Brazelton Devon

“How can a playable 2D dungeon-layout be effectively generated for games by implementing procedural generation techniques using a modern game engine?”

Supervisor: Abeele Vanden Alex

Coach: Verspecht Marijn

Graduation Work 2023-2024

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT & KEY WORDS

Procedural generation serves as a vital asset in Game Development, expediting the development process and transforming the creation of environments and level elements. It introduces variation and uniqueness that would typically demand exponentially more time within the project's scope. In the specific context of 2D dungeon layout generation in popular game engines like Unity, procedural generation techniques assume a pivotal role. By leveraging these techniques and algorithms, developers dynamically generate diverse and engaging dungeon layouts, enhancing player experiences with each unique playthrough.

This thesis explores the application of procedural generation techniques in a modern game engine, assessing their effectiveness in crafting intricate 2D dungeon designs. The primary objectives include evaluating algorithmic adaptability, generating successful and unique outcomes, and generation efficiency. Three Unity prototypes have been developed for this thesis, implementing different procedural generation methods—Wave Function Collapse (WFC), Room-First (Binary Space Partitioning), and Corridor-First (Random Walk).

After a comprehensive exploration and comparison of the challenged posed by all three methods, a final dungeon generator will be further expanded being based off of one of the three. This final generator will incorporate procedural placement of items, enemies, a high-level room system, and 2D top-down shooter gameplay. The purpose of this extra implementation is to prove that the chosen method can be applied to a real-life project.

PREFACE

Embarking on this research journey holds a profound personal significance for me. Growing up immersed in fantasy game worlds like Skyrim, my passion for dungeon crawlers and fantasy games sparked a desire to infuse my own games with unique experiences. Faced with a lack of artistic expertise, I contemplated the prospect of using math and programming to generate game levels based on my creative foundation.

Utilizing skills acquired at DAE Studios during my journey to become a game developer, I initiated the development of a tool to enhance the speed and efficiency of game development. However, this project represents just the starting point of my ambitions. I aim to explore new horizons beyond a conventional dungeon generator.

As someone who has spent countless hours in fantasy game settings, dungeons have always held a captivating allure for me. Games like The Binding of Isaac, Skul: Hero Slayer, Cult of the Lamb, Noita, Enter The Gungeon, and more have not only entertained but also inspired this project. Working on it became a source of joy, even during challenging times throughout researching this topic and the creation of the dungeon generator.

LIST OF FIGURES

INTRODUCTION

The field of video game development undergoes constant evolution, driven by a relentless pursuit of innovation and efficiency. Crafting captivating game environments requires a delicate balance between creativity and practicality. As an enthusiast of dungeon-crawling games and an aspiring game developer, a particular observation ignited the inception of this research.

The catalyst for this project was the intricate and time-consuming nature of manually designing 2D dungeon layouts for games. Having been captivated by fantasy games and inspired by the expansive virtual worlds they offer, it became evident that the conventional approach to level design, while yielding rich experiences, posed a bottleneck in terms of development time and artistic expertise.

The search for a solution led to the exploration of procedural generation techniques—a potent tool in modern game development. The idea of leveraging mathematics and programming to dynamically generate playable 2D dungeon layouts emerged as a promising avenue to address these challenges. Instead of laboriously crafting each level by hand, could a more efficient and creative process be achieved by implementing procedural generation within a contemporary game engine?

This research aims to answer that question: "How can a playable 2D dungeon layout be effectively generated for games by implementing procedural generation techniques using a modern game engine?"

In the pursuit of effective procedural dungeon generation for 2D gameplay within a modern game engine, this bachelor thesis explores three carefully selected methods: Wave Function Collapse, Room-First (BSP), and Corridor-First (Random Walk). The choice of Unity as the prototyping platform was driven by its familiarity and efficiency in rapidly bringing new ideas to life, distinguishing it as the preferred option over Unreal Engine 5.

Motivated by the novelty and intrigue surrounding the Wave Function Collapse method, inspired by its application in Oskar Stålberg's Townscapers, this research seeks to delve into its potential for dungeon generation. Meanwhile, the Room-First and Corridor-First methods were chosen for their demonstrated ability to create coherent layouts with distinguishable rooms, aligning perfectly with the requirements for rogue-like games featuring diverse room types, such as treasure rooms, fighting arenas, and player spawn areas.

The prototyping and exploration of these chosen methods form the core of this research, driven by the ambition to uncover effective and reliable solutions for 2D dungeon generation in video games. The ultimate goal is to create dungeons that are not only playable in their entirety but also provide a unique and varied experience with each generation. The exploration of these methods aims to contribute valuable insights, shedding light on the challenges, advantages, and learning experiences associated with procedural content generation.

To facilitate a comprehensive comparison, this paper meticulously evaluates each method's ability to adhere to a desired dungeon layout, efficiently utilize space, generate to completion, ensure accessibility of rooms, and impact dungeon flow in terms of traversal time and the occurrence of dead-ends. Through this detailed analysis, the research aims to offer developers a nuanced understanding of these procedural generation methods, providing practical insights and considerations for their implementation in game development projects.

LITERATURE STUDY / THEORETICAL FRAMEWORK

In the realm of generating dungeons procedurally, there exists a myriad of creative means to explore. Before initiating my project and finalizing the methods for implementation, I dedicated time to delve into diverse procedural generation techniques that captured my attention during the initial phases of research. These methods are as follows:

TINYKEEP ALGORITHM

A notable dungeon generation method is the TinyKeep algorithm, originally developed for a game of the same name. This method has a very unique approach I have not encountered before.

First, the method randomly places rectangles within the radius of a circle. Then the rectangles are separated either by allowing a physics engine to naturally separate them by adding a physics body with collisions to each room, or by using a separation steering behavior such as in the original implementation made by the developer of TinyKeep.

The rectangles are then snapped to a grid, making sure that the layout conforms to a grid-based positioning and allows for tiles to be easily placed. Following this separation, a handful of rooms are chosen out of the placed rectangles based on a minimum width/height threshold. The midpoints of the selected rooms are then fed into a Delaunay procedure (triangulation). This produces triangles formed from the midpoints of the rooms that are converted into a graph that provides connections to each of the rooms that facilitates nearest neighbor connectivity and avoids overlaps.

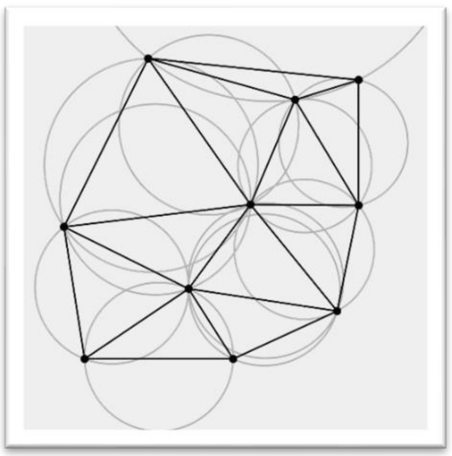


Figure 1: A Delaunay triangulation in the plane with circumcircles shown ("Delaunay Triangulation," 2023)

This graph is then used to create a minimum spanning tree. "The minimum spanning tree will ensure that all main rooms in the dungeon are reachable but also will make it so that they're not all connected as before" (A Adonaac, 2015).

Finally, the corridors are formed in between rooms and rectangles not previously selected as the main rooms are added to the final result if they overlap the corridors.

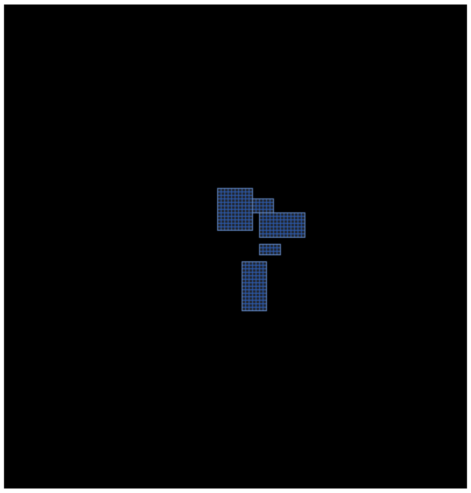


Figure 2: The TinyKeep algorithm from start to finished result (A Adonaac, 2015)

CELLULAR AUTOMATA

Cellular automata find practical application in diverse domains, notably in dungeon generation, particularly for cave-like structures. Their ability to produce organic patterns makes them suitable for this purpose. Advanced cave generation routines may involve one or two passes of a cellular automaton to address specific concerns, such as removing isolated single-point pillars and overall map smoothing. Additionally, cellular automata can be tailored to generate maze-like maps (*Cellular Automata - Procedural Content Generation Wiki*, n.d.). There are many different variations of cellular automata as seen in the Cellular Automata rules lexicon (*Cellular Automata Rules Lexicon - Life*, 2007).

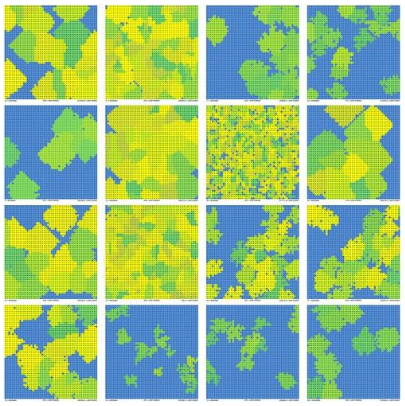


Figure 3: An example of Cellular Automata being used to generate terrain

(*Cellular Automata - Procedural Content Generation Wiki*, n.d.)

Within the domain of computational models, a cellular automaton constitutes a grid structure where each cell possesses a specific state, governed by a predefined rule dictating its transition based on both its individual state and that of its neighboring cells. Uniform rules are applied across all cells, although their initial states may differ. Notably, all state transitions occur simultaneously; cells collectively assess their immediate surroundings and independently determine their subsequent states before undergoing a synchronized update (*Cellular Automata - Procedural Content Generation Wiki*, n.d.). Cellular automata can also be initialized with a noise map such as Perlin Noise, giving the algorithm a starting point.

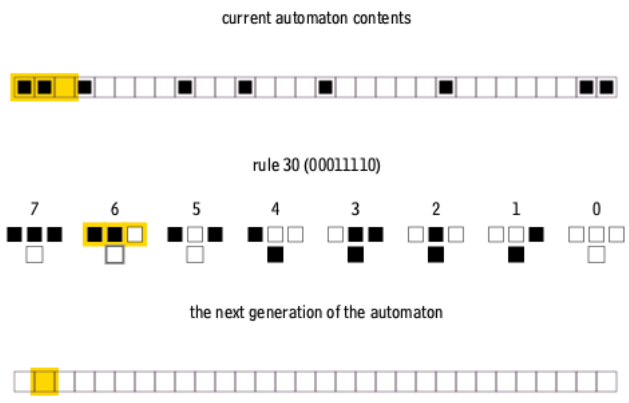

current automaton contents
rule 30 (00011110)
7
6
5
4
3
2
1
0
the next generation of the automaton

Figure 4: An animation of the way the rules of a 1D cellular automaton determine the next generation ("Cellular Automaton," 2024)

WAVE FUNCTION COLLAPSE

This algorithm was developed by Maxim Gumin and released as open source in 2016 (Brian Bucklew, 2019). Wave Function Collapse is a powerful tool in terrain generation and texture synthesis. It can take a small sample in the form of an image and produce a similar-looking, much larger output. You can, for instance, take a small maze and create a much larger maze of a similar layout using this algorithm.

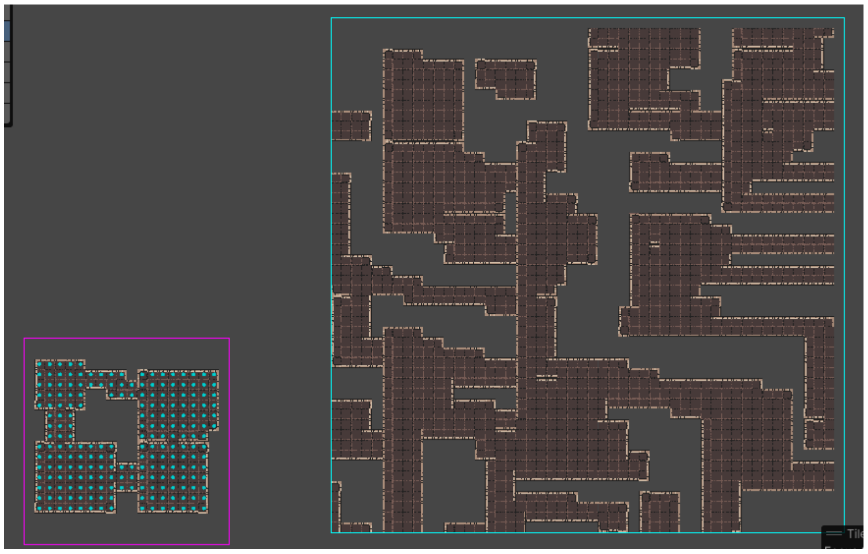


Figure 5: Early testing using a WFC plugin in Unity to generate dungeon layouts using a sample layout

In the context of a grid, where each square represents a cell, the term "domain" refers to the range of possible tile types that a given cell can accommodate. The process involves systematically narrowing down these possibilities until each cell is definitively assigned a specific tile.

The procedure begins with the random selection of a cell. Within that chosen cell, a tile is then randomly assigned from the available options or domain. This initial selection initiates a cascade effect on neighboring cells. The information gleaned from the chosen tile prompts an update to the potential tile types for adjacent cells, a process referred to as propagation (Boris, 2020).

This cycle repeats iteratively. The choice of cells is guided by the "least entropy" heuristic, which prioritizes cells with the least uncertainty or ambiguity in determining the appropriate tile. This systematic approach ensures a gradual refinement of tile possibilities throughout the grid. The iterative adjustments to neighboring cells help resolve the overall puzzle in generating the result as the implications of each tile assignment are considered and propagated through the interconnected cells. The process continues until each cell has only one remaining viable option, indicating the successful completion of the generation (Boris, 2020).

Inevitably, there may be ambiguities or unresolved choices. Addressing these involves additional rules or heuristics to make final decisions, depending on the implementation. The end result is a fully generated grid, where each cell has a determined pattern or value. This grid represents the procedural content, whether it's a terrain, texture, or level layout.

In summary, the Wave Function Collapse algorithm operates by iteratively narrowing down possibilities for each cell based on local observations and propagating constraints to ensure overall coherence. This process facilitates the creation of intricate and contextually rich procedural content, applicable in many fields. However, it's important to note that the algorithm may face challenges leading to failure when conflicting constraints arise. Despite this, it has found applications in various studies, from virtual city generation to NPC pathfinding in games (Cheng Darui et al., 2022). While effective in texture generation, using the algorithm for game level generation may require additional effort and constraints.

ROOM-FIRST (BINARY SPACE PARTITIONING)

There is also the Room-First method. Binary Space Partitioning is a method of dividing an area into smaller pieces. Essentially, you take an area and split it—either vertically or horizontally—into two smaller areas and repeat the process on the smaller areas over and over again until each area is at least as small as your set maximum value. Binary Space Partitioning allows developers to guarantee more evenly-spaced rooms, while making sure you can connect all of them together (Timothy Hely, 2023).

By employing this algorithm, we can divide the total provided space of a dungeon into smaller subsections that are split into smaller subsections until they are too small to be split again but still can fit a room of a defined minimum width and height.

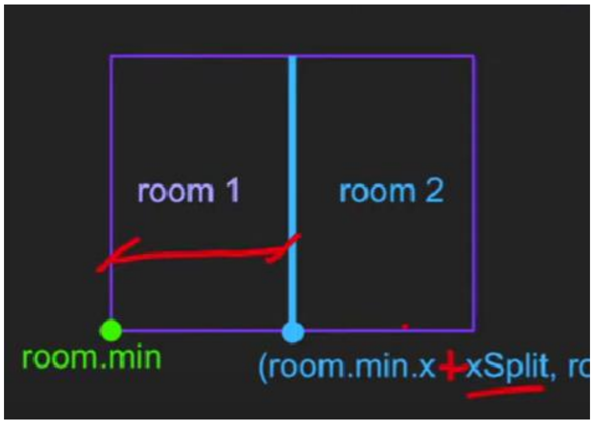


Figure 6: A diagram representing the splitting of a space into two smaler spaces using BSP (Sunny Valley Studio, 2020)

These sections are saved as rooms and floor tiles are placed in the resulting spaces. These rooms are then interconnected through the introduction of corridors. This process begins by selecting a random room’s central position and determining the nearest neighboring room center to it. We then connect these two room centers by checking if the current tile (starting from the room center) is above, below, to the left, or to the right of the other room center. Depending on the results of this check, we walk along the grid placing floor tiles as we go moving up, down, left, and right until reaching the destination. Every time we walk along the grid, we save the position to a list of new corridor floor tile positions. Since this implementation uses hash sets in C#, the saved floor tiles of the corridors are added to the final list of floor tile positions without duplicates. As rooms are connected, their center positions are removed from the list of room centers to be evaluated when finding the closest nearby room to connect a room to. Once all rooms are connected, the tiles for the floors of the corridors are placed in the resulting positions (Sunny Valley Studio, 2020).

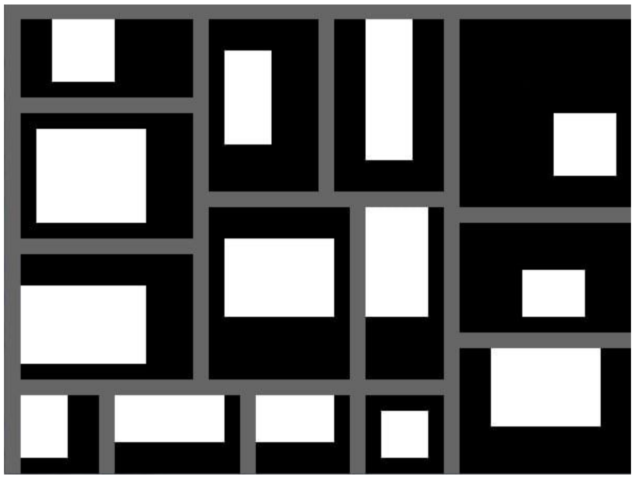


Figure 7: Sample of how BSP splits an area with rooms inside each area or "leaf" {Timothy Hely, 2023}.

Binary Space Partitioning can provide structure, versatility, and logical connectivity. However, it has some limitations at times producing unique, organic variation in its results. As well, there are certain difficulties translating the splitting of areas into 3D. For 2D applications, however, it is certainly a solid choice.

CORRIDOR-FIRST METHOD

In this method, corridors are placed before the rooms and rooms are subsequently placed on the ends of the corridors. This method employs the Random Walk algorithm which can be equated to the developer creating an “agent” that walks in random directions. Originally, this algorithm works by walking in random direction at every step. However, the exact implementation of this algorithm has many variations. In the context of generating corridors, we can make use of a modified version in which the agent only changes directions after walking a certain number of steps.

By storing where the agent walks tile by tile for a length designated as a parameter, and only then changing directions, we can create corridors that are interconnected. To then place the rooms, we create a potential room position at the ends of each corridor before changing direction aka between each iteration of the Random Walk. This method sometimes produces corridors with dead ends, so to account for that we can run a search after generating the corridors and before placing the rooms in which all dead ends have potential room placements stored at the end.

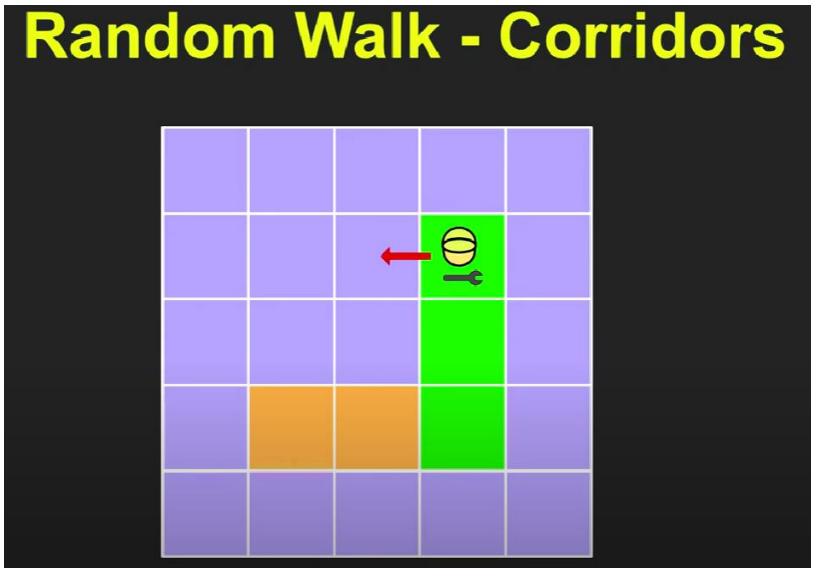


Figure 8: An example of how corridors are generated using Random Walk (Sunny Valley Studio, 2020)

At each potential room position we then use the original implementation of the Random Walk algorithm in which the agent changes directions each step. By changing the amount of steps, making the agent switch to a random position out of the already created tiles after an number of steps, and more, we can create many different diverse room layouts (Sunny Valley Studio, 2020). By making use of hash sets in C# we can also make sure that floor tile positions whether corridor or room tiles are stored in a data container that only allows unique values. This ensures that if at any point we receive a duplicate floor tile position, it is never added twice to the resulting layout before generation.

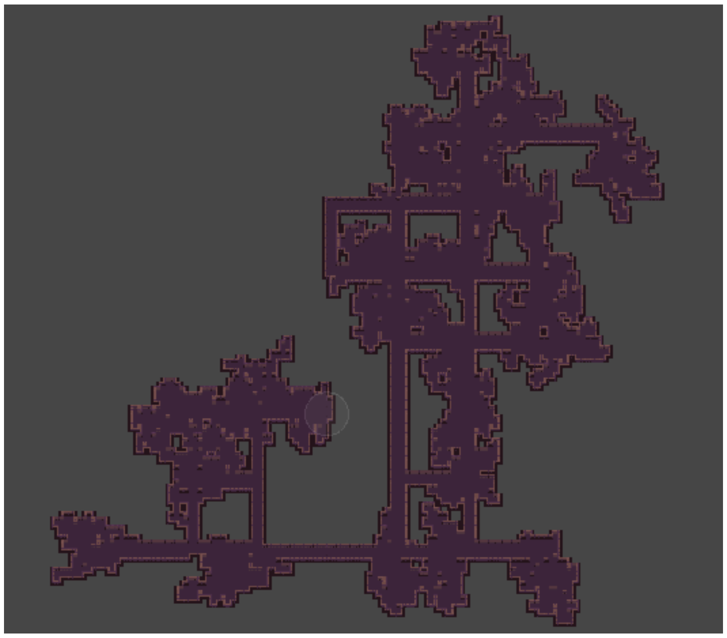


Figure 9: Early results implementing corridor-first generation

This method has the advantage that the corridor length, corridor count, and room generation can be easily configured, allowing the developer to change the look of the dungeon with only a few clicks. This can also be expanded to make the corridors more than one tile wide, incorporating extra passes after the dungeon is generated to make them wider. Corridor-First generation does not have the same control over the splitting of a space, however it does provide a consistent and reliable result for dungeon generation with a great deal of control via parameters and variables. This method can also be complimented with the use of Unity's Scriptable Objects. These are objects that can serve as holders of various information. Essentially, these objects can be used as presets to interchange. These presets can be applied to the algorithms used by this method allowing the developer to quickly change the generation's layout and general appearance. This can easily be interchanged with another preset for the dungeon, allowing different layouts such as island rooms, big dungeons, small dungeons, etc. (Sunny Valley Studio, 2020).

RESEARCH

1. INTRODUCTION

In the following research for this bachelor thesis, we will delve into the creation of the dungeon generators, employing the Wave Function Collapse, Room-First (Binary Space Partitioning), and Corridor-First (Random Walk) methods. Following the successful implementation of each method, a thorough evaluation based on predefined criteria will be conducted to compare their efficacy in dungeon generation.

Our journey into implementing these methods kicks off by understanding the inner workings of each algorithm within the context of a 2D grid-based dungeon generator. This involves tackling challenges such as storing positions for tiles associated with rooms and corridors and deciphering how each algorithm navigates the layout generation process. Once the grid positions are secured, the focus shifts to visualizing the results and placing tiles for floors and walls, achieved through the implementation of sprite sheets featuring individual sprites for each essential tile.

Upon a meticulous comparison and evaluation of these implementations, a method will be chosen for further exploration. This chosen method will be expanded to incorporate 2D top-down shooter gameplay, simulating the real-life application of the dungeon generator in practical projects. The project will also feature a basic procedural placement of items and enemies throughout the dungeon. The procedural item and enemy placement framework will revolve around the concept of distinguishing saved rooms generated by the chosen method, assigning each room a preset layout. For example, a player spawn room will contain no enemies but a specific range of items, serving as the starting point for the player. Conversely, a fighting pit room will host enemies and will not be the player's initial location upon beginning play.

In the project's final stages, the generation process will transition from a button-activated event to being randomly generated each time the project begins, culminating in a comprehensive exploration of procedural dungeon generation with practical gameplay applications.

2. BINARY SPACE PARTITIONING

2.1. IMPLEMENTING THE ALGORITHM

To reiterate on how Binary Space Partitioning works, this algorithm allows us to split a space into smaller sub-sections. In this case, that space is the bounds of the dungeon as defined in the parameters for the generator. To perform this task using BSP, we not only set the size of the initial space but also the desired minimum sizes for the rooms. We then split the dungeon space either vertically or horizontally. The orientation of the split is chosen at random.

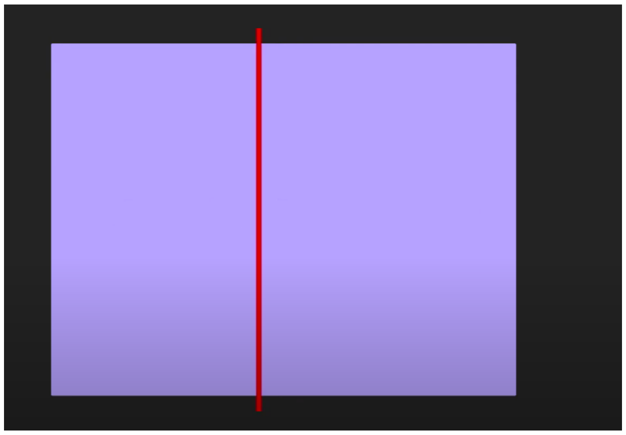


Figure 10: The initial split performed using BSP (Sunny Valley Studio, 2020)

The spaces produced by this initial split are then split further into smaller spaces. So long as the space to split is greater than or equal to twice the minimum height or width, they are split into smaller spaces. If the width of the space is more than or equal to twice the minimum width, it will be split vertically. If the height of the space is more than or equal to twice the minimum height, it will be split horizontally. Given the scenario that both the width and height are more than or equal to the minimum width and height, the random selection will choose whether the width or height will be tested first. This leads to a 50% chance of a split horizontally or vertically.

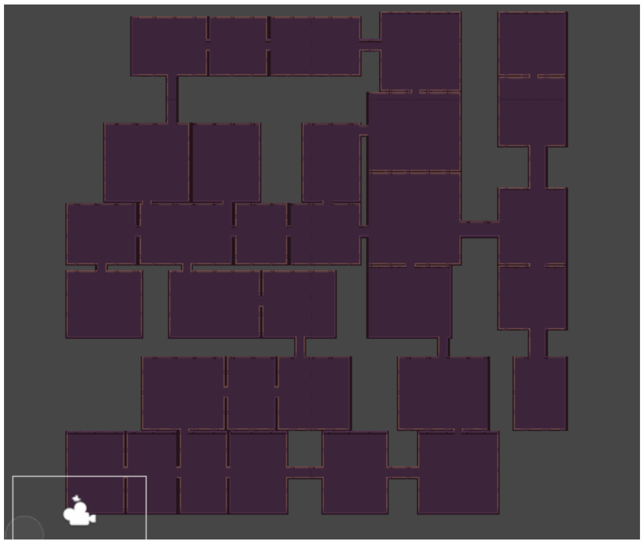


Figure 11: Early results of using BSP in a Room-First approach in the project

By then looping through a queue of the spaces, we can then add the space to a list of rooms should it no longer be able to be split given the minimum width and height of the rooms desired. This sectioning of the dungeon space results in a layout that is structured and coherent with good space utilization.

As well, to ensure that the splitting has a degree of variation, instead of simply splitting them at a constant value or in half, we will make the position of the split a random value. This random value will be a value between one and the size of the space to be split. This will effectively split the space into two different rooms whose dimensions are stored and added to the queue of spaces to be split. The reason the minimum value of the position of the split is one in place of using the minimum width or height of a room is because using those values for the split produces a less random, more grid-like result (Sunny Valley Studio, 2020).

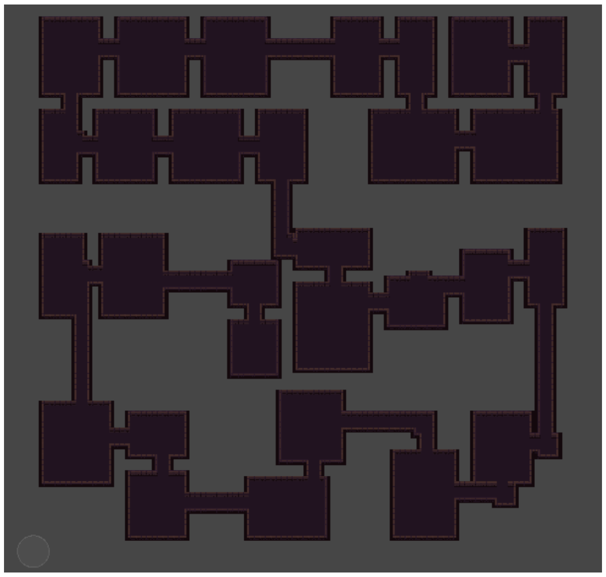


Figure 12: Implementation of Room-First without using Random Walk for the rooms

By designating the width and height of the desired dungeon, the minimum sizes of the rooms, and an offset to increase the separation of the rooms, we can easily generate rooms of a given size once the total dungeon space is split. We finish the generation of the rooms by storing the floor tile positions in a hash set based on the room bounds generated by BSP. We then run that through a method of painting the floor tiles as mentioned below.

2.2 Random Walk Rooms

As stated before, the Random Walk algorithm works by placing an "agent" at a starting position and selecting a random direction for it to walk. This can be configured to walk in how ever many steps in the selected direction is needed before changing directions again. We can also define the amount of total steps the agent should take. The Random Walk algorithm will serve as a method of providing some organic elements into the generation of our rooms.

By allowing this agent to randomly walk and create a path, and following that agent by more agents that start at the end of the path generated by the previous agent, we create abstract yet coherent room layouts that are more visually compelling compared to the normal rectangular counterparts.

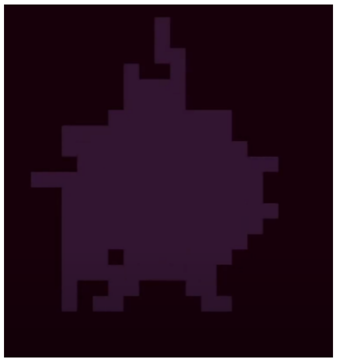
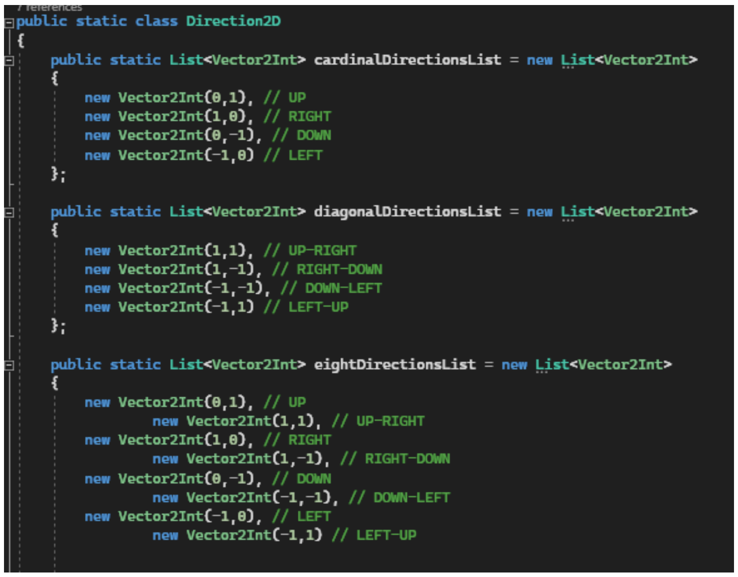


Figure 13: An organic room floor generated using Random Walk {Sunny Valley Studio, 2020}

As this project is in 2D and working with a grid for the placement of tiles that will become the floors and walls of the dungeon, I created a data structure representing the cardinal and diagonal directions in which the agent can travel. These directional data structures are also used in various calculations throughout the project that need to use checks in certain or all directions.



```
public static class Direction2D
{
    public static List<Vector2Int> cardinalDirectionsList = new List<Vector2Int>
    {
        new Vector2Int(0,1), // UP
        new Vector2Int(1,0), // RIGHT
        new Vector2Int(0,-1), // DOWN
        new Vector2Int(-1,0) // LEFT
    };

    public static List<Vector2Int> diagonalDirectionsList = new List<Vector2Int>
    {
        new Vector2Int(1,1), // UP-RIGHT
        new Vector2Int(1,-1), // RIGHT-DOWN
        new Vector2Int(-1,-1), // DOWN-LEFT
        new Vector2Int(-1,1) // LEFT-UP
    };

    public static List<Vector2Int> eightDirectionsList = new List<Vector2Int>
    {
        new Vector2Int(0,1), // UP
                new Vector2Int(1,1), // UP-RIGHT
        new Vector2Int(1,0), // RIGHT
                new Vector2Int(1,-1), // RIGHT-DOWN
        new Vector2Int(0,-1), // DOWN
                new Vector2Int(-1,-1), // DOWN-LEFT
        new Vector2Int(-1,0), // LEFT
                new Vector2Int(-1,1) // LEFT-UP
```

Figure 14: Helper data structures to help with directional calculations such as in room generation or wall placement

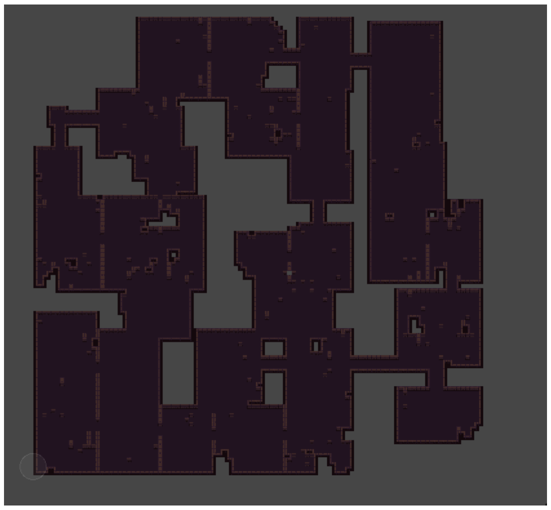


Figure 15: The same parameters with a degree of organic generation using Random Walk

Of course, unlike the above photos this currently does not account for connecting the rooms at all. It simply places rooms in the calculated bounds.

2.3 Connecting the rooms

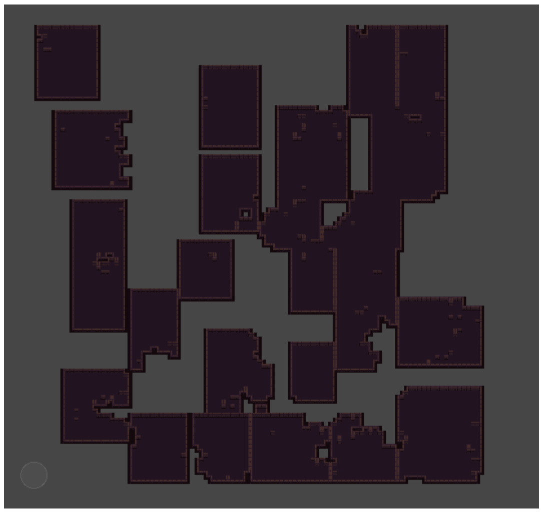


Figure 16: Room-First generation before connecting the rooms

To connect these rooms, we will need to store the center positions of all of the rooms generated. To do this, we will select a random room center and remove it from a list of room centers that are currently unvisited. We will loop through all of our room center positions and find the closest point to the current room center in our list of unvisited room centers. Furthermore, we will then remove this closest room center position from the list of unvisited room centers.

Upon finding the first random room center and its closest other room center, we will connect these two rooms with a corridor. To do this, we will begin at the first room's center tile. While our vertical position is not the same as the vertical position of the other room's center, we will increase or decrease the vertical position of the current tile placing a floor tile position every time we change our position. While the horizontal position is not the same as the horizontal position of the other room's center, we will either move the position to the left or right to become closer to that destination. Therefore, we will first move vertically until we match the destination and followingly we will move horizontally until we match the destination's position.

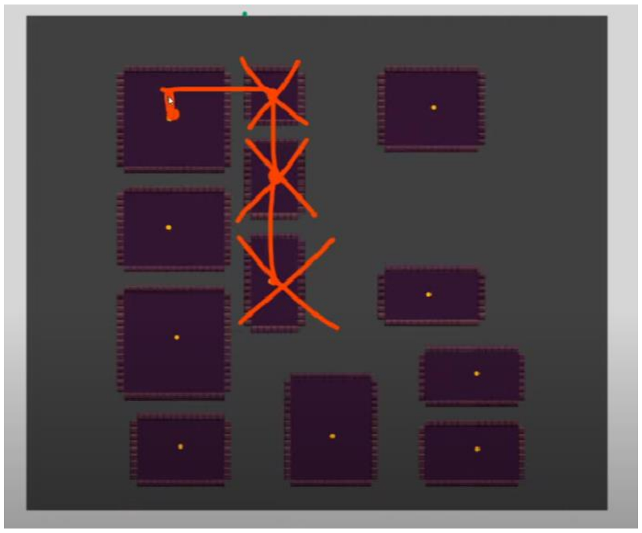


Figure 17: A visual representation of how the connection of each room works sequentially (Sunny Valley Studio, 2020)

Upon generating all of the rooms, connecting them via corridor pathways, and finally placing all of the floor and wall tiles, this method is complete in implementation. The general layout of the dungeon can be changed relatively easy using preset values in the form of Scriptable Objects that dictate the room sizes and certain aspects of generation. These presets, for example, change the values fed into the Random Walk algorithm when using organic rooms.

3. PAINTING THE FLOOR TILES

Once we know where the position of every floor tile should be, we can use a grid and tile map to keep a system of 2D cells that are uniform in size and evenly spaced in position.

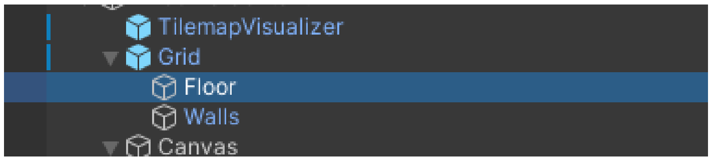
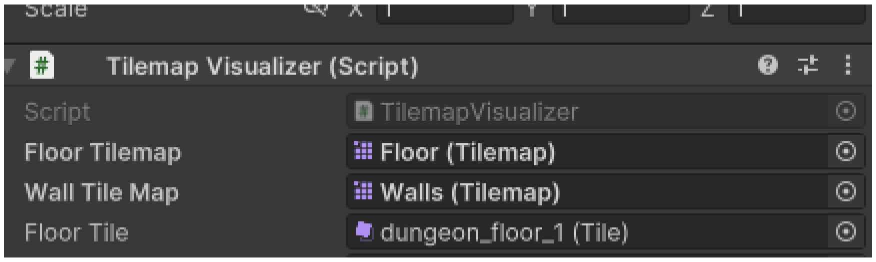


Figure 18: The setup of the grid and tilemaps


Tilemap Visualizer (Script)

Script TilemapVisualizer
Floor Tilemap Floor (Tilemap)
Wall Tile Map Walls (Tilemap)
Floor Tile dungeon_floor_1 (Tile)

Figure 19: The tilemap visualizer stores the location of the tilemaps and the tile sprites to be placed

By storing a reference to the tile map for the floor we can automatically paint a floor tile defined in the Tilemap Visualizer (which paints the tiles onto the grid, visualizing the generation of the dungeon) at every position where our floor should be, adding it to the floor tile map.

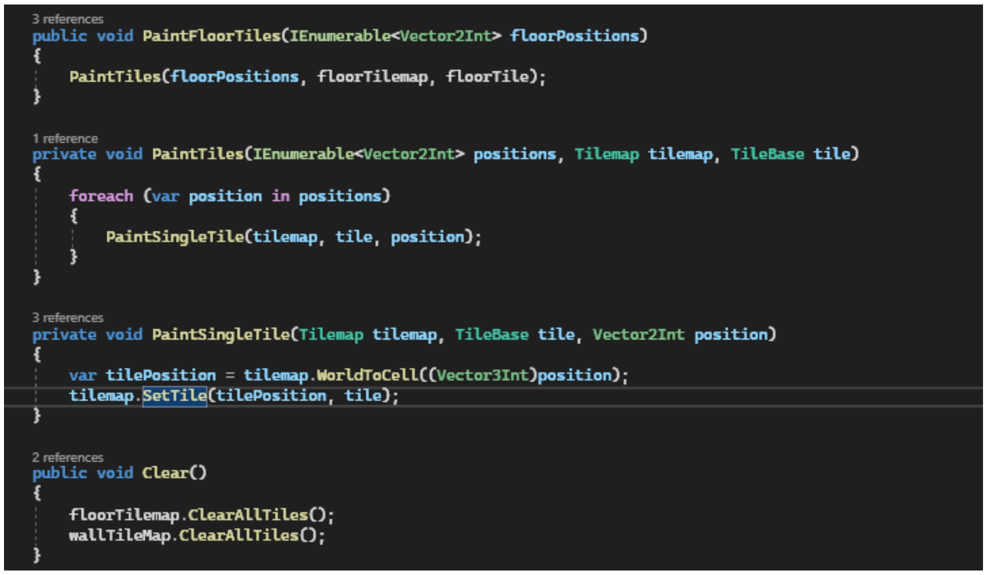

```
3 references
public void PaintFloorTiles(IEnumerable<Vector2Int> floorPositions)
{
    PaintTiles(floorPositions, floorTilemap, floorTile);
}

1 reference
private void PaintTiles(IEnumerable<Vector2Int> positions, Tilemap tilemap, TileBase tile)
{
    foreach (var position in positions)
    {
        PaintSingleTile(tilemap, tile, position);
    }
}

3 references
private void PaintSingleTile(Tilemap tilemap, TileBase tile, Vector2Int position)
{
    var tilePosition = tilemap.WorldToCell((Vector3Int)position);
    tilemap.SetTile(tilePosition, tile);
}

2 references
public void Clear()
{
    floorTilemap.ClearAllTiles();
    wallTileMap.ClearAllTiles();
}
```

Figure 20: A code snippet for painting tiles in a tile map

As seen in the code snippet above, painting a tile onto a tile map is relatively easy. You just have to set a tile at a position to the tile sprite you desire. I also include functions for painting many different tiles in a tile map at a time, as well as clearing the tile maps when regenerating the dungeon.

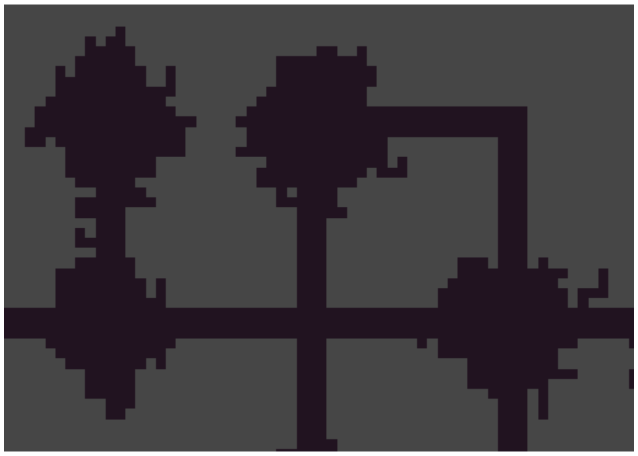


Figure 21: The result after painting the tiles onto the floor tilemap

4. PLACING THE WALLS

4.1. WALL GENERATOR

To begin placing walls around the generator dungeon layouts, we need to create a Wall Generator that takes a look at the surrounding neighbors of a generated floor tile and test what kind of wall needs to be placed. By looping through the neighboring tiles of a floor tile and testing if the neighboring tile is within the list of floor tile positions, we can test what neighbors contain valid floor tiles and where there is a nearby empty space that needs to be the placement of a wall tile.

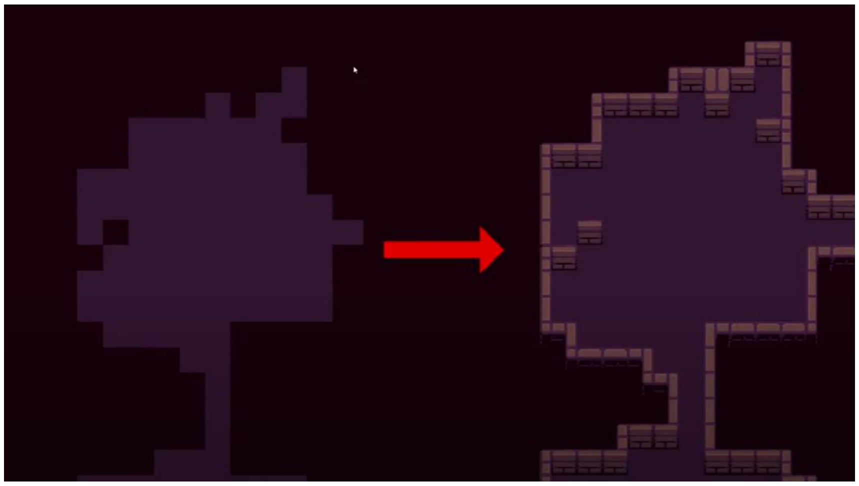


Figure 22: The desired result after calculating which walls with what orientation should be placed where (Sunny Valley Studio, 2020).

To begin, we take the generated floor layout and loop through every single floor tile position that has been placed. For each floor tile, we test if the tiles in any of the four cardinal directions around that floor tile contain a generated floor tile. Essentially, if a floor tile contains a space that does not contain a neighboring floor tile, it is a space where a wall must be placed.

In order to store the information on what neighbors a floor tile has, we will be using a binary representation of the neighbors. If a position contains a neighbor, that place will be a 1 in our binary representation. If the position next to a floor tile does not contain a neighbor, it is a 0. This information will be stored in a string which is easily converted into a binary value in C#.

By storing for example “1010” and associating each number with being a tile in our grid neighboring the current floor tile in a given direction, we can store if a floor tile has neighboring floor tiles or not.

In the case of “1010” the current floor tile has a neighboring floor tile generated directly above it, not to the right of it, a neighbor below it, and no neighbor to the left of it.

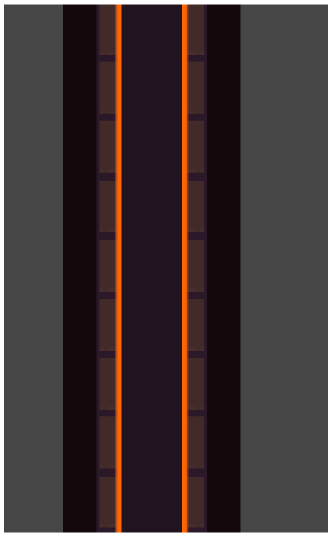


Figure 23: A corridor of tiles in which have neighbors above and below them but not to the left or right

Going by that logic, a wall must be placed to the right and left of this floor tile given that it already has another floor tile directly above and below it.

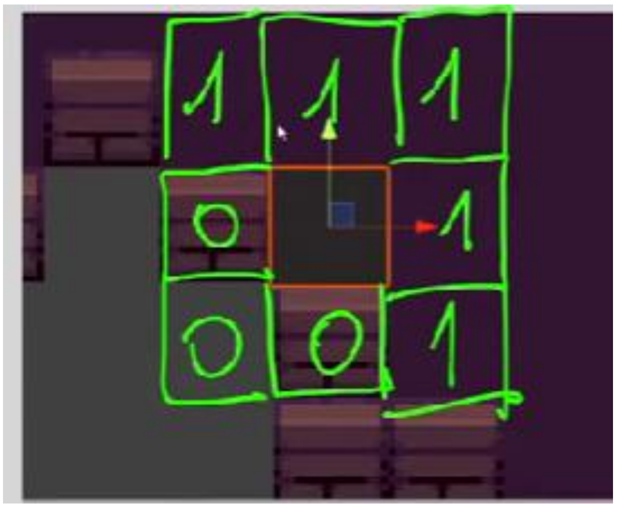
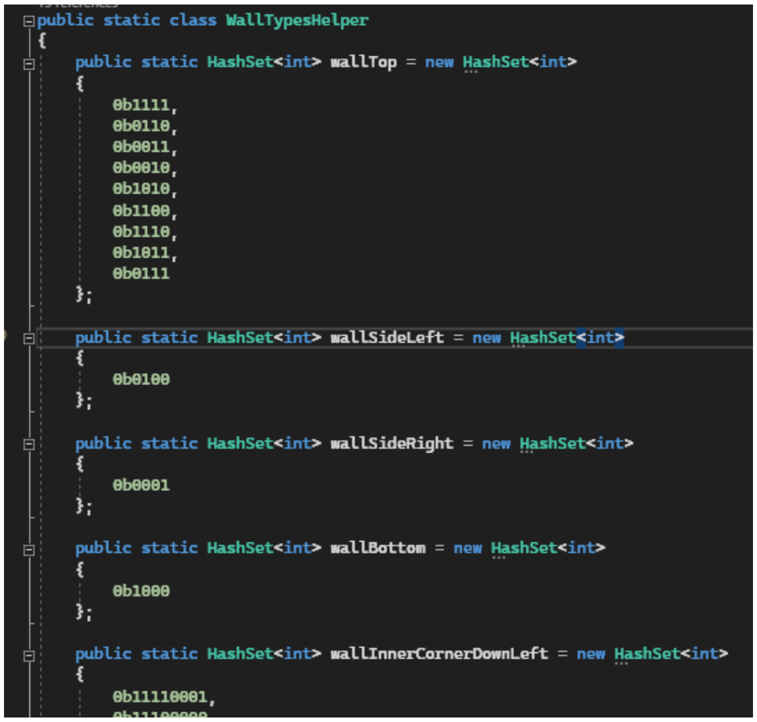


Figure 24: A visual representation of a floor tile's neighbors (Sunny Valley Studio, 2020)

Firstly, we will check the four cardinal directions and create the appropriately oriented walls based on the neighbors of the floor tiles. The specific sprite needed for the tile is pre-defined in the Tilemap Visualizer created earlier. Secondly, we will then use the same logic and binary representation method to check the diagonal neighbors of the floor tiles and find what walls need to be placed in which directions. We also use a helper class which contains all possible scenarios of where a wall should be placed in binary representation.

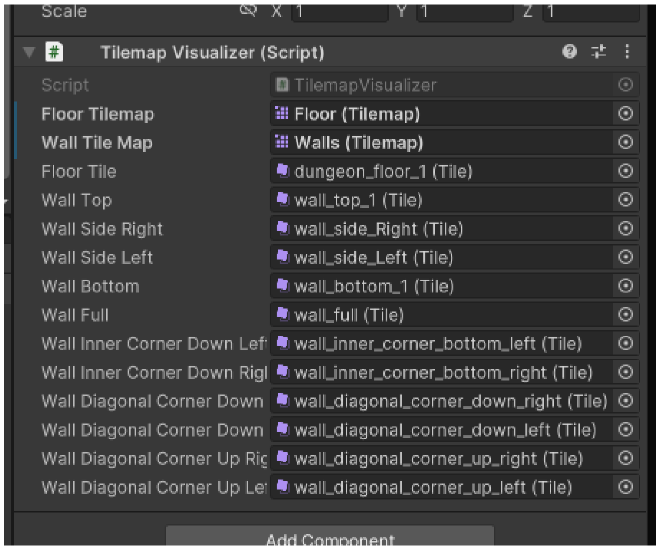


```
public static class WallTypesHelper
{
    public static HashSet<int> wallTop = new HashSet<int>
    {
        0b1111,
        0b0110,
        0b0011,
        0b0010,
        0b1010,
        0b1100,
        0b1110,
        0b1011,
        0b0111
    };

    public static HashSet<int> wallSideLeft = new HashSet<int>
    {
        0b0100
    };

    public static HashSet<int> wallSideRight = new HashSet<int>
    {
        0b0001
    };

    public static HashSet<int> wallBottom = new HashSet<int>
    {
        0b1000
    };

    public static HashSet<int> wallInnerCornerDownLeft = new HashSet<int>
    {
        0b11110001,
```

Figure 25: Binary representations of all possible neighbor scenarios

By checking the stored information on the neighbors of a floor tile and comparing it to these binary representations, we can assess what type of wall needs to be placed where. To help with various directional calculations including where a wall should be placed relative to a floor tile in this case, this project includes data structures storing the cardinal and diagonal directions. These structures contain an X and Y value depending on the orientation described.

These data structures enable us to loop through all possible directions in less code compared to manually having a long list of nested if's evaluating the directions line-by-line. Essentially, we take the current floor tile's position and add or subtract the directional X and Y values depending on the orientation to get the position of the to-be evaluated tile position. Then, depending on the orientation of the wall position compared to the evaluated floor tile, we paint the wall tile in the grid.


Scale X 1 Y 1 Z 1

Tilemap Visualizer (Script)

Script	TilemapVisualizer
Floor Tilemap	Floor (Tilemap)
Wall Tile Map	Walls (Tilemap)
Floor Tile	dungeon_floor_1 (Tile)
Wall Top	wall_top_1 (Tile)
Wall Side Right	wall_side_Right (Tile)
Wall Side Left	wall_side_Left (Tile)
Wall Bottom	wall_bottom_1 (Tile)
Wall Full	wall_full (Tile)
Wall Inner Corner Down Lef	wall_inner_corner_bottom_left (Tile)
Wall Inner Corner Down Rigl	wall_inner_corner_bottom_right (Tile)
Wall Diagonal Corner Down	wall_diagonal_corner_down_right (Tile)
Wall Diagonal Corner Down	wall_diagonal_corner_down_left (Tile)
Wall Diagonal Corner Up Rig	wall_diagonal_corner_up_right (Tile)
Wall Diagonal Corner Up Le	wall_diagonal_corner_up_left (Tile)

Figure 26: A list of sprites for every wall orientation

The sprites for every possible wall are interchangeable but must be set in the Tilemap Visualizer. The walls are then painted in the Walls tile map stored in the grid of the dungeon.

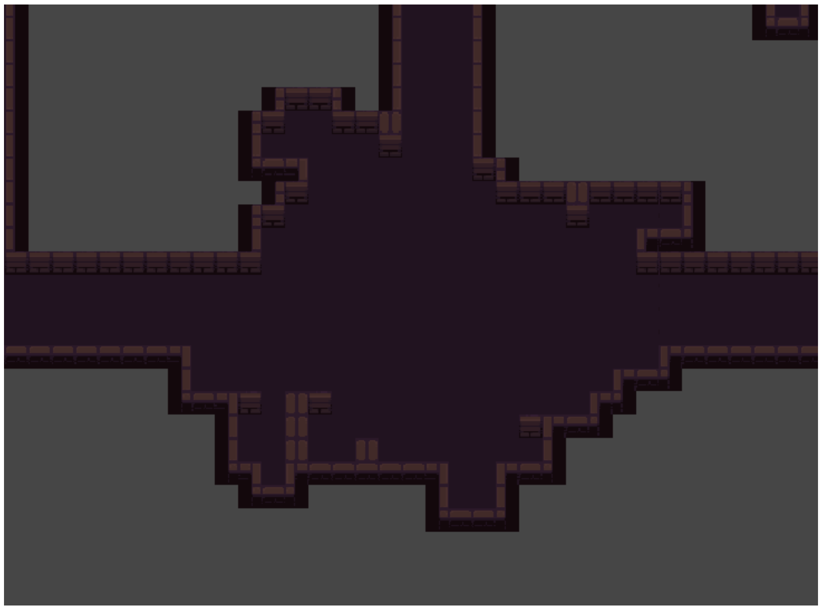
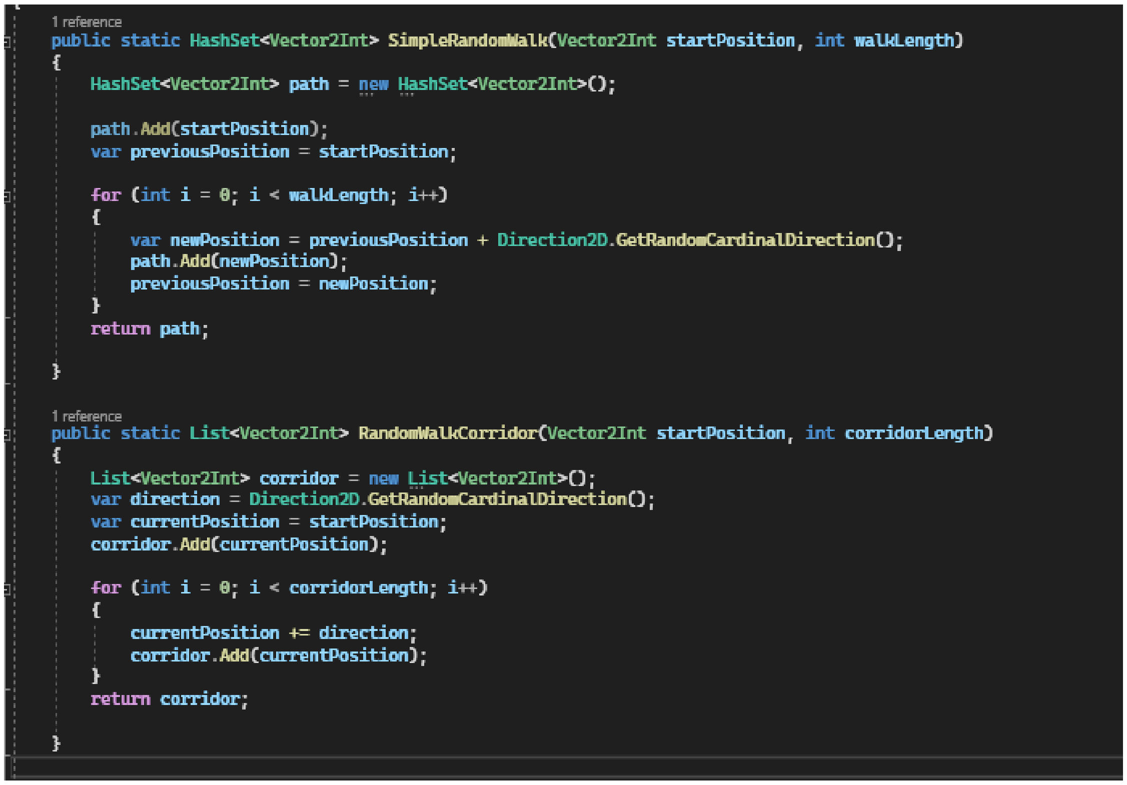


Figure 27: The final result after placing all of the walls

5. CORRIDOR-FIRST

As discovered in the preliminary research, this method of generating a dungeon works by placing corridors using a modified use of the Random Walk algorithm. Instead of randomly picking a direction each time the “agent” walks a step, the agent will instead pick a random direction and continue walking in the direction chosen for the length of the corridor as is designated in the parameters for the Corridor-First dungeon generator.

Each iteration the next corridor is placed on the end of the previous one, allowing all of the generated corridor paths to be interconnected. We can tweak the length of the corridors, the amount of corridors to be placed, and more.



```
1 reference
public static HashSet<Vector2Int> SimpleRandomWalk(Vector2Int startPosition, int walkLength)
{
    HashSet<Vector2Int> path = new HashSet<Vector2Int>();

    path.Add(startPosition);
    var previousPosition = startPosition;

    for (int i = 0; i < walkLength; i++)
    {
        var newPosition = previousPosition + Direction2D.GetRandomCardinalDirection();
        path.Add(newPosition);
        previousPosition = newPosition;
    }
    return path;

}

1 reference
public static List<Vector2Int> RandomWalkCorridor(Vector2Int startPosition, int corridorLength)
{
    List<Vector2Int> corridor = new List<Vector2Int>();
    var direction = Direction2D.GetRandomCardinalDirection();
    var currentPosition = startPosition;
    corridor.Add(currentPosition);

    for (int i = 0; i < corridorLength; i++)
    {
        currentPosition += direction;
        corridor.Add(currentPosition);
    }
    return corridor;

}
```

Figure 28: The difference in code between a simple Random Walk and the modified version for corridors

After running the above code multiple times, each time at the end of the previous iteration, we have multiple connected corridors but no rooms in our dungeons. Therefore, we must then modify the dungeon generator to place rooms at the ends of every corridor.

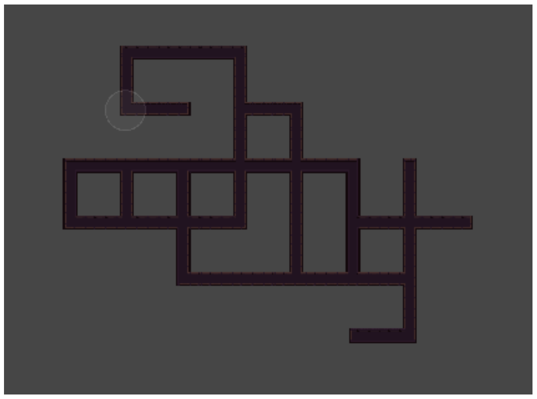


Figure 29: Corridor-First method before generating rooms at the stored positions

To do this, we store a hash set (an unordered data structure that only allows unique elements) of the potential room positions. This hash set will store a unique list of Vector2Int variables that represent the X and Y positions of the room positions. We can also store a parameter to determine what percent of the dungeon's potential room positions should contain rooms. This affects how populated the dungeon is by rooms (Sunny Valley Studio, 2020).

The room positions will be added each time we create a new corridor using Random Walk. Once we ascertain the starting position of a new corridor, we store the first tile of the corridor as a potential room position. At the end of our generation of the corridors, we simply run another pass of Random Walk in the original method to create a randomly generated room at each potential position.

The positions for the tiles to be placed for the rooms is then added to the total hash set of all floor positions, however only unique positions are added making use of the Union function of hash sets. This ensures no duplicate values generating the floors of the dungeon. The aforementioned Wall Generator is then applied to the finished generation of the floor layout. However, should we want to remove the organic appearance of the rooms we can simply generate a rectangular room within a minimum and maximum random width and height.

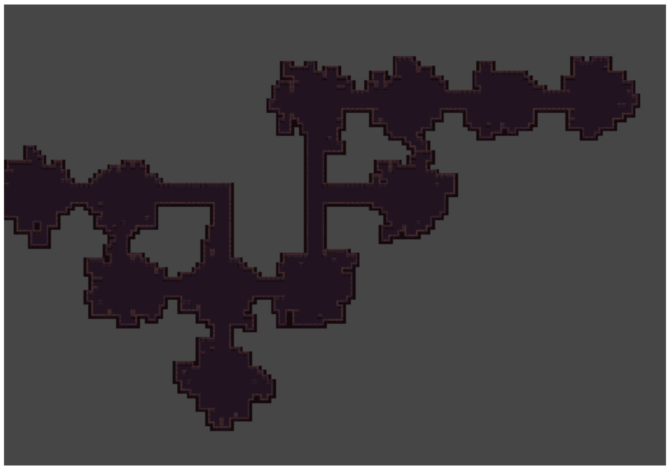


Figure 30: Corridor-First generation with Random Walk organic rooms

In my implementation, I allow a check box in the parameters for the Corridor-First dungeon generator which determines if the rooms should be generated using Random Walk or not. I also created my implementation by instead of allowing a minimum and maximum width and height for the rectangular rooms, I simply used the parameters for the chosen dungeon layout used by the Random Walk algorithm as the guidelines for the size of the rooms.

For example, the size of a room uses the parameter of the Walk Length aka how far a random walk should move in one iteration for the width and height. This produces a similar layout as the generation created using the Random Walk organic rooms while making the rooms rectangular as is seen in many different dungeon generators.

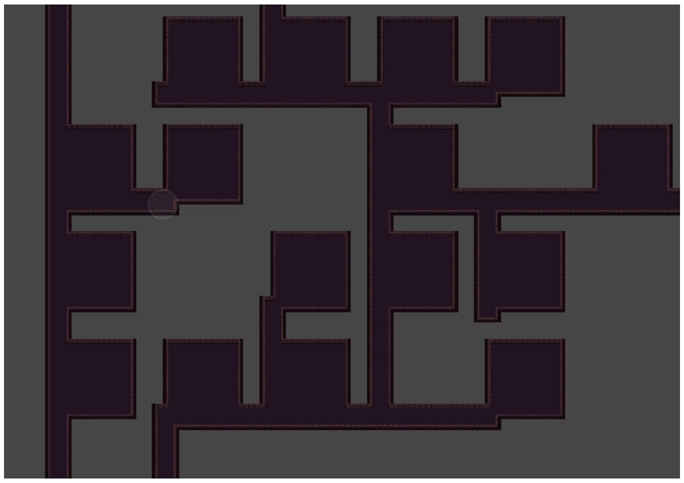


Figure 31: The same Corridor-First implementation without organic rooms

This method of generation does come with an issue. Sometimes corridors are generated with dead ends where rooms are not generated.

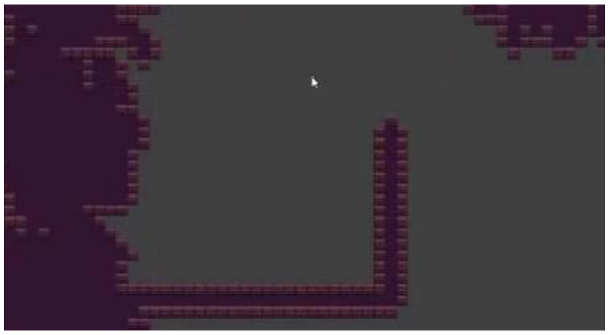


Figure 32: Early implementation demonstrating a dead-end in generation (Sunny Valley Studio, 2020)

To prevent this, we must modify the dungeon generator to account for this. After creating the rooms for our dungeon, we will create a list of dead-end positions. For each floor tile position, we will check the neighbors of the current floor tile in all directions and if the number of neighboring tiles is only one, we know it is a location of a dead-end. While there are not usually a lot of dead-ends, we can now create a new room at each dead-end ensuring the player does not traverse a whole corridor only to find nothing.

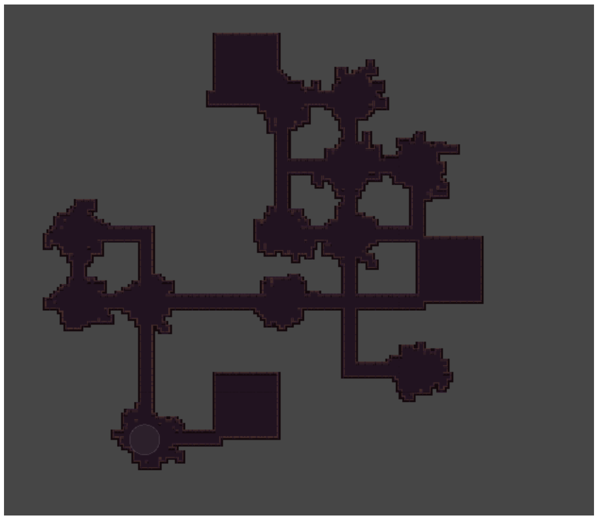


Figure 33: An example where only the dead-end rooms don't use organic generation

In the above example, I ran the corridor-first dungeon generator making only rooms generated at dead-ends not implement Random Walk. This way, I can clearly visualize each time a dead-end is found and a room is generated at its position.

Once all dead-ends are accounted for, all rooms and corridors are generated, and floor tiles along with wall tiles are painted, this method is complete. As with the Rooms-First method, the layout of this dungeon can easily be changed using preset values that are fed interchangeably into the Random Walk algorithm. Whether it is a big dungeon with overlapping rooms, island-rooms that are medium-sized and separated, or a small dungeon, these layouts can be easily switched out.

6. WAVE FUNCTION COLLAPSE

Due to the limited scope of this project in consideration with time constraints, the created project using Wave Function Collapse uses a free Unity plugin provided by selfsame on the website Itch.io. (selfsame, 2020). Researching into WFC I have found limited resources in the field of dungeon generators. While it has been done, it is evident that there are many difficulties in modifying this algorithm to this specific use case.

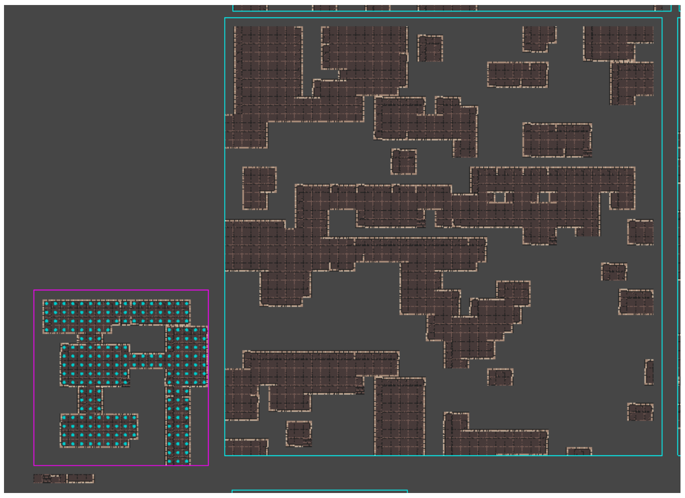


Figure 34: Using a sample of a dungeon layout to create a full level using WFC

The way that the plugin works is that the user must paint a grid of tiles with the appropriate floors and walls in a sort of “sample” to be used for training the algorithm. The user can then use two different methods of registering what tiles should be placed next to each other. One method entails manually typing a list of possible neighbors to a certain tile and the other calculates the neighbors in your sample automatically to provide the algorithm the guidelines on how to generate a result which logically follows the same logic as the sample. For example, if a door should be neighboring a specific floor tile and should not be neighboring another, you must either define this or have the plugin attempt to register this logic automatically. The key to getting Wave Function Collapse to generate a good result is providing a good sample.

To test this plugin and attempt to generate a functional dungeon, I trained the algorithm on many different samples with different layouts.

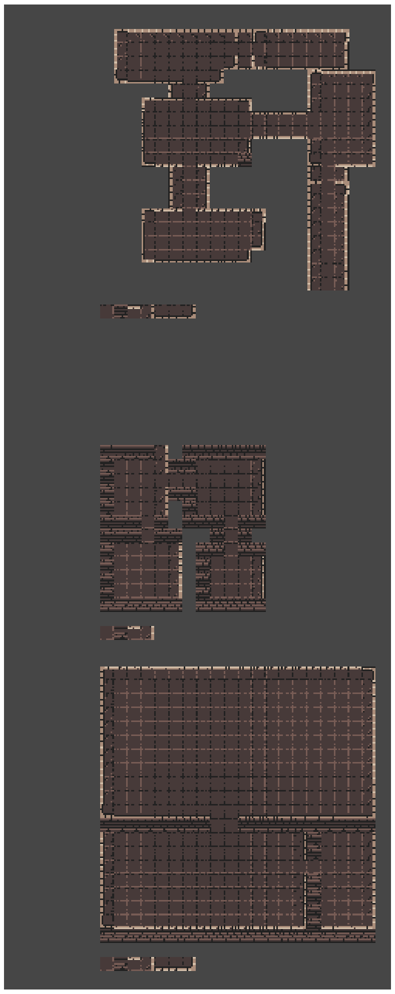


Figure 35: A few samples used to train WFC

In the experiments with this plugin, it was not certain that the results would provide a coherent and playable dungeon all of the time. I often had to change the sample a lot, rapidly trying new layouts to try getting better results. Due to the scope of this research and the time spent implementing the other methods, I was not able to experiment expanding this plugin a lot. In the future, I would love to attempt to test the results of the generation and add constraints and new code to ensure the result is playable at the end of every generation.

However, WFC is a useful tool in generating many other things. It has had great success creating varied and unique terrains as well as other formats of levels. In my experience, while it is useful in many cases, it is difficult to manage when it comes to dungeon generation. For example, it is rather useful in creating mazes, biomes, and images from samples. Another very successful implementation using WFC is the Townscapers application created by Oskar Stålberg (Sweden Game Arena, 2021) in which the player is able to create dynamic town environments generated using this algorithm.

Figure 36: Townscapers, using WFC to generate town environments dynamically.

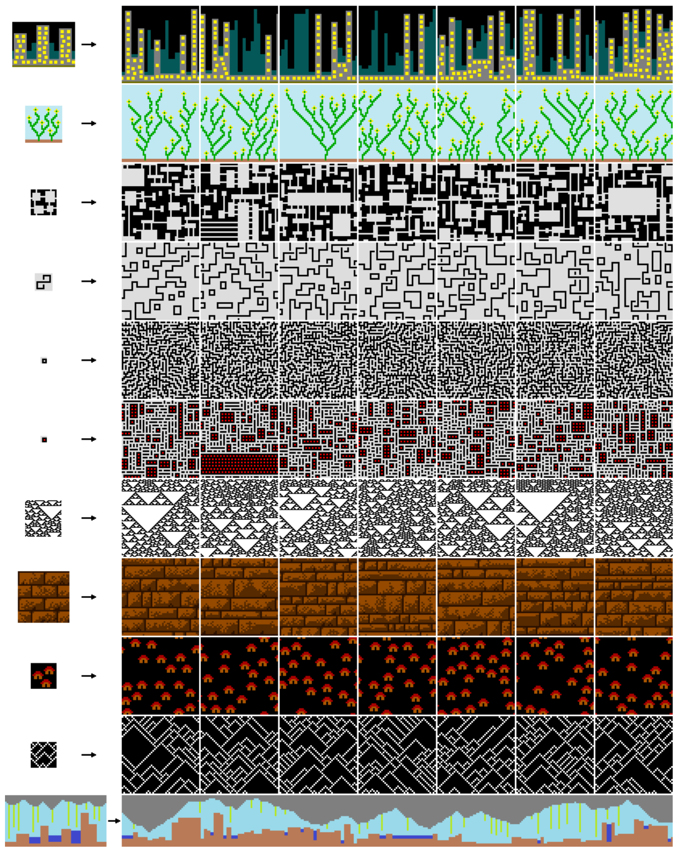


Figure 37: Useful examples of WFC for image creation from a sample provided by the creator of WFC (Gumin, 2022).

7. BEGINNING THE EXPERIMENT

With the completion of all three implementations of the chosen methods for dungeon generation, the phase for evaluating the efficacy of each method based on certain criteria can begin. The next following sections will be answering a set of three different hypotheses focused on different aspects of the dungeon generators and the critical comparison of each method. Once these hypotheses have been proven or disproven, a method has been chosen to be focused on and expanded as one of the last steps of this research. The expansion on the chosen method includes 2D top-down shooter gameplay as well as the procedural placement of items, enemies, and the player.

7.1 USER-DEFINED LAYOUTS AND STRUCTURAL PATTERNS

The first hypothesis to be answered is as follows:

"The Binary Space Partitioning (BSP) algorithm will demonstrate superior adaptability and ease of integration with user-defined constraints in procedural dungeon layout generation compared to Wave Function Collapse (WFC), specifically excelling in scenarios where intricate, user-prescribed structural patterns are a priority."

To confirm or disprove this hypothesis, I used the prototype projects incorporating Wave Function Collapse and Binary Space Partitioning. Using these prototypes, I attempted to produce results with a specific layout or structure. For example, a dungeon with island-like rooms. These rooms must have a layout that consists of similar sized medium/small rooms placed either next to each other or with a slight separation using corridors. This will produce a dungeon layout that is generally evenly spaced out and does not have rooms overlapping each other.

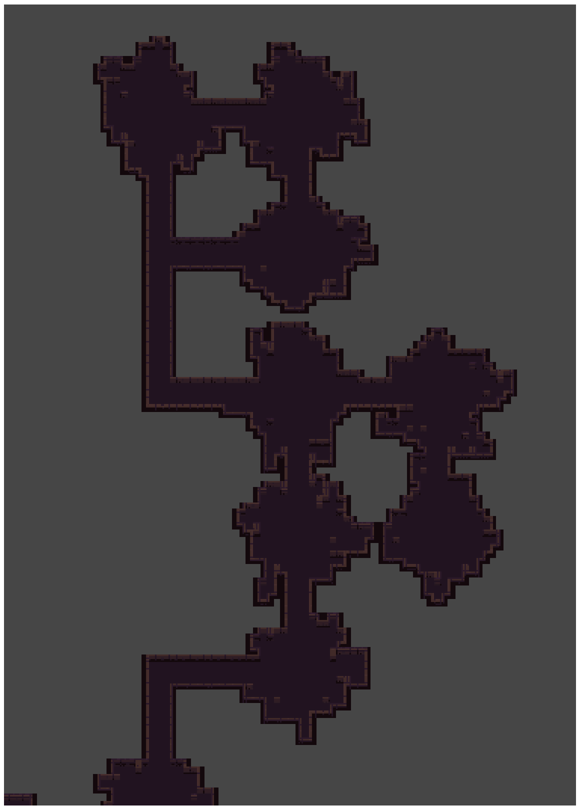
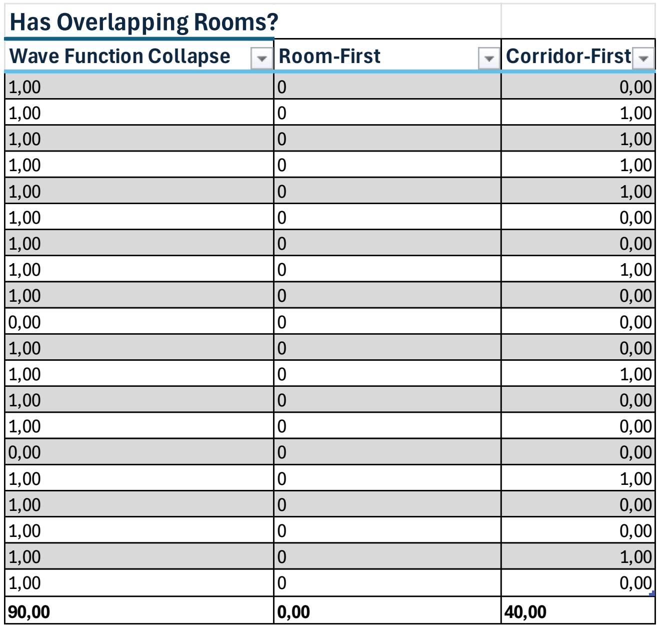


Figure 38: An example of an island-room dungeon layout using Corridor-First generation

The initial hypothesis compares Room-First and Wave Function Collapse, however after implementing all three methods I expanded the experiment to also include Corridor-First in the evaluation. By using a preset of the same values to feed into both Corridor-First and Room-First methods, I was able to produce results with both prototypes using the same data for the layouts. These are the values mentioned previously that are used by the Random Walk algorithm to determine the length of tiles the algorithm should walk as well as the amount of iterations. However, using Wave Function Collapse I produced many different samples to train the algorithm that were based on an island-room layout until I found the one with the best results.

To evaluate if a method successfully produced this user-defined layout, I conducted testing on the amount of overlapping room tiles as well as the degree of variation in room sizes.



Has Overlapping Rooms?		
Wave Function Collapse	Room-First	Corridor-First
1,00	0	0,00
1,00	0	1,00
1,00	0	1,00
1,00	0	1,00
1,00	0	1,00
1,00	0	0,00
1,00	0	0,00
1,00	0	1,00
1,00	0	0,00
0,00	0	0,00
1,00	0	0,00
1,00	0	1,00
1,00	0	0,00
1,00	0	0,00
0,00	0	0,00
1,00	0	1,00
1,00	0	0,00
1,00	0	0,00
1,00	0	1,00
1,00	0	0,00
90,00	**0,00**	**40,00**

Figure 39: The frequency each method produced overlapping rooms

The first test conducted was simply to test the frequency in which each method produced overlapping rooms. Each row in the above data is a different iteration using the same parameters of generation.

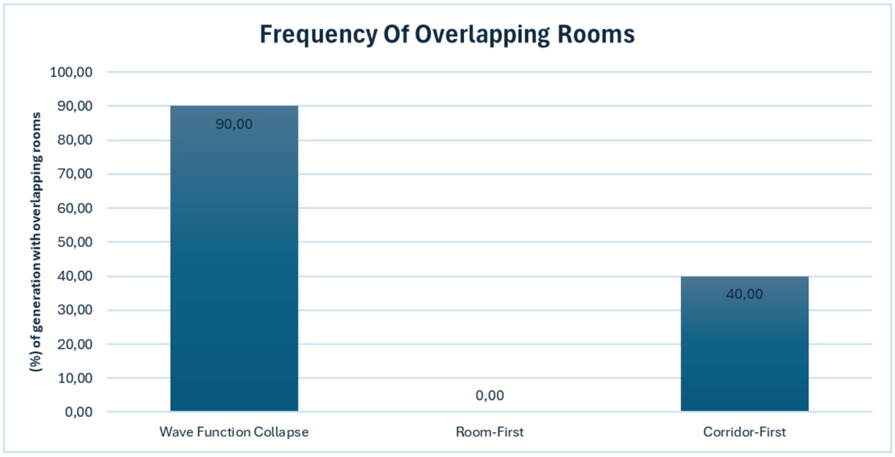


Figure 40: The frequency of overlapping rooms visualized in a bar graph

From the testing I conducted, an implementation of Wave Function Collapse without added constraints and specific code for dungeon generation very often produced overlapping rooms. In fact, ninety percent of the time there was at least one overlapping room in which the generation produced two rooms on top of each other. This test was conducted with twenty total iterations. In these twenty tests conducted Corridor-First produced overlapping rooms forty percent of the time and Room-First never produced an overlapping room. This is largely due to the way that the algorithm sections off the space of the dungeon. In this particular layout, Room-First is considered in my testing the best method.

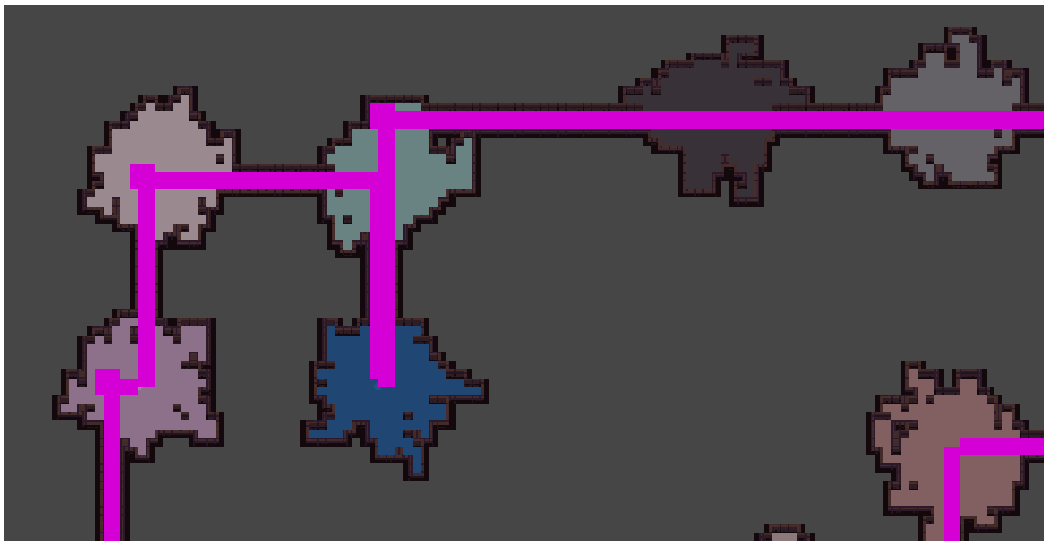


Figure 41: An example of island-room layout using Room-First

In the above example, Room-First generation was used to produce this result using the specified layout parameters and each room has a bit of corridor between them. The spaces with different colors represent the rooms while the lines of pink represent the corridors.

To continue testing if these methods can produce a specified layout defined by the user, I also calculated the amount of overlapping tiles in a percentage per the amount of tiles in the rooms.

Total Overlap Weight		
Wave Function Collapse	Room-First (BSP)	Corridor-First
0,84	0	0,00
0,57	0	0,11
1,65	0	0,08
1,12	0	0,10
0,79	0	0,03
2,67	0	0,00
1,34	0	0,00
1,56	0	0,23
0,92	0	0,00
0,00	0	0,00
0,46	0	0,00
0,13	0	0,14
0,21	0	0,00
0,58	0	0,00
0,00	0	0,00
0,31	0	0,03
0,69	0	0,00
1,27	0	0,00
2,13	0	0,31
0,94	0	0,00
18,18	**0**	**1,03**

Figure 42: The total overlap weight of each method

Here the amount of overlapping tiles is compared to the total amount of tiles in the rooms themselves. Each overlapping pair of rooms is given a weight. For example, if a pair of rooms has 10% of the tiles overlapping then the overlap weight will be 0.1. This is a numerical representation of the percent of overlapping tiles, therefore if a pair of rooms has 50% of the tiles overlapping out of the total amount of tiles, the weight will amount to 0.5. By conducting another 20 tests using all three methods and adding these weights to a total, we can evaluate the amount of overlapping tiles generated by the dungeon.

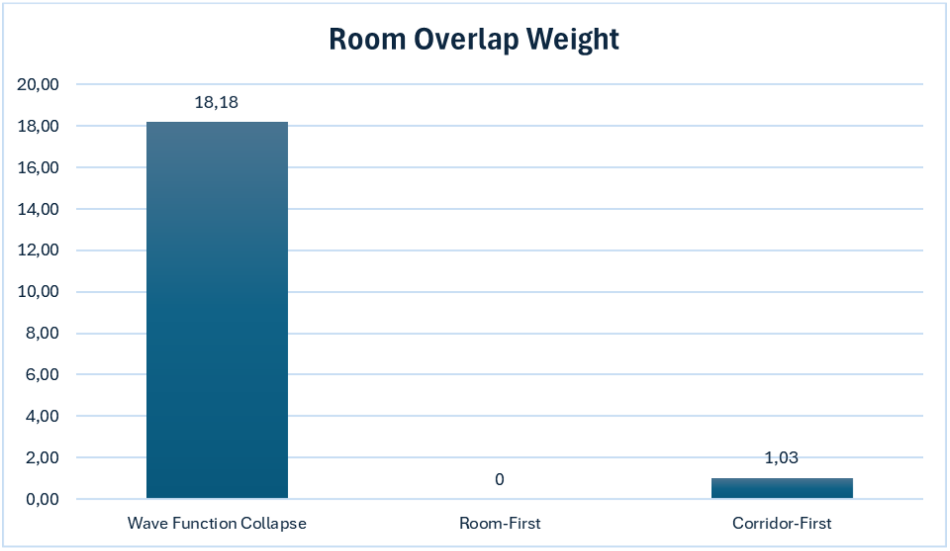
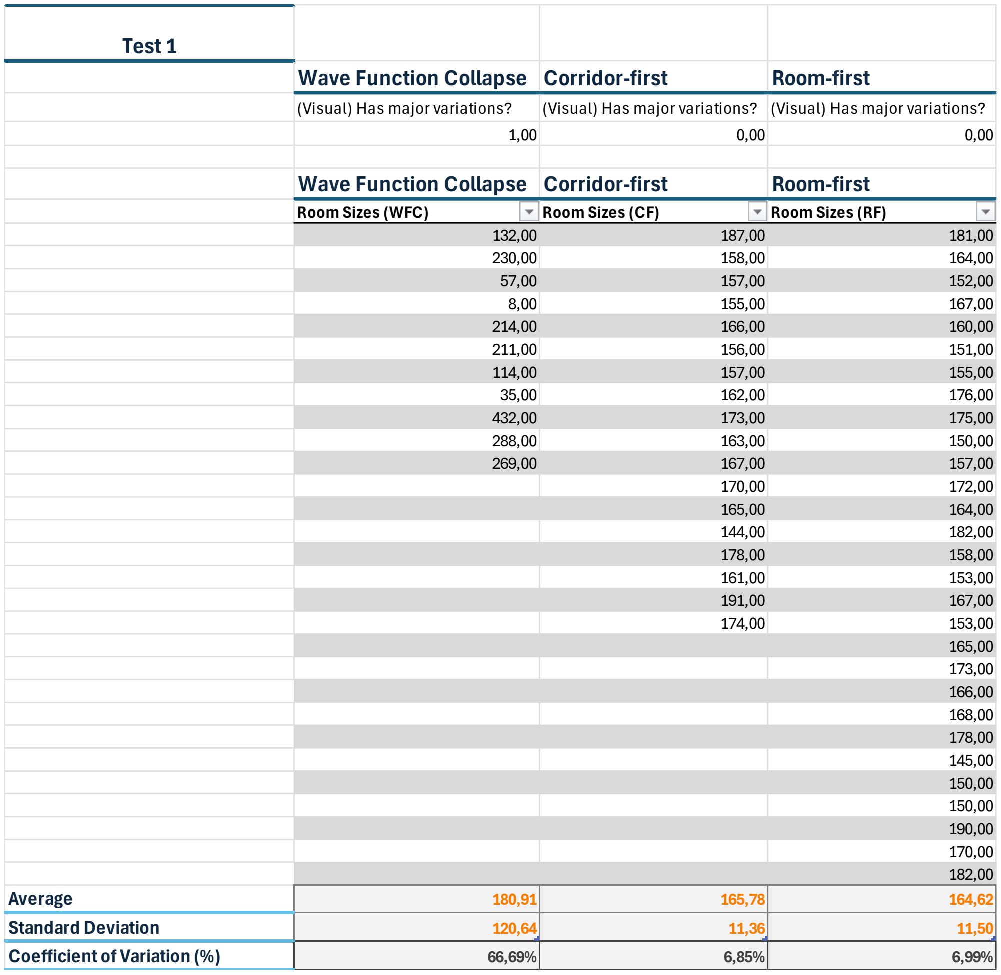


Figure 43: A bar graph visualization of the overlap weight of each method

Once again, Wave Function Collapse generated the most overlapping tiles on average during testing. The amount of overlapping tiles created by Corridor-First generation was minimal and did not have major impact on the layout of the dungeon. For the most part, the only overlapping tiles were usually at the very ends of the rooms produced. Given that the Room-First method never produced overlapping rooms during testing, the overall weight of overlap came out to zero. Once again, Room-First seemed to produce the best results in following user-defined constraints to generate a specific layout. However, the adverse results of Corridor-First were negligible and both methods seem perfectly valid for producing a dungeon given a specific layout with minimal overlapping rooms.

To continue testing if these methods can produce a dungeon of a specific layout, I also conducted many tests regarding the size of the rooms. For this layout, the rooms needs to be of a similar uniform size albeit small amounts of variation is allowed.

For this test, I produced five different iterations of generating the dungeons using all three methods. I then recorded the total amount of tiles in each room and produced the average size and the amount of variation in room size both accounting for the average size and not.



Test 1			
	Wave Function Collapse	**Corridor-first**	**Room-first**
	(Visual) Has major variations?	(Visual) Has major variations?	(Visual) Has major variations?
	1,00	0,00	0,00
	Wave Function Collapse	**Corridor-first**	**Room-first**
	Room Sizes (WFC)	**Room Sizes (CF)**	**Room Sizes (RF)**
	132,00	187,00	181,00
	230,00	158,00	164,00
	57,00	157,00	152,00
	8,00	155,00	167,00
	214,00	166,00	160,00
	211,00	156,00	151,00
	114,00	157,00	155,00
	35,00	162,00	176,00
	432,00	173,00	175,00
	288,00	163,00	150,00
	269,00	167,00	157,00
		170,00	172,00
		165,00	164,00
		144,00	182,00
		178,00	158,00
		161,00	153,00
		191,00	167,00
		174,00	153,00
			165,00
			173,00
			166,00
			168,00
			178,00
			145,00
			150,00
			150,00
			190,00
			170,00
			182,00
Average	180,91	165,78	164,62
Standard Deviation	120,64	11,36	11,50
Coefficient of Variation (%)	66,69%	6,85%	6,99%

Figure 44: Test 1 for variation in room sizes for each method, the room size being represented by the total floor tiles in a room

The first test, for example, produced results slightly in favor of Corridor-First. The amount of variation in room size for Wave Function Collapse was always high even with repeated attempts to change the sample used to train it. In the samples provided to WFC, I even conducted tests to see if a sample with all rooms being exactly the same size and only rectangular would produce a result in which the size of the rooms kept the same uniform size. Despite my best attempts to create a result without major variations, in every test there were always big differences in the sizes of the rooms. Some rooms produced were very small while others were exceptionally large. I also visually inspected each result to see if there were obvious large variations in room size. If a result had major variation, I set a 1 for true and a 0 for false.

To give a visual representation of the variation in room sizes, I also created the following box plot that shows the general sizes as well as outliers.

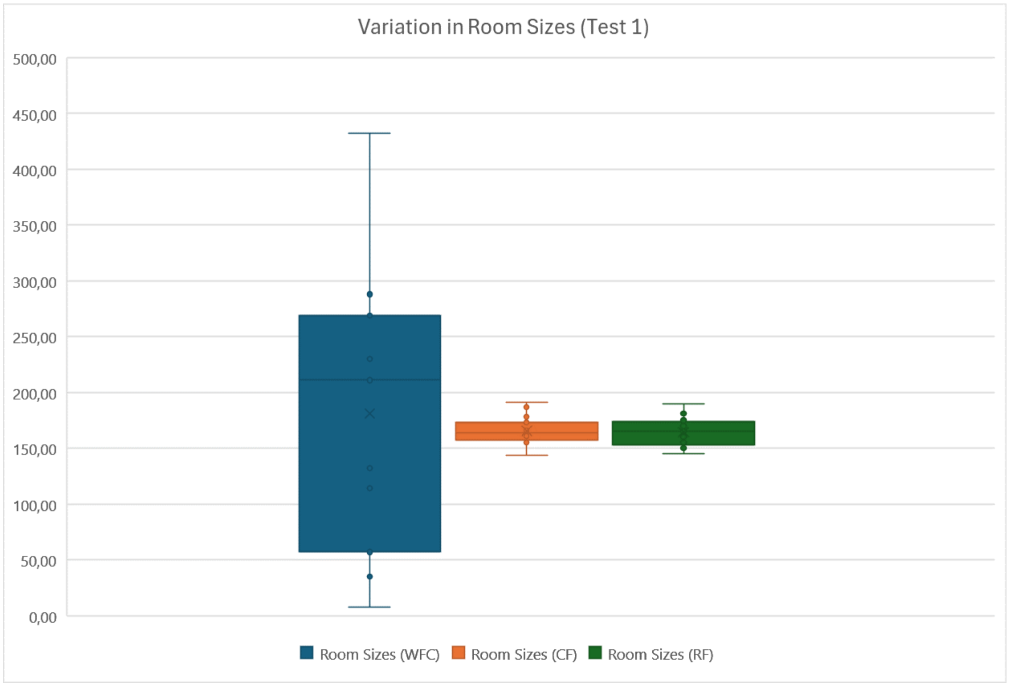


Figure 45: A box plot visualization of the variations in room sizes for each method

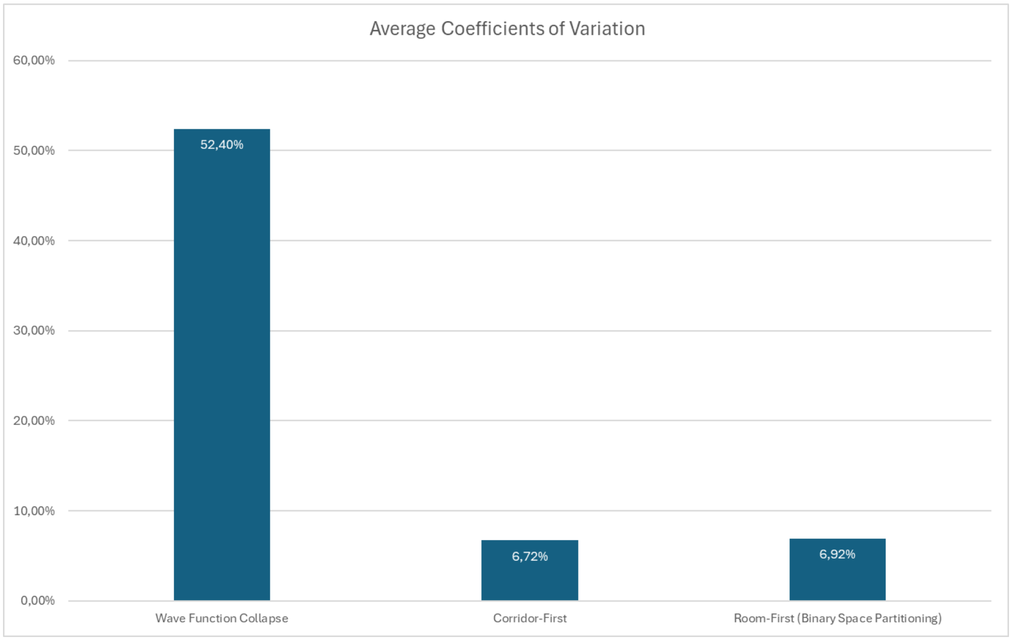


Figure 46: The average coefficients of variation for each method

The average coefficient of variation (the amount of variation in room size represented as a percentage towards the mean room size) for each method of all five tests produced the above results. On average, Wave Function Collapse produced the most major variation in room sizes while Corridor-First and Room-First methods produced very similar but negligible variation in room sizes. This makes Corridor-First and Room-First good contenders for this specifically desired layout.

Using this information we can answer the hypothesis comparing BSP and WFC methods in the field of producing a specific user-defined layout. As was estimated, BSP is the superior method for this desired result. However, due to expanding the results to also include Corridor-First generation, I can also conclude that both Corridor-First and Room-First generation are comparable in terms of results and are both valid choices for this purpose.

7.2 SUCCESSFUL GENERATION OF PLAYABLE DUNGEONS

The second hypothesis to test is regarding whether a given method of dungeon generation can produce a dungeon that is fully explorable and playable.

"The overall success rate of generating a playable dungeon (a dungeon in which the player can explore from start to end as well as access rooms) will be higher when using an implementation of Binary Space Partitioning compared to Wave Function Collapse."

This hypothesis was answered through a series of tests in which I generate a dungeon with a certain number of iterations (ex. 30 times running the generation for both algorithms). Each time the dungeon is generated, I tested two conditions.

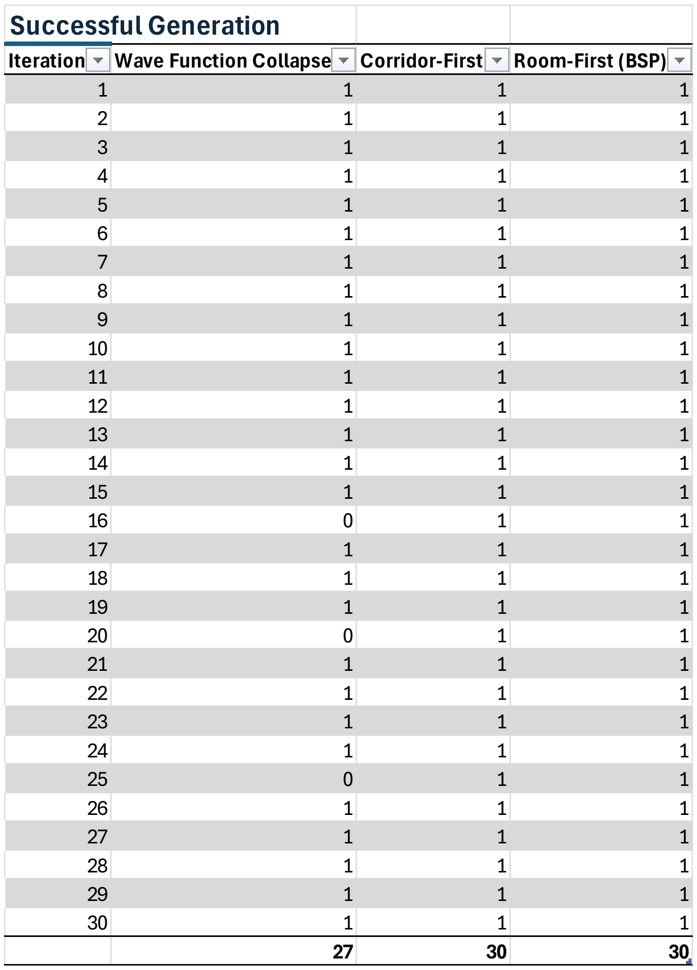
1. Did the dungeon generate successfully to completion or was the generation interrupted?

2. Can the dungeon be explored from start to end? Can the player explore all of the rooms?

I then recorded the overall success rate of generation as well as how often the player can fully explore the dungeon. In the case that the player cannot access all rooms, I hence calculated a percentile result of how many rooms were accessible.

While it is not a requirement to test this hypothesis, during this test I also compared the amount of time in seconds/milliseconds that it took for the generation to finish using each method. As with the previous hypothesis, I expanded the testing to also include Corridor-First in the evaluation.

To test if a dungeon successfully generate to completion, I generated the dungeon using each method 30 times and checked if the generation was finished without missing spaces or errors. The reason this test was conducted is because when using Wave Function Collapse there are sometimes scenarios in which the algorithm fails to generate, often resulting in the whole dungeon failing to be created.

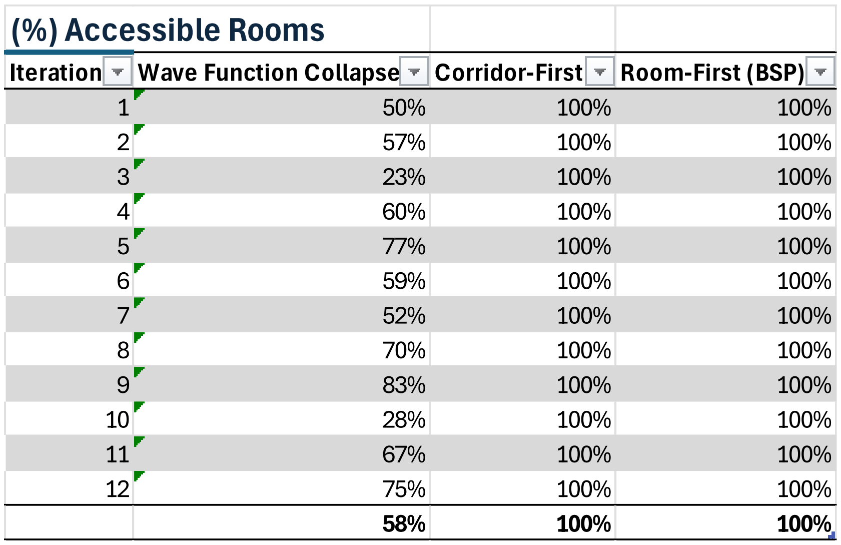


Successful Generation

Iteration	Wave Function Collapse	Corridor-First	Room-First (BSP)
1	1	1	1
2	1	1	1
3	1	1	1
4	1	1	1
5	1	1	1
6	1	1	1
7	1	1	1
8	1	1	1
9	1	1	1
10	1	1	1
11	1	1	1
12	1	1	1
13	1	1	1
14	1	1	1
15	1	1	1
16	0	1	1
17	1	1	1
18	1	1	1
19	1	1	1
20	0	1	1
21	1	1	1
22	1	1	1
23	1	1	1
24	1	1	1
25	0	1	1
26	1	1	1
27	1	1	1
28	1	1	1
29	1	1	1
30	1	1	1
	27	**30**	**30**

Figure 47: The rate in which each method successfully generated a complete dungeon

Every time I tested the generation of a method, I stored a 1 if the dungeon generated without issue and a 0 if the dungeon failed to generate either partially or completely. In 30 iterations of generation, Room-First and Corridor-First successfully generated the entire dungeon while there were three iterations in which WFC failed to generate. In all three cases, the dungeon failed to generate completely due to a tile not successfully being placed. In a custom implementation, there are surely ways to circumvent this such as setting a default tile to be placed when WFC fails to correctly evaluate which tile to place. This is certainly a topic to be further explored in future research.



(%) Accessible Rooms

Iteration	Wave Function Collapse	Corridor-First	Room-First (BSP)
1	50%	100%	100%
2	57%	100%	100%
3	23%	100%	100%
4	60%	100%	100%
5	77%	100%	100%
6	59%	100%	100%
7	52%	100%	100%
8	70%	100%	100%
9	83%	100%	100%
10	28%	100%	100%
11	67%	100%	100%
12	75%	100%	100%
	58%	**100%**	**100%**

Figure 48: The percent of accessible rooms for each generation of a dungeon

As for the percent of rooms that were accessible, therefore having a way of entering and exiting the room, both Room-First and Corridor-First methods demonstrated reliable, consistent results. This testing was conducted in twelve iterations and showed a 58% average room accessibility rate using WFC.

These results already provide an answer to the aforementioned hypothesis, proving Corridor-First and Room-First methods superior in the implementations produced by this research regarding both successful generation and room accessibility.

As well, to expand on this testing I generated dungeons 50 times using each method to test the amount of time in milliseconds needed to produce results.

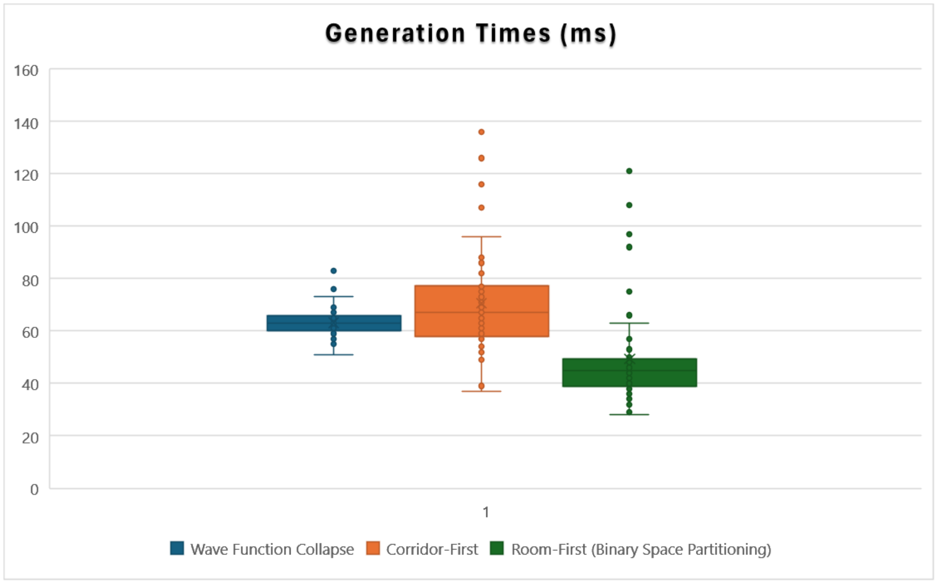
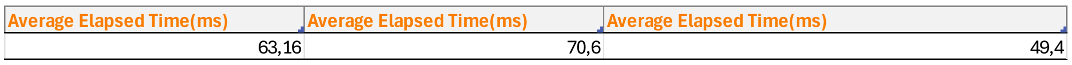


Figure 49: A box plot spread of each method's time elapsed in generating a dungeon



Average Elapsed Time(ms)	Average Elapsed Time(ms)	Average Elapsed Time(ms)
63,16	70,6	49,4

Figure 50: The average elapsed time by each method to generate a dungeon

In my experimenting, I found that Room-First produced results in the smallest amount of time compared to Wave Function Collapse and Corridor-First generation. The results between Wave Function Collapse were comparable, with Corridor-First and Room-First having a few outliers in which the generation time was more than average. To this end, Room-First proved triumphant.

7.3 SPACE UTILIZATION AND FREQUENCY OF DEAD ENDS

The final hypothesis to be evaluated proposed that Corridor-First generation produces results that more effectively utilize the space of the dungeon and have better flow to the layout.

“When generating the dungeon layout using Binary Space Partitioning, corridor-first generation proves to be a more efficient and versatile approach compared to room-first generation. It utilizes space more efficiently and offers greater control in level design by structuring the dungeon's flow.”

The first method involved with testing this hypothesis consisted of recording the percent of potential tiles in the bounds of the dungeons that were used in the final generation out of 30 tests. Essentially, we take the total possible amount of tiles that can fit in the space of the dungeon and record how many are actually placed. These tests are conducted using both Corridor-First and Room-First methods with the same values, dimensions, and layout preset.

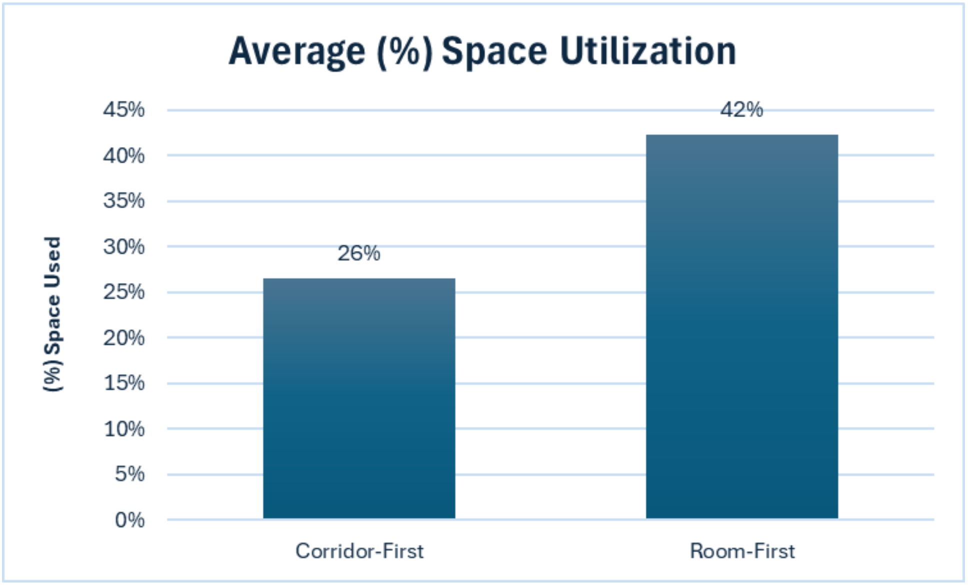


Figure 51: Average space utilization of Corridor-First and Room-First methods

Out of the 30 tests conducted, Room-First generation utilized space more efficiently, leaving less empty tiles in the bounds of the dungeon compared to Corridor-First. Depending on the implementation of your game, this is useful for lowering resource consumption and increasing performance.

Another aspect of this hypothesis that needs to be evaluated is the control a method has over the dungeon’s flow. To test this, a comparison has been made on the average time needed to traverse from room to room as well as the frequency at which isolated rooms or dead-ends are generated. Both methods once again use the same parameters for generation, making use of preset layout values.

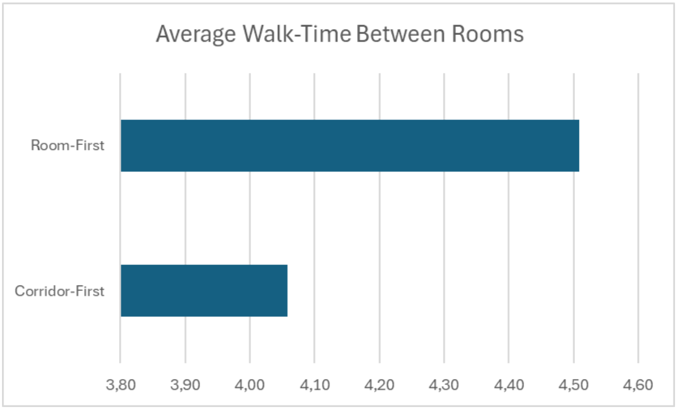


Figure 52: Average walk time between rooms represented by a bar graph

By recording the length of the corridors in between rooms, we can then take the player's maximum speed and make an estimated calculation of how long the player would need to travel from room to room using both methods of generation. Regarding traversal time, Corridor-First outperformed Room-First by providing results with less travel time between rooms. This makes travel less tedious and gameplay more fast-paced. In the results displayed by both methods during testing, the length of the corridors provides a small respite for the player while not making travel a boring chore. The amount of time difference between the two methods is in essence negligible, not creating a stark difference to note.

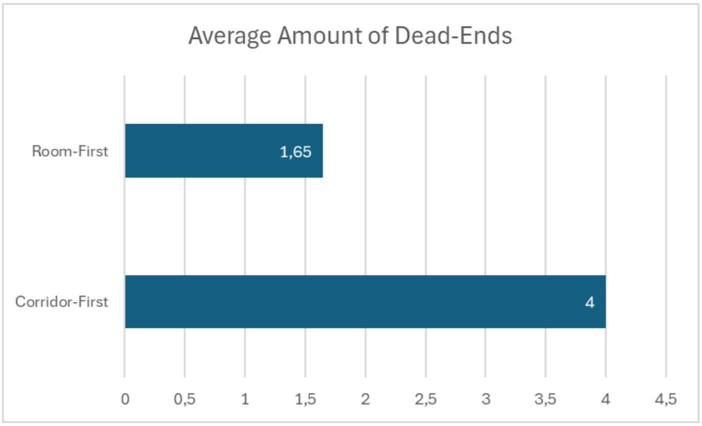


Figure 53: The average amount of dead-ends present in each method

As for how often either method produces rooms that do not connect to a further room, forcing the player to backtrack the same way they entered, Room-First produced better results. The average amount of dead-end rooms is notably less compared to Corridor-First.

In summary, this hypothesis was proven to be false. Room-First outperformed Corridor-First in both space utilization and dead-ends frequency, while only underperforming in traversal time by an insignificant amount. These results were not drastically stark in difference and both options are certainly still strong options for dungeon generation.

DISCUSSION

Out of all of the aforementioned methods of dungeon generation, the results indicate that Room-First and Corridor-First are both very viable methods for generating procedural dungeons in a modern game engine. Room-first excelled in utilizing space efficiently, having less dead-ends, and slightly less generation time. It also proved slightly better at producing specific layouts compared to Corridor-First. Even given these differences, both methods have a distinct effect on the flow of the dungeon. Room-First generation often produces more organized and blocky layouts, making the dungeon more traversable from multiple directions.

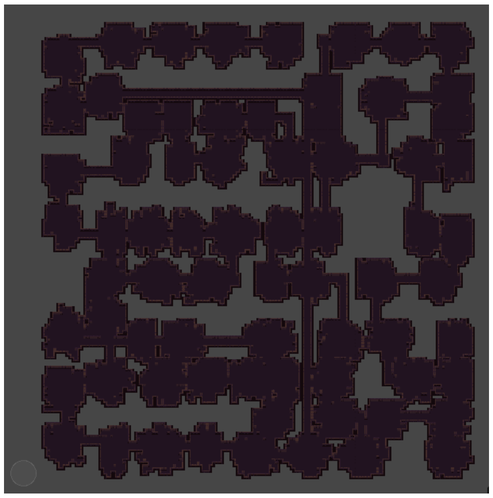


Figure 54: Room-First generation providing a more organized, grid-like appearance (left)

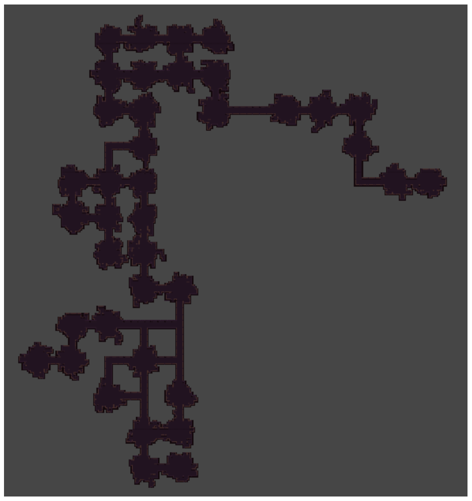


Figure 55: The more linear result of Corridor-First generation (right)

However, Corridor-First often produces more linear progression in the layouts of the dungeons. While there are often multiple directions to travel in, the flow of the dungeon can have a very distinguishable start and end point. The generation of the rooms follows a path more along that of a long line while Room-First is split into sections and filled.

Some methods are not easily viable for this purpose such as Cellular Automata given the very organic appearance of its generation that is difficult to split into distinct rooms. Some methods were also not chosen due to time constraints such as the TinyKeep method. Wave Function Collapse, however, was an experimental avenue of discovery, given that the method was not originally designed for dungeon generation and I was not very familiar with the concept before this research. After much experimentation and attempts to get the algorithm to work for this designed purpose, I have to say that it is not very suitable for this purpose without adding a lot of extra code and constraints. Perhaps it is also due to the lack of time that I was unable to produce reliable results using WFC. There is a chance there were more kinds of samples or methods I could have employed to get better results.

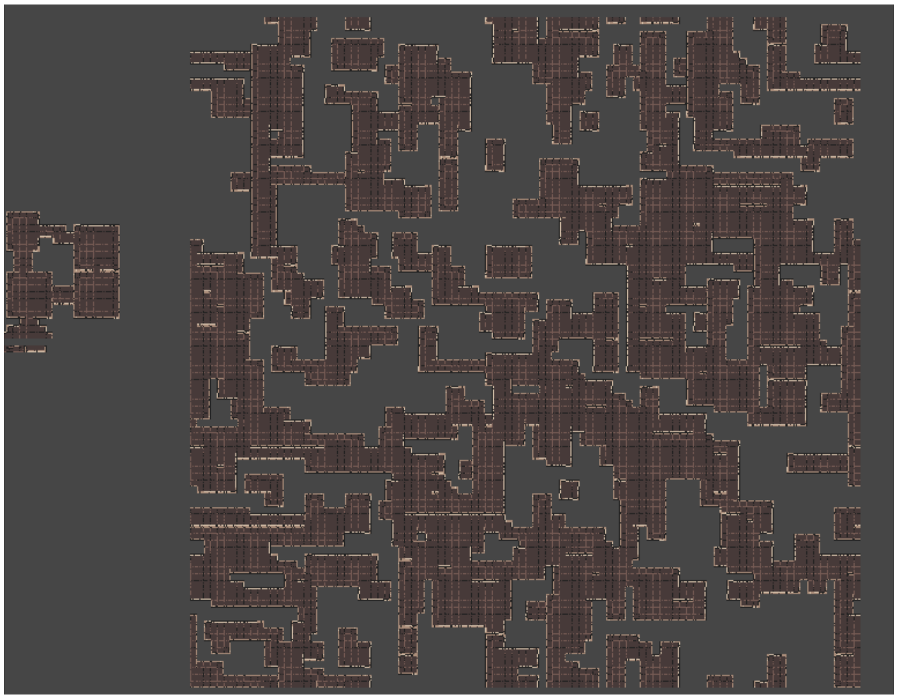


Figure 56: A failed dungeon layout attempt using WFC

However, in the large amount of time and countless samples training this algorithm, I was never able to produce a fully functional and completely playable result. There were always at least a few small isolated rooms where the player cannot access or there are outright blockages in the pathways. The result of the generation is very dependent on the sample provided to train the algorithm, as well as the rules applied to the possibilities of what tiles can be neighboring each other. While WFC can certainly be interesting in use of different applications, I do not find it a good solution to generating this type of environment. I can, however, definitely see its potential in generating terrain, biomes, images, and more.

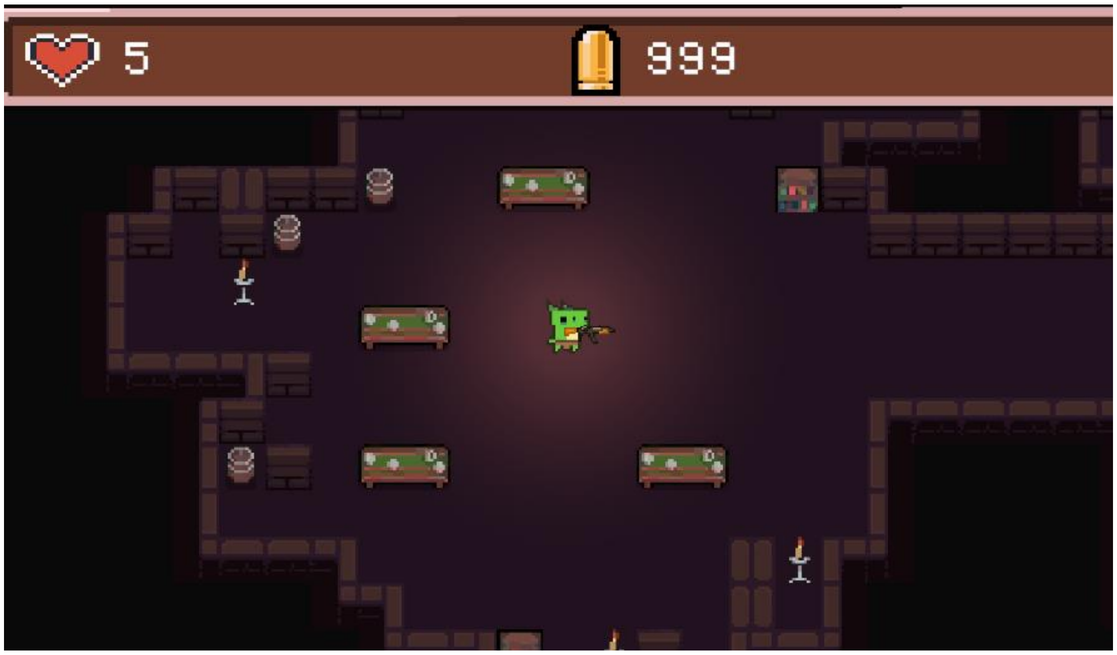


Figure 57: A finished result as seen from the game view

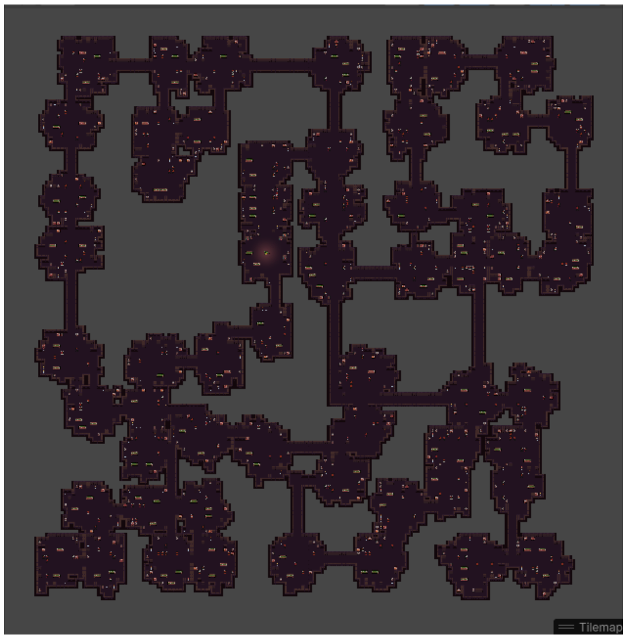
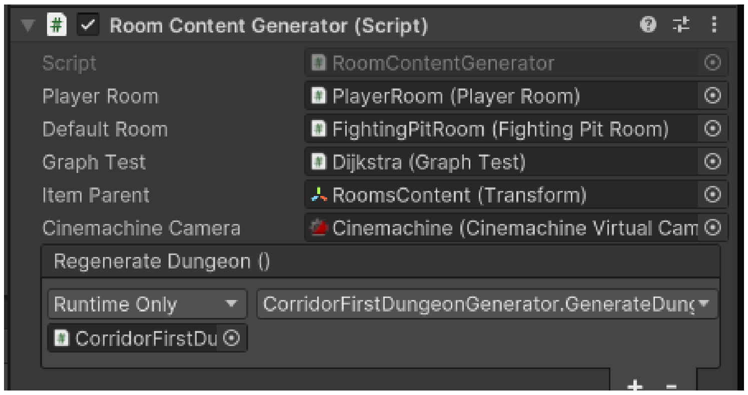


Figure 58: A finished result also works with Room-First generation

I am however, pleased with the final result after implementing both Room-First and Corridor-First methods as well as expanding the project from the dungeon generator created to include procedural item placement and 2D shooter gameplay. The gameplay was added by following and adapting a long string of video tutorials in a course that goes over many fundamental concepts of working in 2D in Unity as well as delving into user feedback, shaders, and more (Sunny Valley Studio, n.d.). I chose the Corridor-First method to be used in the demo, however the procedural item and enemy placement works in both Room-First and Corridor-First implementations. The methods for placing items procedurally and setting rooms to use certain presets (such as a spawn room or enemy room) to be scattered around the dungeon have plenty of potential for expansion and seem to work reliably.



Room Content Generator (Script)

Script	RoomContentGenerator
Player Room	PlayerRoom (Player Room)
Default Room	FightingPitRoom (Fighting Pit Room)
Graph Test	Dijkstra (Graph Test)
Item Parent	RoomsContent (Transform)
Cinemachine Camera	Cinemachine (Cinemachine Virtual Cam

Regenerate Dungeon ()

Runtime Only | CorridorFirstDungeonGenerator.GenerateDung

CorridorFirstDu

Figure 59: The Room Content Generator, which takes in scripts defining the behaviour and content rooms to generate

The current implementation of the high-level room system, which effectively delegates what should be in a room as well as where they should be placed, is a bit hard-coded at the moment. For the future, I will want to make this more easily configurable.

There is a small issue with the procedural item placement, however, which results in items sometimes blocking certain parts of the rooms.

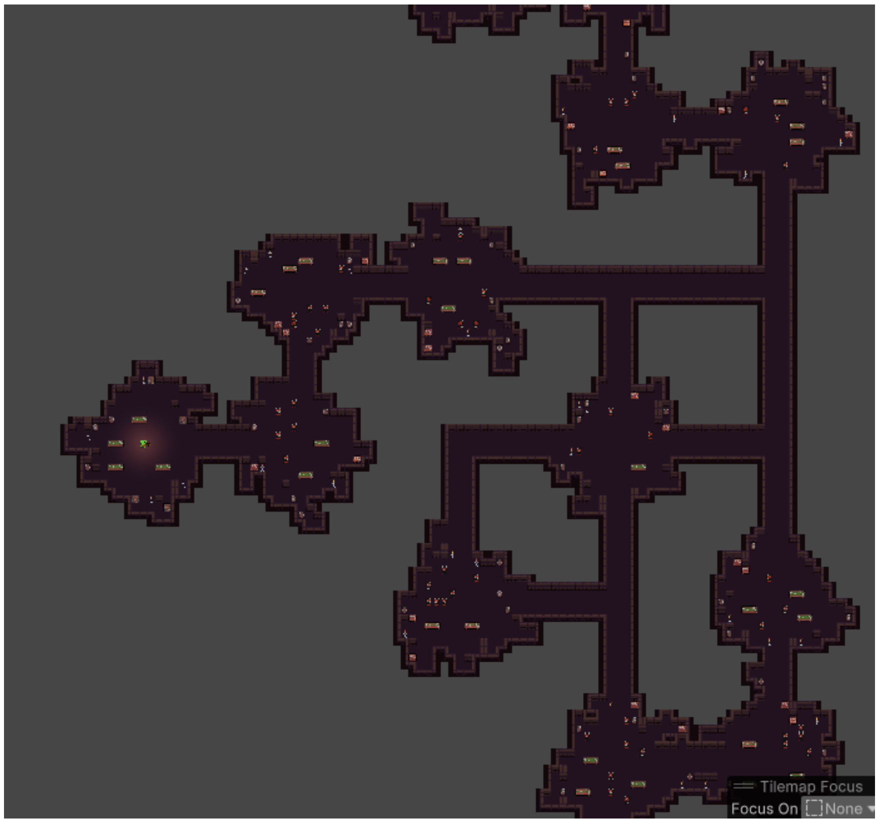


Figure 60: A finished result as scene from a bird's eye view using Corridor-First generation

However, by making the items scattered around the dungeon destructible, the player never needs to worry about getting stuck or not being able to access a part of a room. As well, by excluding the positions of the corridor floor tiles from the possible list of positions to spawn an item at, we can ensure items are never spawned blocking the pathways in between rooms. The exact placement of an item can be configured between being placed against a wall or away from walls. This distinguishes between the placement of a torch, for example, and a table. As well, there are plenty of things I would still try to improve with the gameplay aspects of the dungeon generator.

Dijkstra’s algorithm can be applied to the room content generator in the future to provide logic for how to place certain rooms. It allows me to calculate how far each floor tile is from a certain position. This is useful for possible expansion to spawn certain rooms based on distance from a point. For example, if I wanted to create a boss room that is placed as the room farthest from the player spawn room, I can make use of Dijkstra’s algorithm to see which room to place it in. The project has been expanded to include a basic implementation of this algorithm.

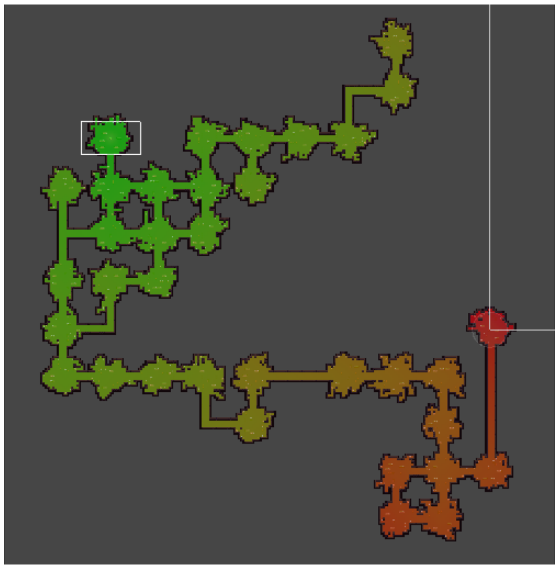


Figure 61: A visualization of Dijkstra's algorithm in the project

In the above Figure 57, the distance from the player’s spawn room center is represented by a color. The closer a floor tile is from the spawn area’s center, the more green it is. The farther it is, the more red the color becomes.

By implementing these extra features to the existing dungeon generator, we can already prove that the generator is able to be successfully applied and expanded upon when creating a game project. This framework has even more room for improvement and additional features that can make it into a fully fledged game given enough time and resources.

CONCLUSION

Procedural content is a good way to reduce the exponentially growing work load involved in making levels while simultaneously creating a unique, replayable experience in games. As stated previously, Corridor-First and Room-First methods of generation proved to be solid methods of producing dungeon layouts with individual rooms connected by corridors. They are also capable of creating dungeons with many different layouts and are rather customizable.

Wave Function Collapse on the other hand, is not as efficient for this particular use where each room must be clearly defined and connected via corridors. It might be possible to eventually create a working implementation, however that would require further research involving many more methods of employing the algorithms in different ways. For most of the testing and evaluation, the algorithm was trained on a sample made of individual floor and wall tiles that were fed into the generation.

However, perhaps there are different results depending on what kind of sample we use? The results may be drastically different by making a small change such as replacing the individual tiles in the sample with full room prefabs. Or perhaps WFC could be used to create an image similar to a noise map that can be used to generate floor and walls in the given positions indicated by the colors of the pixels produced by the algorithm? I still have many questions about the usefulness and application of the Wave Function Collapse algorithm but in the limited time span of this research it was not found to be a good solution for creating a dungeon generator.

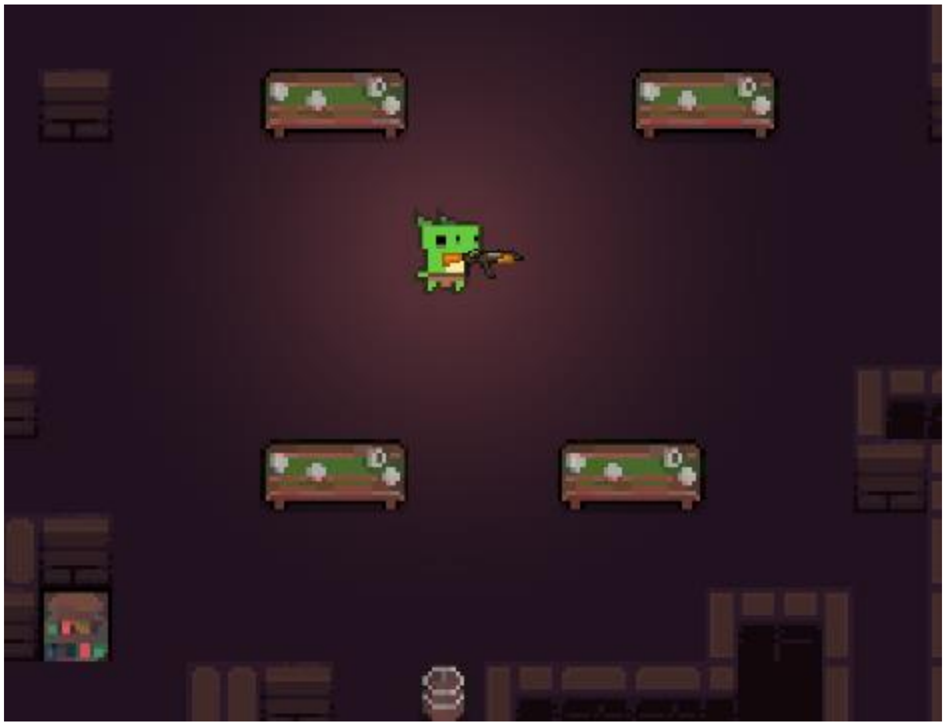


Figure 62: A close-up of the player character and a few of the items generated

The insight provided by this research can be pivotal for developers navigating time constraints and seeking efficient dungeon generation methods. In the realm of indie game development, where resources are often limited, procedural content generation emerges as a valuable tool. This study serves as a valuable resource for developers, offering a nuanced comparison of a couple favored methods, presenting their outcomes, and addressing challenges associated with alternative, experimental approaches. By shedding light on these different methods, I aim to provide essential guidance for developers, whether new to dungeon generation or in search of innovative methods for their projects.

FUTURE WORK

Regarding future research and exploration using this framework created, there is certainly more room for improvement and additional work. I would personally like to attempt to make a full rogue-like game using this generator. To do this, many other features would have to be implemented. For example, I would have to delve deeper into the placement of different types of rooms using Dijkstra's algorithm and other logic as needed. Boss rooms, treasure rooms, key rooms, and more can be added.

As well, most roguelikes feature unique items or treasures that modify the player character's abilities, add unique effects, or modify stats. On top of spawning items with unique effects and modifiers out of an item pool in a specialized treasure room, I would also like to explore the possibility of item synergies. This is a concept very familiar in the genre of roguelikes. In specific scenarios where the player collects items that synergize, the effects have a coded interaction or are even combined. For example, a moon ring and sun ring combining to create an eclipse ring that combines the effects of the previously collected items.

I would also love to expand the generator by not only spawning items but also various weapons the player can use. In the 2D shooter gameplay implemented, the weapon used by the player has various parameters for how many bullets it can shoot, the rate of fire, and more. By creating more weapons with different sprites and values in the parameters for the behavior of the guns, I can add more variation to the gameplay. This can be even further expanded by adding weapons with unique behaviors or types. An example could be a gun that whose bullets ricochet or a laser gun that incinerates enemies.

Furthermore, the generator only uses one variant of enemy. Another possible future work is adding more variations with unique attack patterns, behaviors, and even bosses. Boss battles are a subject I have not delved deep into, so it is a very fascinating possible new avenue of discovery for me.

Finally, I would also add more levels using different layouts. This is easily done using the implemented framework and would add much more to the variation of the dungeons. By swapping out the sprite sheet for the dungeon tiles with new ones every level, I can use the same generator and have an entirely new looking dungeon.

Now, independent from the currently implemented dungeon generator, I would certainly delve deeper into Wave Function Collapse and its possible uses. It is possible there is an implementation I overlooked during this research that could solve the challenges I faced when attempting to generate dungeons using it previously.

CRITICAL REFLECTION

Undertaking this research has taught me a lot of new skills and allowed me to work on a project that I am very invested in. Not only did I learn how various algorithms are implemented as well as how to create many of the different aspects of 2D gameplay, graphics, and mechanics, but I also learned a lot of personal lessons. From time management to skills in academic research, I have had to endure a lot of challenges to learn the lessons that I have.

I discovered more how to manage my time and plan a long-term project completely on my own, I learned to be more independent and self-sufficient, and more. I also learned the best ways to quickly mess up and fix your sleeping schedule several times over the course of this research. Alongside these skills, I also strengthened my communication skills with my fellow classmates. Asking feedback, discussing certain sections of the thesis, and more improved my connections, bonds, and experience. I am very happy that I have formed friendships with the people that I have.

I personally feel as if this research has taught me not just a lot about procedural generation but also about many fundamental skills in creating graphics, user feedback, shaders, and many other concepts in Unity. My familiarity with the engine and the processes involved in many aspects of making a game on my own has been increased significantly by this project.

REFERENCES

A Adonaac. (2015, September 3). Procedural Dungeon Generation Algorithm [Blog]. *Game Developer*. https://www.gamedeveloper.com/programming/procedural-dungeon-generation-algorithm

Boris. (2020, April 13). Wave Function Collapse Explained. *BorisTheBrave.Com*. https://www.boristhebrave.com/2020/04/13/wave-function-collapse-explained/

Brian Bucklew (Director). (2019, October 23). *Dungeon Generation via Wave Function Collapse* [Youtube Video]. https://www.youtube.com/watch?v=fnFj3dOKcIQ

Cellular Automata rules lexicon—Life. (2007, November 17). https://web.archive.org/web/20071117041941/http://www.mirwoj.opus.chelm.pl/ca/rullex_life.html

Cellular Automata—Procedural Content Generation Wiki. (n.d.). Retrieved January 12, 2024, from http://pcg.wikidot.com/pcg-algorithm:cellular-automata

Cellular automaton. (2024). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Cellular_automaton&oldid=1193387331

Delaunay triangulation. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Delaunay_triangulation&oldid=1179277287

Gumin, M. (2022). *WaveFunctionCollapse* (v1.00) [C#]. https://github.com/mxgmn/WaveFunctionCollapse

selfsame. (2020, June 4). Unity-wave-function-collapse. *Itch.Io*. https://selfsame.itch.io/unitywfc

Sunny Valley Studio. (n.d.). Make a juicy 2d Shooter prototype in Unity 2020 [Academic Courses Webshop]. *Sunny Valley Studio*. Retrieved October 18, 2023, from https://courses.sunnyvalleystudio.com/p/make-a-juicy-2d-shooter-prototype-in-unity-2020

Sunny Valley Studio (Director). (2020, December 18). *Unity Procedural Dungeon Generation 2D* [Youtube Video]. https://www.youtube.com/watch?v=-QOCX6SVFsk

Sweden Game Arena (Director). (2021, December 8). *SGC21- Oskar Stålberg—Beyond Townscapers* [Youtube Video]. https://www.youtube.com/watch?v=Uxeo9c-PX-w

Timothy Hely. (2023, March 21). How to Use BSP Trees to Generate Game Maps [Blog]. *EnvatoTuts+*. https://gamedevelopment.tutsplus.com/how-to-use-bsp-trees-to-generate-game-maps--gamedev-12268t

ACKNOWLEDGEMENTS

I would like to thank my dad for supporting me with the occasional encouraging words, listening to me rant about my research and all of the information I learned, and for taking me out and cheering me up when the work became too much and I felt burned out and down on my luck.

Thank you to my mom, who passed away far too soon in life, for always loving me and watching over me all of these years.

To my sister, who lives still far away in my home country at the time of writing, I thank her as well for being in my life for many years. I am extremely proud of you and what you are doing, sis.

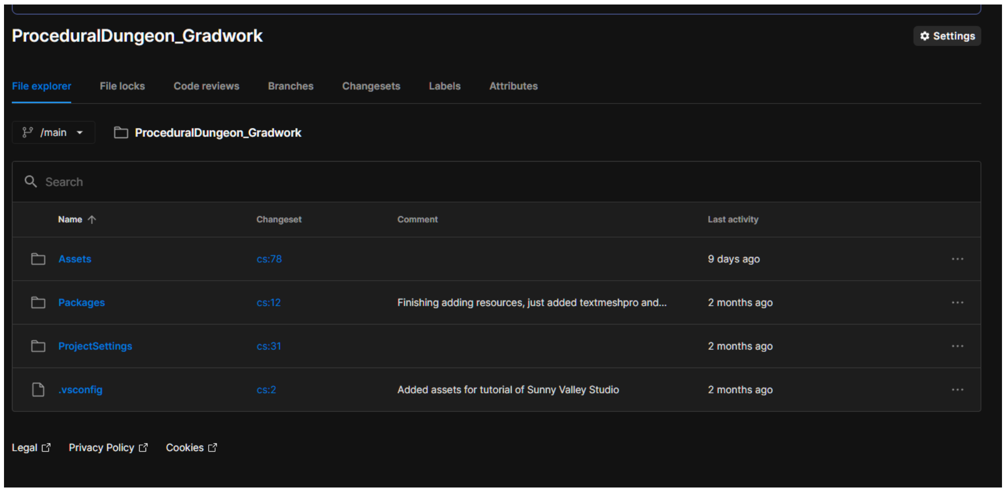
I would also like to extend an additional thanks to my supervisors, teachers, and coaches.

Thank you for your guidance and support on this journey and I wish every one of you the best.

APPENDICES

1. The full spreadsheet of data accumulated for the experiments
2. A screenshot of the online Unity version control repository

https://docs.google.com/spreadsheets/d/1Ieve2wfHGcojIt1TT7xigiIBG2mcZVNdoNV_m4AEFfQ/edit?usp=sharing



Name	Changeset	Comment	Last activity
Assets	cs:78		9 days ago
Packages	cs:12	Finishing adding resources, just added textmeshpro and...	2 months ago
ProjectSettings	cs:31		2 months ago
.vsconfig	cs:2	Added assets for tutorial of Sunny Valley Studio	2 months ago

Figure 63: The online Unity version control repository, which to my knowledge cannot be shared via link