Senior Comprehensive Project

Game Design and Infinite Procedural Level Generation: Creating a 2-D Roguelike with Lock & Key Puzzle Elements in Unity

Sean Emery
semery@oxy.edu

Occidental College, Los Angeles, CA
Bachelors of Arts in Computer Science and Economics

Creating a 2-D Roguelike Game with Infinite Procedural Level and Puzzle Generation in Unity

December 7, 2020

1 Abstract

The final product of this comprehensive project can be best described as a Minimum Viable Product (MVP) of a roguelike game developed in the Unity Engine with C#. The game contains a procedural generation algorithm with lock and key puzzle elements to obfuscate the player's path to the final boss. Additionally, all the art and animations in this game are original assets that were created using Aseprite (a pixel/animation editor), leaving the sound effects and music as the only components that came from an open-source library. This paper will describe and discuss the various algorithms used to create the dungeons in this game alongside the challenges and difficulties associated with implementing them. Some algorithms of particular note that were implemented include Kruskal's greedy minimum spanning tree algorithm and a modified breadth-first search algorithm.

2 Introduction

Most 2-D dungeon-crawler games are considered to be roguelikes: a genre of games which are notorious for being fun due to its extreme difficulty. These games tend to be difficult because of its typical inclusion of the following two elements— procedural level generation and permadeath. The combination of these two gameplay mechanics means that whenever the player dies, they would have to start the game over from scratch in an entirely new and unique dungeon layout. Although the first and most well-known roguelikes fit the traditional *Dungeons & Dragon* RPG themes, games in this genre are not necessarily bound to it. Many of the most popular modern roguelike games have abandoned the fantasy dungeon motif in favor of their own unique ideas. For instance, the game *FTL: Faster Than Light* has the player control their own spaceship and crew "through a randomly generated galaxy filled with glory and bitter defeat." Another roguelike, *The Binding of Isaac*, its title, plot, and gameplay inspired by the biblical story of the same name.

What makes the roguelike genre of games particularly fascinating is how technically challenging it is to create the dungeons (or levels in more general terms) for the game. This is because in

order for a game to qualify as a roguelike, the dungeons must be 100% procedurally generated to satisfy the design philosophy of the genre. This means that every room, item, enemy, boss, lock and key placement in a level must be completed by an algorithm. These procedural dungeon generation methods can incorporate famous computer science and mathematical concepts such as: random walk algorithms, spanning tree algorithms, maze algorithms, cellular automata, etc. This specific topic alone will be covered more in-depth in the literature review section. On the other hand, another challenging aspect of this project is the game design aspect. Although developing a custom procedural level generation algorithm is the main goal of this project, all of the essential components that make a game will also have to be implemented. This includes the physics, general game mechanics (such as a combat/health-point system, enemy AI, etc), art assets, HUD (aka Head-up Display) elements (such as health status indicators, minimaps, etc), sound effects, and music. It should also be noted that the game design element will also have to be fully integrated with the custom procedural level generation algorithm to make a cohesive final product.

3 Literature Review

The term roguelike gets its namesake from the game *Rogue: Exploring the Dungeons of Doom* (aka Rogue). In Rogue, the player is tasked with descending down to the dungeon's lowest level to retrieve the Amulet of Yendor, battling monsters and gathering loot along the way. As McHugh indicates:

> The main hook for the game is that each time you enter the dungeon, its layout, monsters, and loot are all procedurally generated. This means that every time you start the game you'd face a new challenge, one you hadn't seen before. The game also incorporates permadeath; each time you die your character is gone forever and you must start over afresh with an entirely new dungeon layout [10].

Because of Rogue's influences, these two design core elements— procedurally generated levels and permadeath— can be found in virtually every modern roguelike released today. This literature review will examine the ramifications that these two design elements have on roguelike game design as it differs drastically from traditional game design conventions. Additionally, a section of this literature review will be dedicated to exploring the various ways modern roguelikes have innovated in the field of procedural level generation to create more fun and interesting dungeons. Lastly, there will also be a brief examination of integrating solvable lock and key puzzles into a procedural level.

3.1 Game Design Considerations - Roguelikes

The core design elements of roguelikes place an emphasis on repeated and iterative attempts at success: permadeath raises the stakes for each playthrough and procedural level generation ensures a new experience and limits the carryover of any advantages from previous games [9]. An advantage of roguelikes having this emphasis on an iterative approach to gameplay is that this design almost exclusively rewards player skill rather than rote memorization of level layouts. As Brown states:

> Because the game never changes from run to run, the overall difficulty level of the game- across multiple attempts, that is - is completely flat. That means the only way to smash through that barrier and win is to improve your own skill: by learning the ropes, practicing the controls, and becoming more familiar [2].

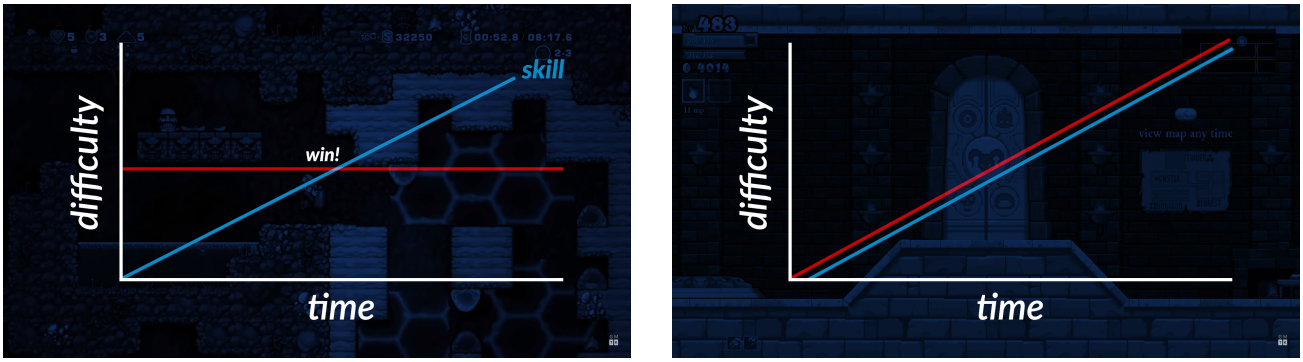
A consequence of this approach is that new players are unlikely to make it to the last level of a dungeon in a roguelike on their first attempt, let alone their second or even tenth attempt. Generally, they are expected to die along the way, but they should also learn from their mistakes when they inevitably restart their dungeon crawl from scratch. For instance, in the cave-exploring roguelike *Spelunky*, an unsuspecting novice player may immediately die to a boulder trap when they steal a Golden Idol treasure from the mines (the first level of the game). Eventually, the player should be able to learn to escape the trap by ascending to higher ground with either the aid of a ladder or ropes [12]. Similar lessons can be learned from dying to difficult enemies and bosses before players are able to discover a method to kill them or evade their attacks. These methods can then be slowly practiced and mastered over multiple playthroughs.

(a) The Idol looks deceptively easy to obtain

(b) Stealing it activates a deadly boulder trap

Figure 1: Screenshots from the game Spelunky by Derek Yu [12].

It should be noted, however, that this approach differs drastically from how most other games approach setting their difficulty level. Normally, games tend to start off easy when the player is unskilled but they get progressively harder as the player becomes more skilled at playing the game. Graphically, the progression of difficulty and player skill between roguelikes and other games can be visualized in the following the way (Legend: red line = difficulty, blue line = skill):



(a) Progression in roguelikes

(b) Progression in other genres

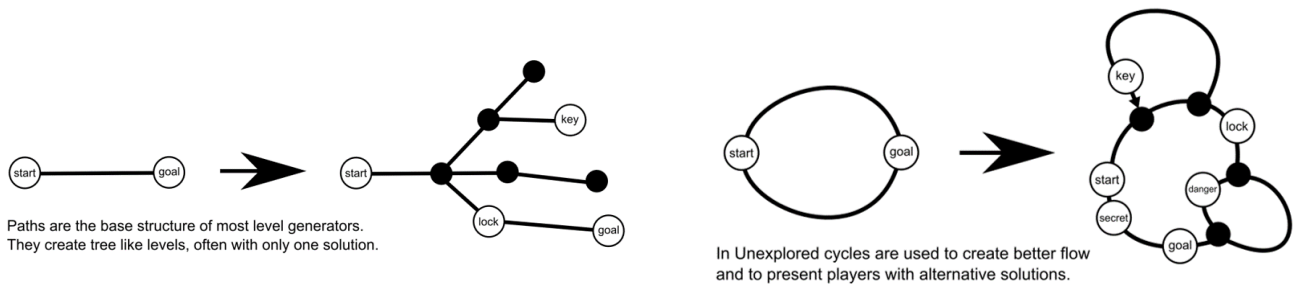
Figure 2: Adapted from Mark Brown's Roguelikes, Persistency, and Progression [2].

What Figure 2 highlights is one of the genre's biggest pitfalls: because the difficulty level starts off so high and remains at that constant level, low skilled players may never never finish the game no matter how many times they try. However, although roguelikes tend to never reduce the difficulty of their games to accommodate novice players, most do try to implement methods to give players a helping hand without changing the balance of the game [2]. For example, when your squad of soldiers inevitably die in the turn-based tactical roguelike *Into the Breach*, you can keep one levelled up soldier who died so that they can then be used during your next playthrough of the game (while the rest of the squad are permanently gone). But because these characters do not stick around forever, it is not making a persistent change to the game's difficulty. Instead, it just gives players that tiny bit of a leg-up and also an excuse to try playing 'just one more go' [2].

3.2 Procedural Level Generation - Roguelikes

One of the most common strategies that roguelikes employ to procedurally generate dungeons is to use some sort of tree-generating algorithm [6]. At the most basic level, the dungeon is grown from a single starting room and the algorithm attempts to keep adding more generated or pre-designed rooms to the level. Although this typically results in many branches and dead-ends (which forces players to constantly backtrack), many algorithms will try to circumvent this problem by randomly looking for places to randomly reconnect the branches to varying degrees of success [6].

The game *Unexplored*, however, takes a completely different approach in how it tackles the problems of branches and dead-ends from tree-generating algorithms. As Dormans highlights, "where good cycles and interesting cycles might randomly appear using the old [tree-generating] technique, in *Unexplored* they [are] a planned feature of the generator's output" [6]. Instead of simply creating a branch to a target goal, the algorithm in *Unexplored* creates a cycle consisting of two paths between the entrance and the goal [6]. Moreover, the algorithm will always try to augment one of the paths to introduce variety: possibly turning one of them into a hidden shortcut, or a dangerous trap infested route [6]. Figure 3 illustrates the differences between the standard tree procedural level generation method and *Unexplored*'s cyclical procedural level generation method:



(a) Trees (b) Cycles

Figure 3: Adapted from Joris Dormans' A Handcrafted Feel: 'Unexplored' Explores Cyclic Dungeon Generation [6].

On the other hand, a more common method involves using Binary Space Partitioning (aka

BSP) algorithm on a 2-D plane to generate rooms in a procedural dungeon. The BSP algorithm is a recursive method for subdividing a space into two convex shapes until the partitions satisfy one or more requirements (in any dimensions above the 3rd, hyperplanes would be used to partition the space into convex sets). Additionally, this process of subdividing can be represented as a binary tree data structure. Generally, this algorithm is used for 3-D rendering applications to break down complex, overlapping 3-D shapes into smaller, primitive 3-D shapes. Doing so allows the rendering pipeline to determine which order the primitive shapes should be rendered in when traversing the corresponding tree:

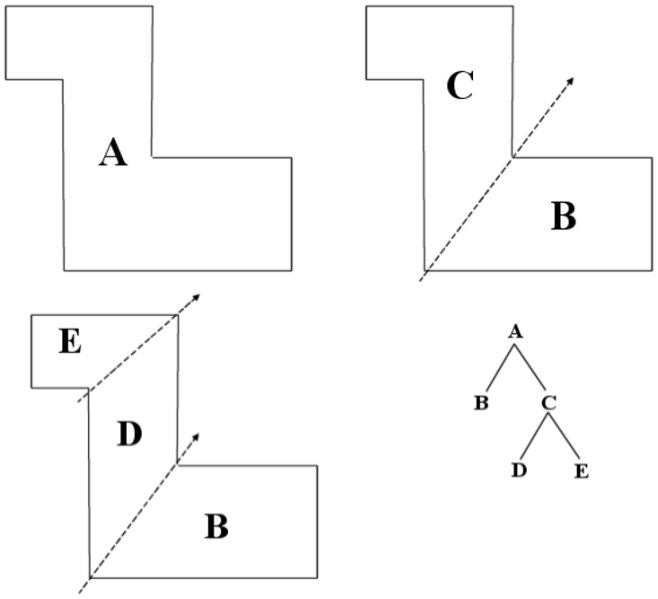
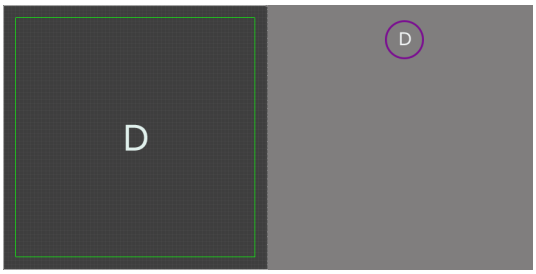


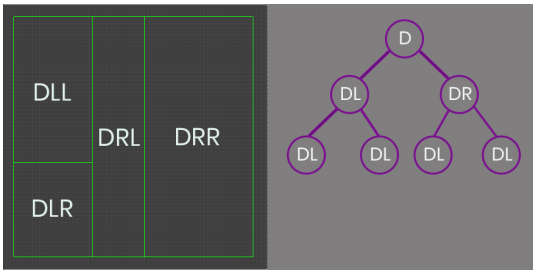
Figure 4: In a Back-to-front renderer, far away objects like walls are rendered first and are obscured by closer walls. Either B or E could be the furthest wall depending on how the tree is traversed. Adapted from Simon Howard's BSP and its use in 3-D Rendering [8].

However, typical BSP algorithms found in roguelikes tend to divide the world space (a 2-D grid with a predefined height and width) into random, small, rectangular partitions that can fit a room. A step-by-step process of this can be seen in Figure 5.



(a) Initialize root leaf that bounds the maximum size of the dungeon

(b) Randomly partition the leaf vertically or horizontally into two child leaves

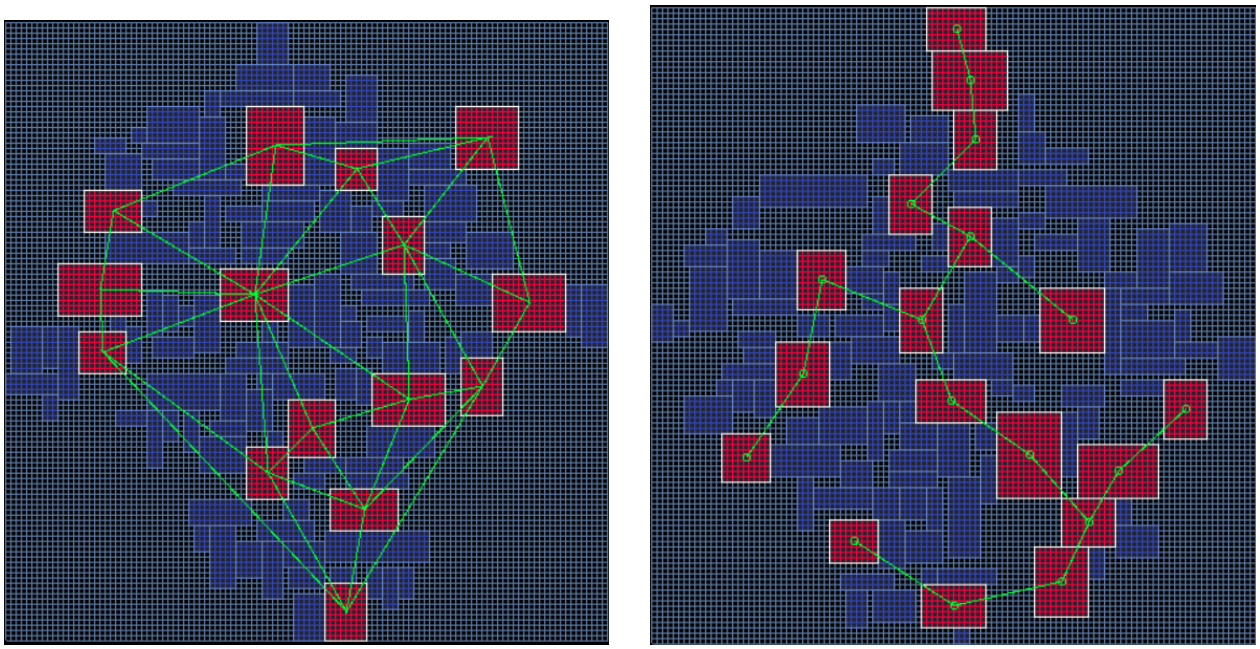


(c) Repeat random vertical/horizontal partition until minimum leaf size is reached

Figure 5: Adapted from Júlio Rodrigues' How to Procedurally Generate a Dungeon Using the BSP Method on Unity [11].

Another popular method used by roguelikes typically involves the implementation of either a minimum spanning tree algorithm or a Delaunay Triangulation algorithm. There are many different ways to go about incorporating these algorithms into a procedural level generation context, but generally they all involve connecting the disjointed pre-generated rooms together with either hallways or several smaller intermediate rooms.

For instance, the dungeon-crawling roguelike *TinyKeep* first generates a cluster of overlapping rooms of varying sizes. Afterwards, a separation steering algorithm is used to eliminate any overlapping room tiles while also keeping them tightly packed [5]. Next, *TinyKeep*'s algorithm then establishes a small subset of rooms that will be the 'main' rooms by using a size threshold criteria. Lastly, TinyKeep's procedural level generation algorithm would then construct two graphs: a graph of all of the main rooms' center points using Delaunay Triangulation and a minimum spanning tree based on those aforementioned rooms [5]. A Delaunay Triangulation picks a triangulation that maximizes the smallest angle between all vertices/rooms (and thus it tries to avoid thin triangles wherever possible) while a minimum spanning tree attempts to connect all the vertices/rooms with the shortest paths. An illustration of these graphs can be seen in Figure 6.



(a) Delaunay Triangulation paths between selected rooms

(b) Minimum Spanning Tree paths between selected rooms

Figure 6: Adapted from Adonaac's Procedural Dungeon Generation Algorithm [1].

The two graphs from Figure 6 represent two extremes: the Delaunay Triangulation graphs connects all the main rooms to each other without intersecting lines but offers no clear guidance on which path to take; the minimum spanning tree ensures that all rooms are connected but contains no loop. The developers behind *TinyKeep* found a compromise between the two solutions that involved reincorporating a small number of edges from the triangulated graph on top of the minimum spanning tree paths to add sufficient variety [5].

Lastly, some roguelikes use cellular automata to produce cave-like structures as a part of their procedural level generation algorithms. With its foundation based in Conway's Game of Life, the map is first randomly filled with cells that are either 'alive' or 'dead' (in this particular case for procedural level generation, the terminology would be 'wall' and 'not wall'). Next, an iterative process begins whereby new maps are created using the 4-5 rule: "a tile becomes a wall if it was a wall and 4 or more of its eight neighbors were walls, or if it was not a wall and 5 or more neighbors were" [3]. This process is repeated numerous times, and eventually the output might look like the following figure:

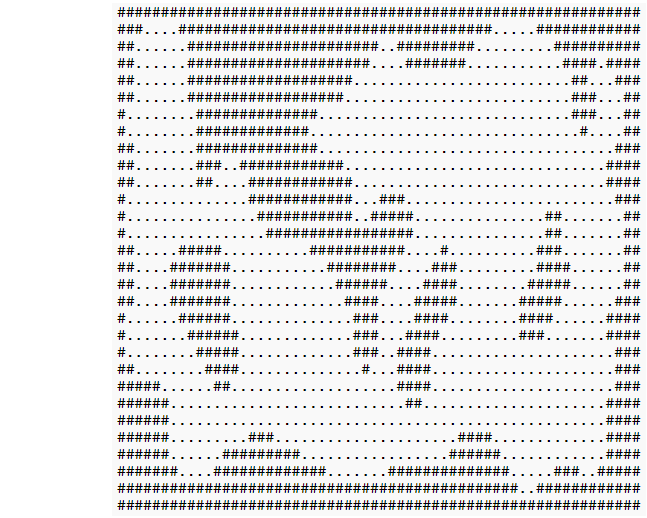


Figure 7: Adapted from RogueBasin's Cellular Automata Method for Generating Random Cave-Like Levels [3].

Despite its simplicity, a major disadvantage of this approach is that this algorithm is prone to generating isolated cave sections and/or maps with huge open spaces [3]. Neither of these things are ideal for dungeon-crawler type games, but luckily both of these issues can be resolved by making slight modifications to the original cellular automata rules [3].

3.3 Procedural Puzzle Generation - Zelda-esque Locks and Keys

The Legend of Zelda is a critically acclaimed video game franchise developed by Nintendo whose games have popularized various action-adventure gameplay tropes. More specifically, the very first game that gave the franchise its namesake, *The Legend of Zelda*, was a revolutionary 2-D medieval fantasy game released in 1986 that codified the Action-Adventure genre with its mix of puzzles, action, exploration, and battle/adventure gameplay [7]. Moreover, gameplay influences from *The Legend of Zelda* series can be found in many modern game titles and among various genres. However, the only gameplay remnant that will be discussed in this section is the series' iconic dungeons and puzzles (any future reference to Zelda-inspired gameplay ideas will be described as Zelda-esque).

One of the most iconic aspects of the 2-D Zelda games is its unique dungeon designs. Despite their linear layouts, these Zelda-esque dungeons implement item-based puzzles to create the illusion of non-linearity. Take for instance the game *Link's Awakening* from the Zelda franchise where the dungeons in this game can usually be broken down into discrete rooms of a fixed size, laid out in a grid. These rooms have walls separating them and some of those walls have doors, which are possibly locked [4]. Generally these rooms will contain the following items:

- Keys that unlock a door (and leave the door unlocked, but can only be used once)
- A single boss key that unlocks the door to the boss

- A special equipment that enables access to previously inaccessible rooms/areas (for instance, the Roc's Feather item allows you to jump over pits that were previously impassable).

Coxon describes an example of what a minimalist dungeon in Link's Awakening might look like [4]. While the actual dungeons in Link's Awakening would have multiple locked doors, multiple keys, and a miniboss or two; the essence of these dungeons could be reduced to the following layout:

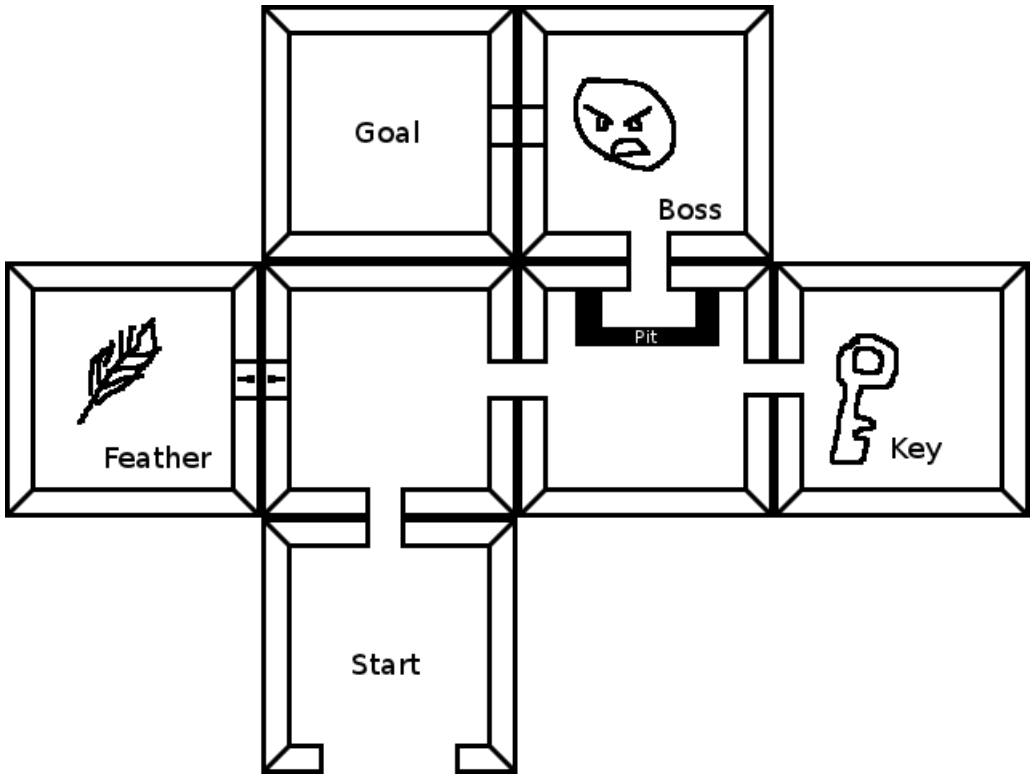
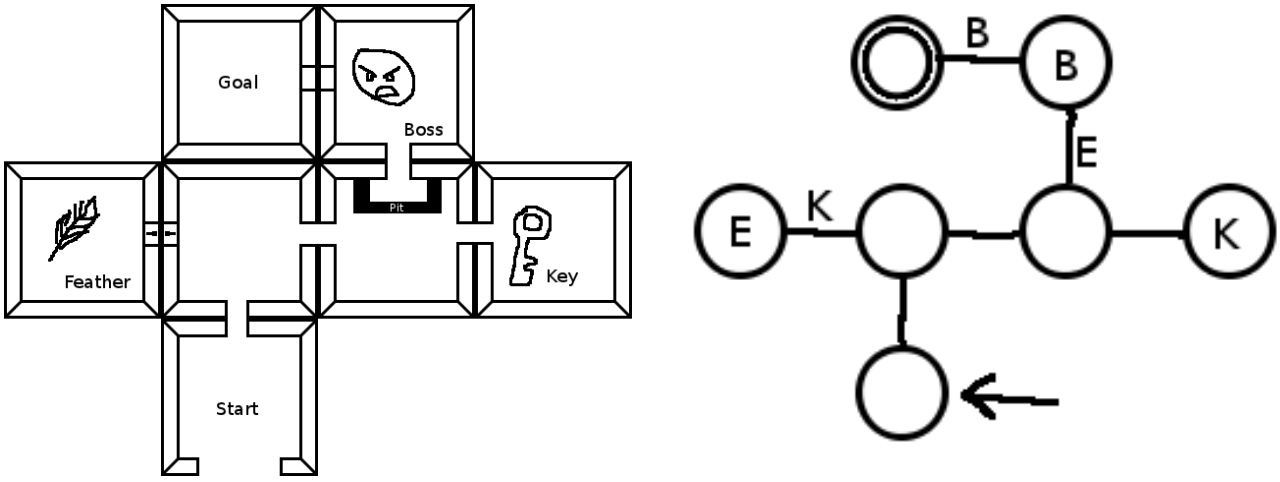


Figure 8: By reasoning backwards, the player discovers the right sequence of rooms to get to the goal from the start. The player cannot get to the goal without beating the boss; the player must obtain the feather to jump over the pit surrounding the entrance to the boss; the player cannot get the feather without first unlocking the door to it; the player cannot unlock the door without getting the key. Adapted from Coxon's Procedural Dungeon Generation - Part I [4]

.

All of these puzzle elements act as obstacles that the player cannot pass until they find the necessary items to overcome them. Specifically, this dungeon can be turned into an abstraction as demonstrated by the following figure [4]:



(a) Reduced dungeon map

(b) Graph abstraction of the reduced dungeon map

Figure 9: The items, keys, and boss fight can be treated as boolean variables: either the player has the key to unlock the door or they do not ("K"); either the player has the feather to traverse the pit or they do not ("E"); either the player has defeated the boss or they have not ("B"). Adapted from Coxon's Procedural Dungeon Generation - Part I [4].

This particular graph abstraction is known as an Extended Finite State Machine (aka EFSM). It extends the model of a regular Finite State Machine by augmenting some transitional edges with "if conditions." These conditional edges will have an initial starting value of false which will represent an impassable obstacle. These boolean variables will turn from false to true (thus making the edge traversable) whenever the player obtains an item that would make the corresponding obstacle passable [4].

4 Methods

4.1 Creating the Procedural Backend

The procedural generation algorithm behind this project can be described broadly as a combination of some of the various algorithms described in the literature review section of this paper. Ultimately, these algorithms work together to successfully produce procedural dungeons with lock and key puzzle elements that are solvable by the player. In order, the algorithms used are as follows:

1. Binary Space Partitioning (aka BSP) Algorithm
    - Used to segment a 2-D grid space to accommodate rooms of random sizes with adequate space for corridors between them.
2. Dwyer's Delaunay Triangulation Algorithm (Divide and Conquer)
    - Used to convert each room's center vertex into a Delaunay Triangulation graph/mesh which maximizes the minimum angles of all the triangles in the mesh.

3. Kruskal's Greedy Minimum Spanning Tree Algorithm (Disjoint Set, Union-Find variation)
    - Used to find a minimum spanning tree of an undirected edge-weighted graph (the Delaunay Triangulation mesh meets this criteria).
4. Doubly-Chained, Breadth-First Search Algorithm
    - Used to find the diameter of the Minimum Spanning Tree (the longest distance between two vertices of a given graph). This information is then used to determine both the starting room and the final boss room of the game.
5. An Original, Work-in-progress, Lock and Key Placement Algorithm.
    - This algorithm attempts to generate locked rooms and corresponding key placements that would produce a valid, solvable path from the start of the dungeon to the final boss room.

In the class ProceduralMapGenerator, the method GenerateBSP() is first called to divide the world-space into random partitions that are able to house rooms that range from a variable minimum to a variable maximum size. The GenerateBSP() method utilizes a recursive Leaf subclass as its data structure. This allows for the abstraction of the subdivision of the world-space, turning the algorithm into one that is simply splitting the leaf nodes of a binary tree. This ultimately results in a partition like the following figure:

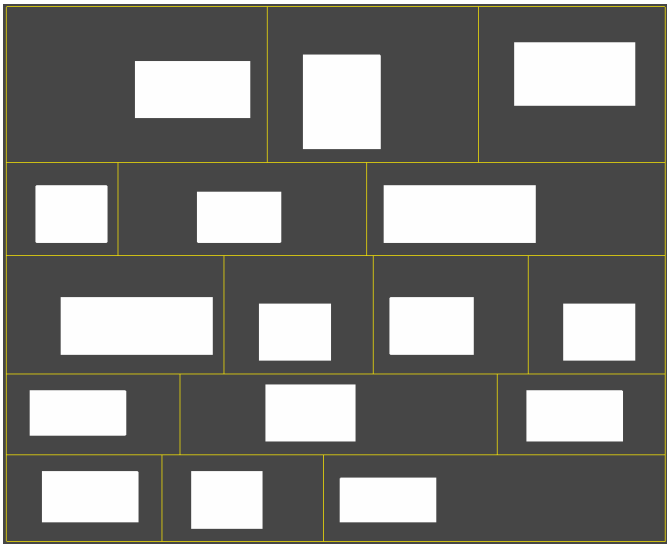


Figure 10: A 2-D bounded grid is repeatedly bisected horizontally/vertically (yellow lines) until a minimum leaf size is reached. Afterwards, a room gets randomly placed into each leaf. Screenshot taken from a debug view of the final game.

The next step in this project's procedural generation code is using the Triangle.Net library to run a Delaunay Triangulation algorithm over the central vertex of every room. More specifically, the exact algorithm that was used is the built-in Dwyer's divide-and-conquer algorithm (under the DelaunayTriangulation() method in the ProceduralMapGeneration class).

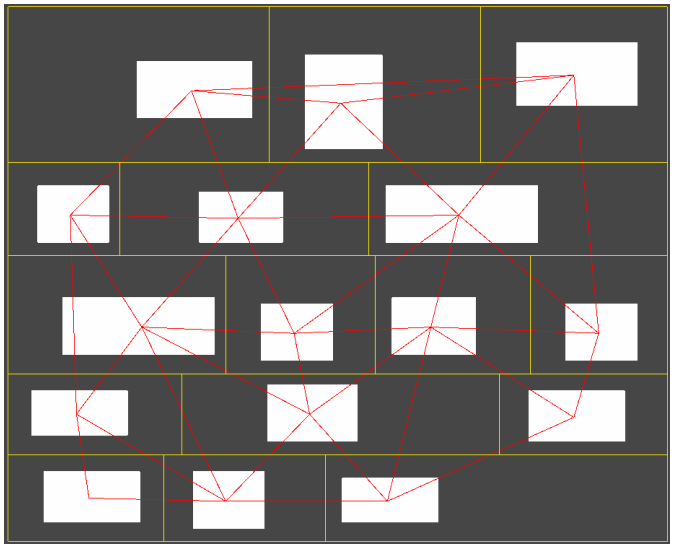


Figure 11: The resultant Delaunay Triangulation (red lines) from Dwyer's algorithm maximizes the size of the smallest angle, and by extension, it makes almost every edge a viable candidate for a direct, unobstructed hallway connection. Screenshot taken from a debug view of the final game.

In order to find the minimum spanning tree (aka MST) of a set of vertices, any standard MST algorithm can be used such as Prim's algorithm or Kruskal's algorithm. Generally, these algorithms would be used on a complete graph that connects each vertex to every other vertex (using the euclidean distance as the weights for these edges) before running the MST algorithm in $O(n^2)$ time. However, it should be noted that it is possible to speed up the time complexity by exploiting a special property of Delaunay Triangulation. A unique property of every Delaunay Triangulation graph is that it contains a MST that connects every vertex as a subgraph. By running Kruskal's algorithm on this reduced set of edges in the triangulation (relative to the number of edges in a complete graph of each room's vertex), it can be completed in O(n log n) time. The FindMST() method can be found in the ProceduralMapGenerator class.

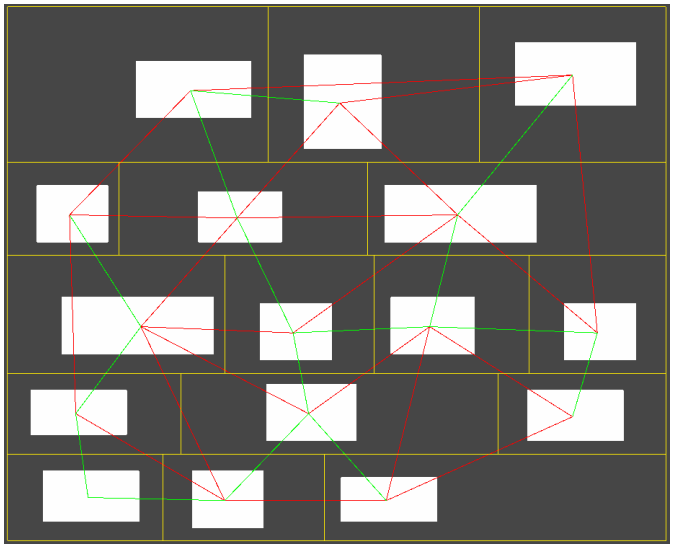


Figure 12: As seen above, the Minimum Spanning Tree (green lines) is a subgraph of the Delaunay Triangulation (red lines). Screenshot taken from a debug view of the final game.

With the MST and Delaunay edges found, it is now possible to reincorporate a small number of edges from the triangulated graph on top of the MST edges to produce a branching dungeon with some looping connections that acts in a similar vein to the game *TinyKeep* [5]. The reincorporated edges should serve at least one of two purposes: create meaningful shortcuts, or reduce the player's need to backtrack from the branching dead-ends of the MST dungeon by providing an alternative path. However, not every Delaunay edge would be a suitable candidate for reincorporation for various reasons. For instance, the final boss room should only be connected by a single hallway to one other room to give the dungeon a sense of progression and finality. If the boss room had multiple hallway connections, then a player might feel that the final boss was misplaced somewhere in the middle of the dungeon rather than being placed at the end of it.

Additionally, as these reincorporated Delaunay edges act as a shortcut of sorts, allowing a triangle of hallways to form between 3 adjacent rooms (with 2 MST edges and 1 Delaunay between 3 vertices) would yield a shortcut that bypasses only a single room. While this would not necessarily be a problem for games with dungeons that have unidirectional connections between rooms (i.e. one-way paths between A -> B, B-> C, C -> A) as this can lead to interesting dungeon designs, it is problematic for dungeons designed with bidirectional hallways in mind, such is the case with the game in this project. This is because any Delaunay edge that completes a triangle on the MST graph in this game would fail to act as a significant shortcut, nor would the edge's corresponding hallway connection be able to meaningfully reduce the player's need for back-tracking when inevitable encounter a dead-end.

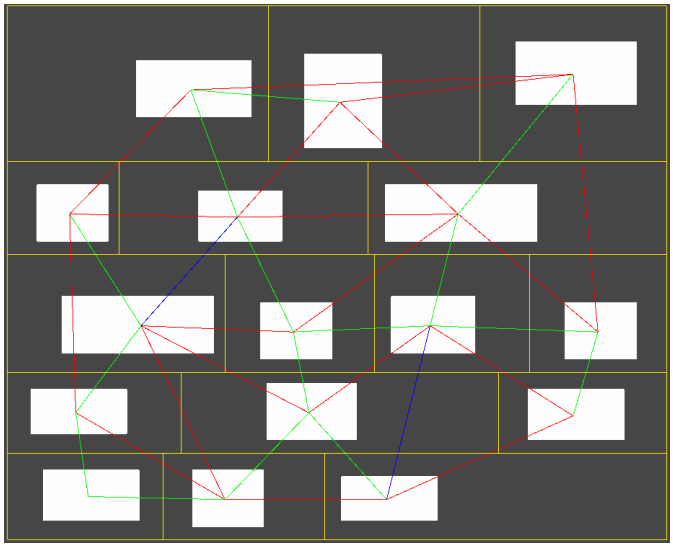


Figure 13: Some Delaunay edges (blue lines) have been reincorporated into the MST graph (green lines) to form substantial shortcuts and interesting loops within the dungeon. The Delaunay edges (red lines) that would form complete triangles with the MST were considered as non-viable candidates for reincorporation. Screenshot taken from a debug view of the final game.

For reasons that will be discussed shortly, our final game also excludes the starting room where the player spawns from being connected to multiple rooms via a Delaunay edge shortcut (there can only be a single hallway to one other room, much like the boss room constraint that was previously discussed). This means that one extra step must be taken before any extra Delaunay edges can be

reincorporated with the MST: both the starting room and the final boss room must be identified before FilterDelaunayCandidates() can be called.

A naive solution to determining both a start and end point would be to select them randomly. However, doing so would hinder player exploration as random selection could easily lead to the player spawning in a room that is right next to (or is very close to) the final boss room. In such instances, the player would have no incentive to explore the majority of the dungeon since they could just immediately go straight to the boss room. Even if locked doors and keys were implemented to restrict a player's ability to immediately enter a boss room that is close to spawn, players might become frustrated with exploring away from the boss room since back-tracking would become an inevitability. A better, less naive approach might be to randomly select rooms that are found on the branch-ends of the MST graphs, but this too also suffers from the same problems as the last proposal.

With both these naive solutions yielding unsatisfactory outcomes, it becomes apparent that both the spawn room and the boss room need to be spaced far apart (to compel dungeon exploration) on the outermost branches of a MST graph (to ensure the start/boss rooms have only one connecting hallway). In other words, the best approach to solving this issue would be to identify a diameter for the MST graph and then select the diameter's end points as either the starting room or boss room.

The FindGraphDiameter() method in the ProceduralMapGenerator class utilizes a novel approach to quickly identify the diameter of a MST graph. Although there is plentiful literature on the issue of finding the diameter of a graph, it should be noted that the amount of literature varies greatly depending on whether the graph in question is: cyclic or acyclic; directed or undirected; identically or dissimilarly weighted; and lastly if the weights are negative or non-negative. The most popular method of finding the diameter of an arbitrary, connected graph without negative cycles is to use a modified Floyd-Warshall's All-Pairs Shortest Path (aka APSP) algorithm. Instead of having the APSP algorithm return a matrix of all of the shortest paths for all pairs of vertices, the algorithm would just need to return the longest of the shortest paths between all pairs of vertices. For added clarity, this specifically refers to the shortest path between any pair of vertices that has the greatest number of edges.

Although the Floyd-Warshall's APSP algorithm would have been more than sufficient for the FindGraphDiameter() method in this game, there is a faster and better alternative method that can exclusively be used on acyclic, undirected graphs (a Minimum Spanning Tree is an acyclic, undirected graph). However, it should be noted that literature on this precise approach is lacking as finding the diameter of a tree seems to be a trivial affair that has never warranted a formal name. The approach involves running two breadth-first searches back-to-back in the following way:

1. Pick any arbitrary vertex u
2. Find u such that $e(v, u)$ is maximum
    - A breadth-first-search algorithm with v at the front of the queue is used to identify u
    - Note: $e(v, u)$ denotes the eccentricity, or length, between vertex v and u. This specifically refers to the number of edges between v and u.

3. Find w such that $e(u, w)$ is maximum
    - A breadth-first-search algorithm with u at the front of the queue is used to identify w
    - Note: $e(u, w)$ denotes the eccentricity, or length, between vertex u and w. This specifically refers to the number of edges between u and w.
4. Return vertices u and w
    - These vertices are returned because $e(u, w)$ will have the largest eccentricity/length in the MST graph, thus $e(u, w)$ is the diameter.

Running the FindGraphDiameter() method will locate a suitable starting room and final boss room along the MST's diameter as seen in Figure 12. It should be emphasized that the FindGraphDiameter() method is called before FilterDelaunayCandidates() (the method which finds suitable shortcut candidates and reincorporates them into the MST). Additionally, a miscellaneous hallway placement algorithm is needed for a 'lowering' phase. At some point, the abstract vertices and edges will need to be translated into tile-able dungeon floors/walls with the appropriate collision physics. Figure 12 will also exhibit the resultant hallways after 'lowering' the MST/Delaunay edges into actual hallways.

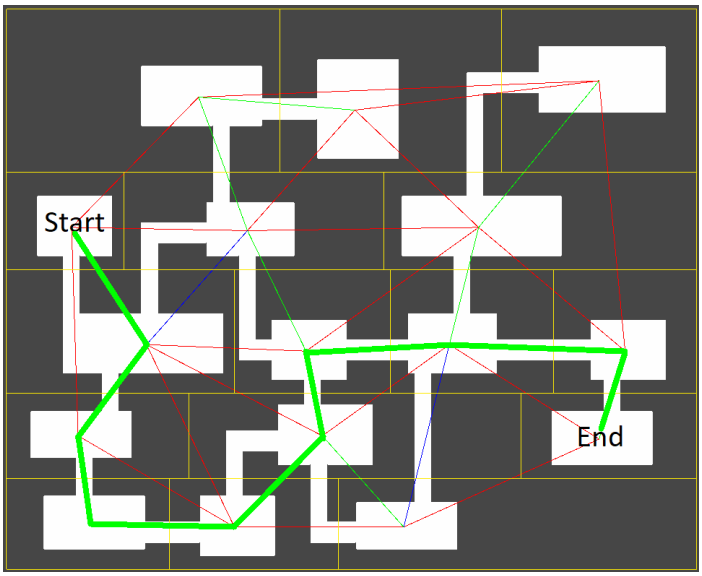


Figure 14: The player's starting room ("Start") is the furthest number of rooms away from the final boss room ("End") along the critical MST path. The critical path refers to the shortest and most direct path the player can take (thick green line) barring any Delaunay shortcuts (blue lines). In-game hallways that correspond to the MST/Delaunay edges are also shown (white paths). Screenshot taken from a debug view of the final game.

Lastly, an algorithm to populate the dungeon with locked doors between rooms and solvable key placements is needed to divert the player off the critical path (the shortest, most direct path between the start and end rooms). Without any locked doors and keys, a player could in theory skip the majority of the procedurally generated rooms and head straight to the boss. This is something that should be avoided because this makes the dungeons less challenging if the player

can skip multiple rooms full of enemies. Moreover, the linearity of the critical path could feel too monotonous if experienced players eventually learn how to locate the approximate location of the boss room consistently (the furthest distance from the start room).

Just like the games from the Zelda franchise, the keys found in this game's dungeons are capable of unlocking any door as opposed to being paired to one specific locked door. Additionally, no special equipment currently exists in this game that acts as a "key" that enables the player to bypass some obstacle that acts as a "locked door" (for instance, bombs that could destroy obstructing rocks). However, it would definitely be possible to adapt the lock-and-key puzzle algorithm to incorporate more than one type of key if special equipment items were implemented alongside their corresponding obstacles. Currently, the game's algorithm to place the locked doors and keys in a solvable manner is as follows:

1. Spawn a key and locked door in the starting room. The player cannot progress without picking up the key and unlocking that very first door, and so having an adjacent key/locked door will serve as a tutorial on how to unlock locked doors.
2. Traverse from the starting room to the final boss room along the critical MST path until a fork in the path is discovered.
   (a) On the room with the forked path, use a modified BFS algorithm to determine which pathway leads to the final boss room and which pathway leads to a dead-end branch of the MST.
   (b) On the pathway that leads to the final boss room, place a locked door.
   (c) On the pathway that does not lead to the final boss room, traverse to the room furthest away from the fork and spawn a key in that room.
   (d) Repeat steps (a) - (c) for all subsequent rooms with multiple branching pathways along the path to the final boss room.
3. In the penultimate room that leads to the final boss room, place a locked door on the hallway between those two rooms if no such locked door currently exists. Then, randomly select a room that was previously locked and spawn a key in it.

With this algorithm, a procedural output like the one found in the following figure is generally expected (please note that the following figure will not have any Delaunay shortcut hallways as not every procedurally generated map is guaranteed to contain a Delaunay shortcut hallway):

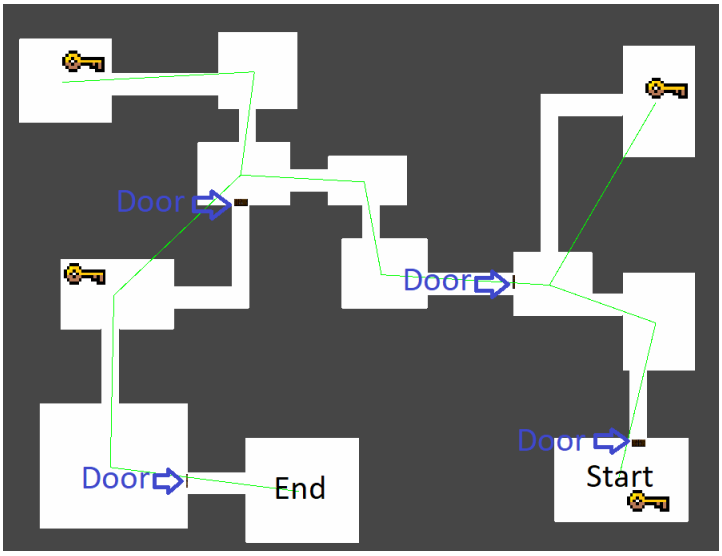


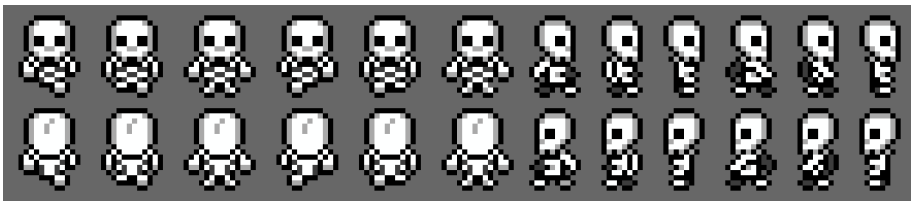
Figure 15: The lock-and-key placements in this figure was generated using the aforementioned algorithm. The algorithm first instantiates both a key and a locked door in the starting room before it goes searching for forks to find suitable rooms to spawn both a locked door and a key. Screenshot taken from a debug view of the final game.

Additionally, the difficulty of each room should be increasing the further away each room is from the start (and the closer the room is to the end). As only one type of enemy currently exists in this minimum viable product, only the number of skeletons is varied across each room (the player can expect to encounter more skeletons the closer they are to the boss room and fewer skeletons the closer they are to spawn). However, it should be noted that the exception to this is that the skeletons found in the final boss room have a wider field of view and they take 3 hits before they are killed instead of the usual 2 hits that all the skeletons in the other rooms have.

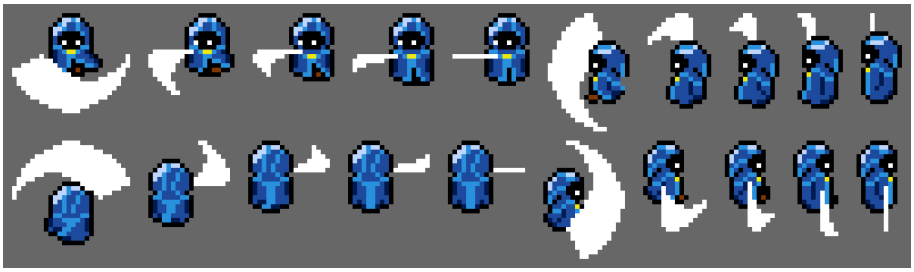
4.2 Graphical Methods

While designing the game, a lot of effort and care was put into creating the 2-D pixel art and animation assets for both the player's and the enemy's sprites. All the art and animations are original assets that were made from scratch. These assets were made using the 2-D pixel sprite editor/animation tool called Aseprite. Most of the animations in this game went through a lot of through a lot of trial and error to make them look just right. Individual pixels were constantly being shifted until the transition between any two frames became seamless. Reference guides on 2-D walk cycles and melee sword attacks were used to ensure that standardized 2-D pixel animation principles were being implemented accurately.

(a) Player's Walk-cycle Animation Sprite Sheet



(b) Enemy Skeleton's Walk-cycle Animation Sprite Sheet



(c) Player's Attack Animation Sprite Sheet

Figure 16: Taken from the art asset folder of this project.

Another graphical problem that had to be surmounted was the issue of clipping sprites when using Unity's built-in 2-D orthographic camera. Although the game in this project is technically a 2-D game, the art assets were designed in a such a way to give the appearance of a three-quarter's view (a type of axonometric projection). Generally, 2-D games that utilize an axonometric projection are often categorized as having a 2.5-D (two-and-a-half-dimensional) perspective. These types of games tend to have gameplay that is restricted to a two-dimensional plane while the environment appears to be three-dimensional. The following figure will demonstrate the 2.5-D aspect of this project:

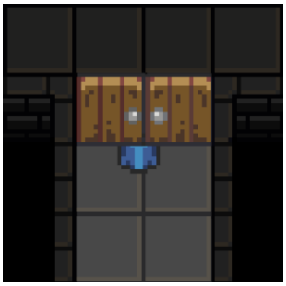
(a) Player correctly appears behind the door when they are situated above it

(b) Player correctly appears in front of the door when they are situated below it

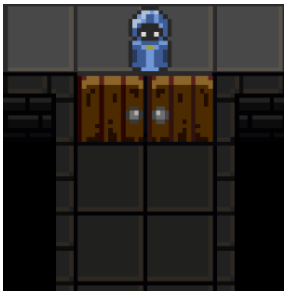
Figure 17: Screenshots taken from the final game.

While Unity does have native support for rendering sorting layers which allows for player and enemy entities to always appear in front of background/level elements or vice versa, their implementation is completely static. In other words, if a player is designated to a sorting layer that will cause it to be rendered on top of environmental objects, then there is no way for the environmental object to ever be rendered on top of the player (the same applies to the converse case).

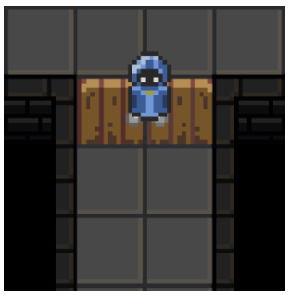
(a) Doors are statically rendered on top of the player: no clipping



(b) Doors are statically rendered on top of the player: clipping occurs



(c) The player is statically rendered on top of doors: no clipping

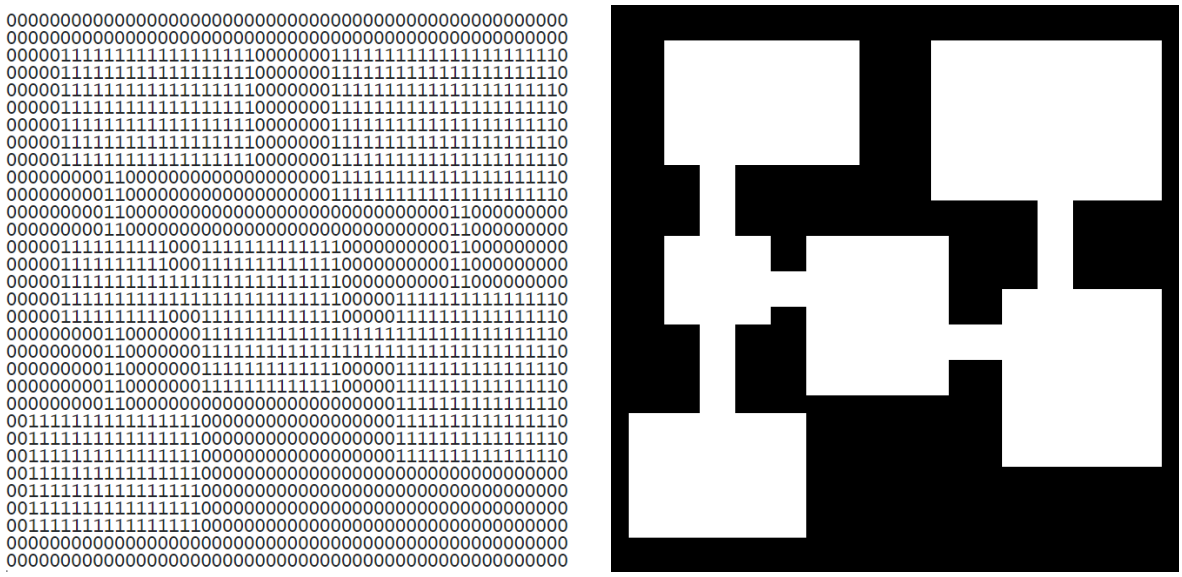


(d) The player is statically rendered on top of doors: clipping occurs

Figure 18: Screenshots taken from an early development build of the game.

This issue was ultimately resolved in the final version of the game (as seen on Figure 17) with the implementation of the PositionRenderSorter.cs script that is attached to every game object that needs to be dynamically sorted. In summary, this script will constantly check a GameObject's Y-position in the world to determine its sorting order within a rendering layer. GameObjects that have a high Y-position will have a lower sorting value, thus they will be rendered first and so any other GameObject with a lower Y-position will be rendered on top (as they will have a higher sorting value).

Another major graphical problem that had to be resolved was the issue of translating the binary matrix output of the procedural map generation algorithm into the appropriate floor or wall tiles the game. In essence, the algorithms in the ProceduralMapGeneration class all write to a singular 2-D matrix whose rows and columns corresponds to the X and Y coordinates of the actual game. This process is demonstrated in the following figure:



(a) Binary Matrix Output from ProceduralMapGeneration

(b) Resultant Tiles Placed in the Game's World-space

Figure 19: Visual diagram of the game's backend processes.

What is particularly remarkable about Figure 19 is the fact that it demonstrates how the procedural dungeon algorithm only has a binary output that does not try to distinguish the perimeter from the area that is contained within. This distinction would be useful in determining whether a floor or wall tile should be placed. However, in the final game, there are many different types of wall tiles that work together to make this game look 2.5-D and so another method of determining what type of wall is appropriate relative to its placement (i.e. cardinal-direction facing walls, inner-corner walls, outer-corner walls) is needed.

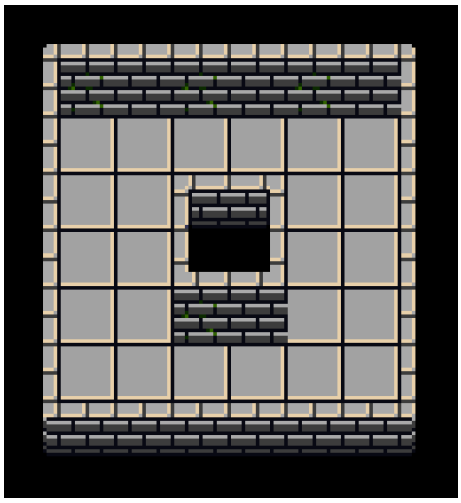


Figure 20: An exhibition of all the various types of walls found in the game. The middle section showcases all the inner-corner walls, while the outer-corner walls can be seen on the corners of the image. Of particular note, every top wall is actually composed of a pair of vertically adjacent wall tiles to give this game its axonometric perspective. Sourced from this project's original art assets.

The solution that was ultimately chosen for this project was to implement the Rule Tiles GameObject from Unity's 2D-Extras package. With this, a single rule tile can replace all the multiple different tiles for each of the unique walls/floors. This is because each rule tile can adapt its sprite to match its surrounding tiles.

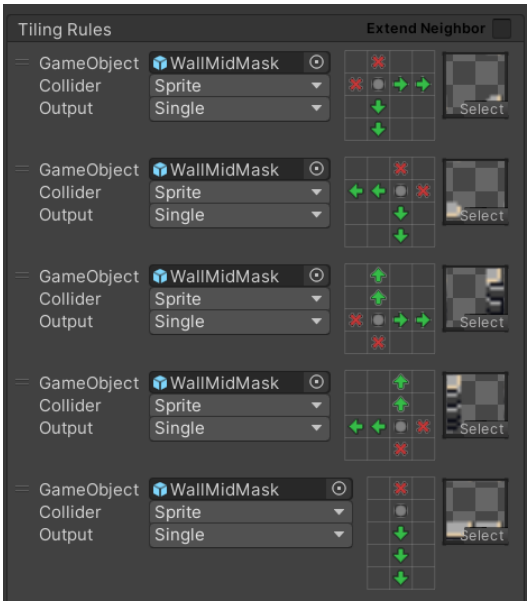


Figure 21: By examining the configuration of its neighbors, each rule tile can dynamically adapt itself to become the correct type of wall or floor sprite. This image is only a snippet of all the adjacency rules (the green arrows indicate the necessary presence of a tile; the red crosses indicate the necessary absence of a tile). Screenshot of Floor.prefab as seen in the Unity Editor.

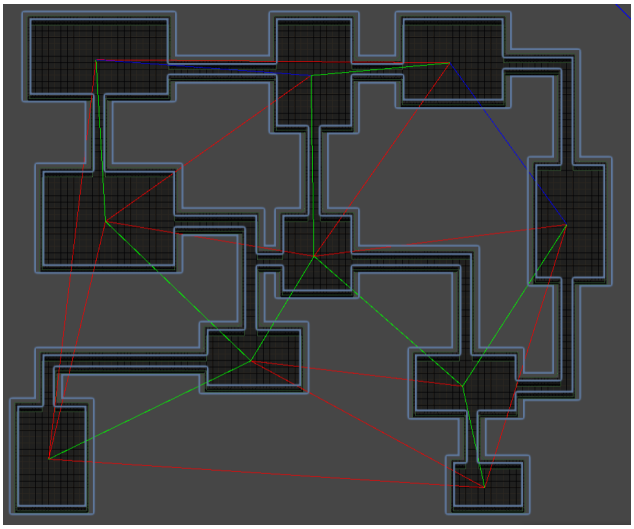
5 Ethical Considerations

This game was developed with an E10+ ESRB (aka Entertainment Software Rating Board) rating in mind. The E10+ rating indicates that the game is only suitable for ages 10 and up. The game features cartoon/fantasy violence and this criterion is consistent with the E10+ rating. Although skeletons and enemy combat are featured in this game, blood and gore are completely absent from the game (instead the player/enemy flashes red and plays a hurt sound effect whenever they get hit). Additionally, the game also contains no explicit language nor does it contain any mildly suggestive themes.

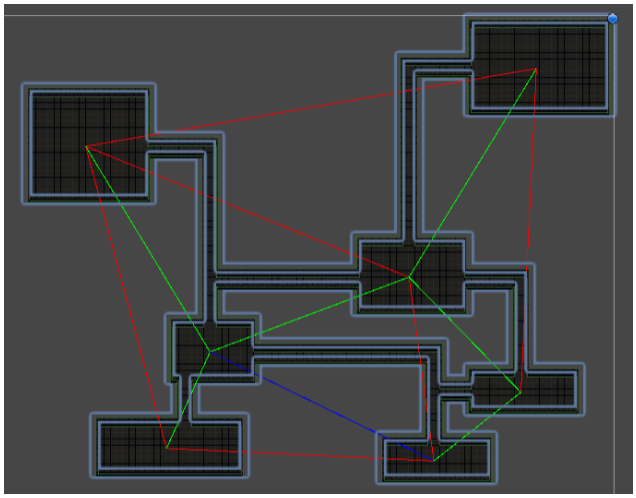
6 Future Work

There exist one major bug that needs to be patched out in a future release. Currently, the algorithm may occasionally produce a bugged dungeon because of an unreliable 'lowering' phase. Because the procedural dungeon algorithm decides on the hallway connections and lock-and-key placement based on the graphical representation of the dungeon (it treats rooms as vertices/nodes

and hallways as edges), 'lowering' this abstract representation of the dungeon into something concrete may cause some problems with the game's current design implementations.



(a) The 3-way hallway in the middle is inconsistent with the graphical representation of edges



(b) The 3-way hallway in the bottom right is inconsistent with the graphical representation of edges

Figure 22: Screenshots taken from a debug view of an early game prototype.

As demonstrated above, occasionally the 'lowering' phase will cause 2 or more hallways to become connected to each other. Graphically, however, this is impossible as no two edges can ever intersect each other; instead, they can only be connected through a common room/vertex. Moreover, occasionally the overlapping hallways may cause doors to be misplaced and hallways to be merged with rooms in unintended ways.

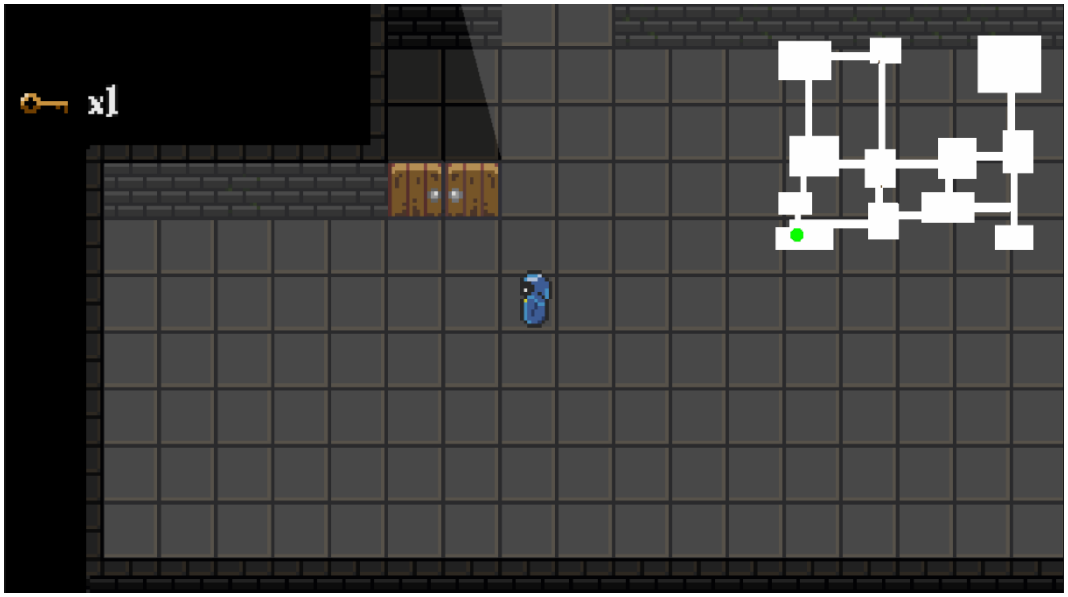


Figure 23: In this in-game screenshot, the horizontal hallway has accidentally merged with the room along its length. As a consequence, the door placement is bugged. Very rarely, this type of bug caused by the 'lowering' phase may make getting to the boss room/getting a victory impossible. Screenshot taken from the final game.

Although some optimizations to the procedural dungeon generation algorithm have been made to reduce the frequency of this happening, the source of the problem has not been addressed. Currently, the hallway placement algorithm is a static algorithm that does not take other hallways into account. What needs to be implemented is a smart hallway path-finding algorithm that can avoid merging with other rooms and intersecting with other hallways.

Lastly, the project will need many more features in order become a fully-fledged game. As it stands, this game can be best described as a minimum viable product since the game contains only 1 type of enemy. While the skeleton's basic random-walk/attack-on-sight AI is fit for purpose (the skeleton's behavior are predictable given enough playthroughs), the lack of variety in enemy types makes the game too easy for the player to achieve mastery. Moreover, the game also has a single boss which happens to be missing a competent AI. Currently, the boss has only has an idle state and thus it is unable to directly attack the player since it lacks an attack state.

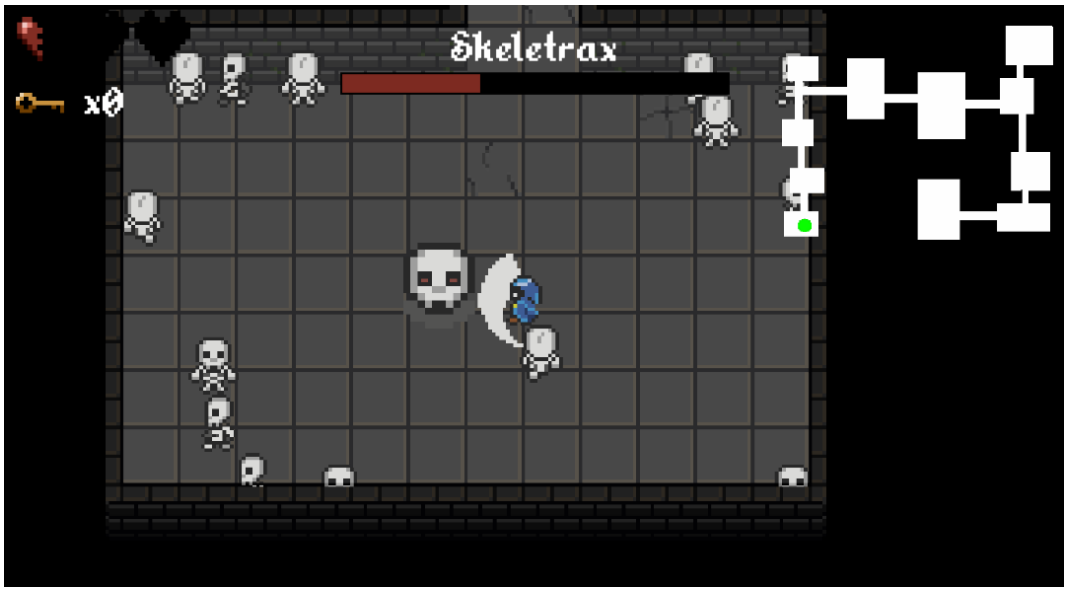


Figure 24: The current boss fight completely relies on the skeletons to attack the player on the boss' behalf since it has no attack AI. Screenshot taken from the final game.

Aside from adding more enemy and boss variety, the current implementation of the lock-and-key puzzle algorithm is capable of being augmented to work with special equipment based keys and locks (in the same vein as the item-based puzzles found in the dungeons of the Zelda franchise).

References

[1] A Adonaac. Procedural Dungeon Generation Algorithm. Sept. 3, 2015. URL: https://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php.

[2] Mark Brown. Roguelikes, Persistency, and Progression - Game Maker's Toolkit. Jan. 28, 2019. URL: https://www.youtube.com/watch?v=G9FB5R4wVno.

[3] Cellular Automata Method for Generating Random Cave-Like Levels - RogueBasin. URL: http://www.roguebasin.com/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels.

[4] Tom Coxon. Procedural Dungeon Generation - Part I - Bytten Studio. Jan. 21, 2012. URL: https://bytten-studio.com/devlog/2012/01/21/procedural-dungeon-generation-part-i/ (visited on 05/12/2020).

[5] Phi Dinh. My Procedural Dungeon Generation Algorithm Explained : roguelikes. May 4, 2013. URL: https://www.reddit.com/r/roguelikes/comments/1dodsv/my_procedural_dungeon_generation_algorithm/.

[6] Joris Dormans. A Handcrafted Feel: 'Unexplored' Explores Cyclic Dungeon Generation – CONTROL500. URL: https://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/.

[7] Travis Fahs, Justin Davis, and Lucas Thomas. IGN Presents the History of Zelda. Aug. 27, 2010. URL: https://www.ign.com/articles/2010/08/27/ign-presents-the-history-of-zelda.

[8] Simon Howard. BSP Partitioning in 3-D Rendering. URL: https://soulsphere.org/apocrypha/bsp/.

[9] Alexander King. The Key Design Elements of Roguelikes. Apr. 10, 2015. URL: https://gamedevelopment.tutsplus.com/articles/the-key-design-elements-of-roguelikes%E2%80%93cms-23510.

[10] Alex McHugh. What is a Roguelike? July 11, 2018. URL: https://www.greenmangaming.com/blog/what-is-a-roguelike/.

[11] Júlio Rodrigues. How to procedurally generate a dungeon using the BSP method on Unity - Bladecast. URL: http://bladecast.pro/unity-tutorial/how-to-procedurally-generate-a-dungeon-bsp-method-unity-tilemap.

[12] Derek Yu. Golden Idol. Aug. 8, 2013. URL: https://spelunky.fandom.com/wiki/Golden_Idol.