, and explores the work's broader implications.

The key goals of this dissertation were to design and implement a procedural generation algorithm capable of building interconnected dungeon layouts, ensuring optimal corridor placement, providing flexibility in generation methods, and providing a customisable framework for developers. Many of these goals were addressed successfully through a combination of algorithmic design, iterative testing, and the production of a visual artefact to help with debugging and refinement.

The building of an artefact to test and visualise the algorithm was an important part of this project because it gave a concrete means to examine and improve the procedural generating process. Implemented in Unity, the artefact provided a real-time visualisation of the dungeon generation process, allowing developers to observe the algorithm in operation and identify possible difficulties as they developed. One of its important features was step-by-step visualisation, allowing users to trace the generation process from room placement to corridor building. This provided a clear knowledge of how the algorithm functioned. The artefact also featured debugging tools including visual indicators for overlapping rooms, connecting hallways, and other typical faults, facilitating problem resolution. The inspector interface allows for easy modification of generation parameters, including room size, corridor width, and dungeon density, with real-time results. This combination of visualisation, debugging, and customisation tools not only streamlined the development process but also showcased the algorithm's potential in a practical and understandable manner.

The artefact served not just as a testing tool, but also as an example of the algorithm's abilities. The artefact demonstrated the algorithm's versatility and adaptability, emphasising procedural generation's capacity to generate dynamic and compelling dungeons. This project provided a lot of lessons about procedural generation and algorithmic design, which can be used in the future. Iterative testing is vital for discovering and resolving difficulties during the algorithm's development process. This iterative method meant that the final solution was stable, scalable, and achieved the project's goals. Supporting numerous generation methods and customisable parameters increased the algorithm's versatility and adaptability to various use cases, highlighting the need for design flexibility. This flexibility is critical for developing tools that can be used in a number of settings, allowing developers to modify the system to their own requirements. Visual feedback helped debugging and refining by allowing for simpler identification of faults and understanding of code changes' impact on output. Breaking the project into manageable segments and focussing on each aim individually helped maintain a controlled process, ensuring a high-quality outcome. These concepts emphasise the significance of careful preparation, adaptability, and ongoing improvement in algorithmic design and procedural production.

This project demonstrates how procedural dungeon generation may generate dynamic, interconnected, and visually consistent areas. By integrating algorithmic design, iterative testing, and visual feedback tools, the project created a strong and flexible system that fits programmer demands while also expanding the creative potential of procedural generation. The findings from this study serve as a firm platform for potential projects and emphasise the need for thorough planning, testing, and refining in algorithmic design. As the area of procedural generation evolves, tools like those developed in this project will become increasingly crucial in understanding the subject even further.

# Recommendations

While the project met most of its primary goals, there are various ways in which it might be enhanced or expanded to improve its functionality, usability, and flexibility. A huge improvement would be to convert the current Unity script into a specialised tool using the Unity Editor Window. This eliminates the requirement to tie the script to a GameObject, resulting in a more intuitive and accessible interface for users. By designing a custom editor tool, developers may generate dungeons right within the Unity Editor, eliminating the requirement to launch the game engine, and making the tool easier to integrate into their projects. This technique would help streamline the workflow by allowing users to tweak parameters and see changes in real-time without having to click through the inspector or launch the game in the editor.

Another area of Improvement could be to increase corridor pathfinding and connecting success. Currently, the algorithm enables random and sibling node connections, as well as employing A\* to improve the likelihood of corridors linking to a room, but further techniques might be added to provide even greater flexibility and control over dungeon construction. Implementing Dijkstra's method can optimise corridor location, reducing overlaps and dead ends. Furthermore, introducing weighted connections based on room size or distance may result in more dynamic and varied layouts, increasing the algorithm's adaptability.

Another option is to make the tool compatible with other game engines, such as Unreal Engine or Godot. By abstracting the algorithm's essential logic into a standalone library or framework, the generator might be customised to function across several platforms, making it more accessible and useful to a wider range of developers. This would entail developing engine-agnostic code and providing documentation for integration with different platforms.

The project might benefit from more extensive customisation options, including support for rooms that are not square, multi-level dungeons, and procedurally placed assets like doors, traps, and treasure. These characteristics would increase the depth and variety of the created dungeons, making them better suited for complicated game concepts. For example, using circular or polygonal rooms could result in more organic and visually appealing layouts, whereas multi-level dungeons with stairs or lifts would offer a new layer to the development process.

# References

​Putra, P. A., Tarigan, J. T., & Zamzami, E. M. (2023). Procedural 2D Dungeon Generation Using Binary Space Partition Algorithm and L-Systems. Proceedings - 2023 10th International Conference on Computer, Control, Informatics and Its Applications: Exploring the Power of Data: Leveraging Information to Drive Digital Innovation, IC3INA 2023, 365–369. <https://doi.org/10.1109/IC3INA60834.2023.10285811>

​Shaker, N., Liapis, A., Togelius, J., Lopes, R., & Bidarra, R. (2016). Constructive generation methods for dungeons and levels (pp. 31–55). <https://doi.org/10.1007/978-3-319-42716-4_3>

​Niemann, M., & Preuß, M. (2015). Constructive Generation Methods for Dungeons Seminar-Thesis in Procedural Content Generation for Games Declaration of Authorship. <https://www.wi.uni-muenster.de/sites/wi/files/users/mpreu_02/games-material/ba-vm-ss2015/marco_niemann_-_constructive_generation_methods_for_dungeons_-_thesis.pdf>

​​Baron, J. (2017). Procedural dungeon generation analysis and adaptation. Proceedings of the SouthEast Conference, ACMSE 2017, 168–171. <https://doi.org/10.1145/3077286.3077566>

Cui, X. and Shi, H. (2010) *(PDF) a\*-based pathfinding in modern computer games*, *A\*-based Pathfinding in Modern Computer Games*. Available at: <https://www.researchgate.net/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games> (Accessed: 20 January 2025).

Guo, J. *et al.* (2020) *Inverse procedural modeling of branching structures by inferring L-Systems*, *ACM Transactions on Graphics*. Available at: [https://dl.acm.org/doi/pdf/10.1145/3394105](https://dl.acm.org/doi/pdf/10.1145/3394105%20) (Accessed: 15 January 2025).