Game Design-friendly Procedural Generation of Game Levels

# Abstract

Procedural Generation (PG) is often used in games for smaller aspects of the game’s scope and as such is scarcely used to produce entire levels for games, specifically in the Action genre, however, the increasing popularity of Roguelike games has led to further exploration with the concept of fully PG levels and experiences. The aim is to solve this through the utilization of the A\* Pathfinding algorithm and the Delaunay Triangulation method managed by a cost function. The benefit of this is that the pathfinding will be guided to avoid structures within the game environment.

# Goals and Reasoning

The goal of the implementation of this project is to use and modify an existing workflow to better understand the ideas behind it as well as to explore the balance between Procedural Generation and Designer Built content. This was chosen as a topic since procedurally generated content makes individual game development more viable and as a technical challenge to build a fully autonomous dungeon generation system on which to build an action-roguelike.

Action-roguelikes form a good structure to use this sort of level generation due to their nature of requiring multiple runs to make progress and each run is different from the last such as in *Hades*, and the *Binding of Isaac*, thus the dungeon will rely on a library of prebuilt encounters, to form the gauntlet the player faces on each run.

Building an action-roguelike presents unique challenges in the sphere of system design to accommodate the need for multiple runs, difficulty scaling, and player progression. These systems are deeply entwined and form the basis for any roguelike experience. However, most mainstream roguelikes use a top-down or isometric camera, an added layer to the project will be implementing a third person follow camera like *NieR: Automata. Hades* and *NieR: Automata* serve as the main inspirations for this game since *Hades* uses a relatively simple model for combat which has enemies having clear distinctions and abilities based on enemy types, while *NieR: Automata* presents a great controller for the character to switch between combat and non-combat states seamlessly with situational cameras.

This project is meant to improve on previous games I have made, expand my understanding of the Unity Engine, Algorithms and models – such as A\*, Triangulation, and Data Structures - often used in games, and act as an implementation for game design-related research, specifically related to roguelikes and combat design.

# Introduction

For Procedurally Generated content to be Game Design-friendly, the content must use elements of handcrafted and designed content in tandem with the content produced by the system when run. This means that for this project, a system will be built to showcase the power of Procedural Generation while still using the work and content created by a designer without the designer creating entire levels themselves. To that end, a Roguelike game will give the structure needed to create a game in which the player aims to clear a dungeon of enemies without knowing the layout of the level each time the player enters the game.

Roguelike games such as *Hades* (SuperGiant Games, 2020) feature a string of rooms filled with enemies, these rooms form the level or dungeon that the player must clear to either finish the game or progress the story. *Hades’* dungeon however does not have the rooms occupying the same space, instead, the rooms are treated as if in a vacuum. The proposed system for a procedurally generated dungeon is then to have these rooms explicitly connected through hallways and have them coexist with one another.

To achieve this and achieve a desirable design level, a similar workflow to Vazgriz’s *Procedurally Generated Dungeon* (Vazgriz, 2019) is adapted to use prebuilt rooms that contain at least one designed encounter inside. The proposed workflow has been chosen due to its flexibility and to lighten the work needed to create the hallways by hand.

The workflow includes three key concepts to succeed, Delaunay Triangulation (DT) (Rebay, 1993), Minimum Spanning Trees (MST) (D.Kalpanadevi, 2013), and the A\* (A Star) pathfinding algorithm (Xiao & Hao, 2011). DT allows the rooms to be connected to each of the closest immediate neighbours, MST allows for the path through the dungeon to be the shortest route through every room, and A\* will handle the positioning and groundwork for the generation of hallways connecting each room as dictated by the MST.

Once completed, the workflow should produce a well-connected dungeon that includes all the placed rooms with the shortest path. The workflow requires a low level of input from the developer at runtime and runs fully autonomously once started.

# Literature Review

Vazgriz’s *Procedurally Generated Dungeon* (Vazgriz, 2019) outlines the basest parts of the Dungeon Generation Workflow and provides a good base on which to build a usable Procedurally Generated Dungeon. The workflow places cubes of random sizes on a grid and then places unit cubes on the paths between the rooms, while this is effective for building unique structures, the project’s current capabilities are not fit for what is needed to build a game without some modifications and expansions on the base system. The main modification is the usage of prebuilt rooms and encounters, with defined entrances and exits. This is a clear improvement over the coloured cubes already used. This also allows for designing encounters inside the rooms as the rooms are built using the Unity Prefab system, and thus are instantiated with all their child objects in the correct positions.

Most modifications will lie in the placement of the rooms themselves, the endpoints of each of the paths, and the creation of hallways. Rooms must take up the correct amount of space on the grid, even when rotated, endpoints must match the entrances and exits to the rooms, and the hallways need to be walkable for a player – including floors and walls to form a coherent dungeon between all the rooms.

# Methodology

**Dungeon Generation – 2 weeks**

This method is largely based on Vazgriz’s workflow (Vazgriz, 2019), but with modifications to make the resulting dungeon more useful as a staging ground for a rogue-like game – to replace cubes with assets.

* Prebuilt rooms are placed within the bounds of a grid.
  + The rooms are chosen randomly from a list and then placed at random coordinates.
  + The grid is of a specified size that can be adjusted without causing faults.
    - The size of the grid can be a limiting factor to the number of rooms being placed.
* The rooms are then connected into a triangulated mesh through a Delaunay Triangulation.
  + This triangulation uses the Bowyer-Watson Algorithm (Rebay, 1993).
* The edges of the Delaunay Mesh are then used to create a Minimum Spanning Tree (MST)
  + This is obtained through Prim’s Algorithm (D.Kalpanadevi, 2013).
  + This gives the shortest path that visits all vertices on the Delaunay Mesh.
* Each of the edges left in the MST is then used to guide the Pathfinding Algorithm
  + The specific Algorithm is A\* (A Star) (Xiao & Hao, 2011)
  + It uses a cost function[[1]](#footnote-1) as well as the start and end points to find the fastest route from A to B.
* Given the data and changes to the grid, the hallways are then built procedurally.
  + A\* marked paths for the hallways to take.
  + Floor tiles are placed at the marked coordinates.
  + Floor tiles are then combined into one mesh.

**Gameplay Engineering – 4-8 weeks**

Once the dungeon is completed, the focus will shift to creating gameplay within the dungeon. This will need to include:

* Character Controller
  + Capable of combat, motion, and interaction
  + Using custom animations and character model
  + Player Stats and Manager
    - Experience Handling
    - Skill Progression
* Level Rules
  + Establish failure/Success conditions.
    - Ideas for having different goals for different runs.
      * Time Trial
      * Key Hunting
      * Clearance
* Enemy AI
  + Purely Combat AI
    - Different behaviours depending on the type.
  + Enemies to be sourced from Mixamo with as many premade animations as possible to save time.
* Combat Design
  + Making AI behaviour
  + Player Combat Actions
  + Combat Modifiers

Following this stage, the remainder of project time will be given to playtesting, extra polish, and reacting to playtesting feedback. Should the time prove to be unfilled by any of these, the dungeon will be expanded to include verticality in the dungeon[[2]](#footnote-2)

## Difference to References

Vazgriz’s workflow (Vazgriz, 2019) uses cubes to showcase the capabilities of the workflow showcased, however, it uses the centre of Room cubes as the destination for all the paths, which leaves the chance for the paths to hit dead ends by missing room entrances, and may mean that predesigned encounters would be difficult to implement and will take too much time to find the designated spaces and place the enemies.

Figure Vazgriz Dungeon Layout and Action



A screenshot of a video game

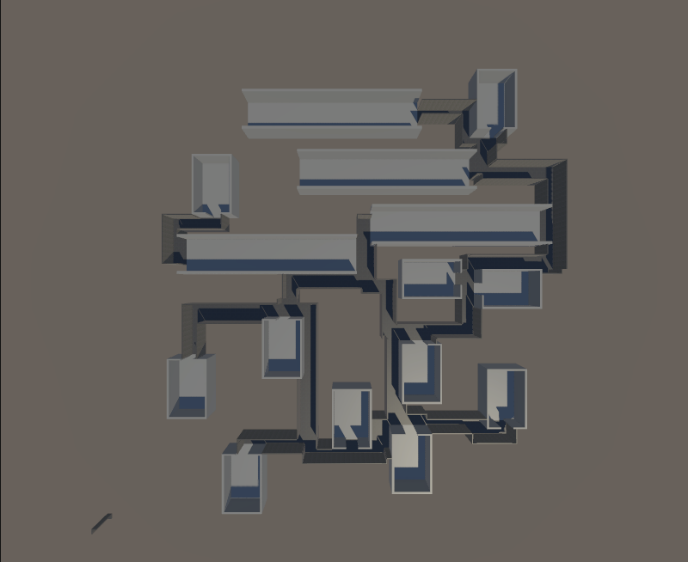
Description automatically generated/As seen above, the generated paths in blue are one unit wide (1m in the Unity Engine) and pass through rooms to reach the room’s centre, the demonstration is meant to showcase the designation of space on a grid to mimic a possible layout for a dungeon, thus it shows how rooms would connect to each other through the pathfinding algorithm.

Figure Built Dungeon Layout and Action

By manipulating the paths' start and end points and the Pathfinding cost function, the hallways form a very different shape to Vazgriz’s own system. The positions of the entrances change the shape of the halls the algorithm outputs since the designation of a hallway cell on the grid will alter the cost of a given cell and will be described fully. The cost function is altered to make existing hallways far more appealing to the algorithm as pathways to the destination.

The dungeon itself is far larger than Vazgriz’s showcase to allow for a greater amount of available space, thus much larger rooms and wider hallways fit much more comfortably as a level for a game.

## Grid Mapping

Grid mapping is used primarily for the designation of space within a given range. Grid positions can currently be referenced as empty, a room, a buffer, a soft buffer, or a hallway. These designations help to keep rooms separated and to clearly demarcate where hallways can and cannot go. In Fig 2, the room, soft buffer, and hallway cells are shown in blue, white and red respectively.

Hallways can fill any cell on the grid that is not a room, but the type of room before being a hallway changes the value of that position when the Pathfinding Algorithm calculates the cost of the path, e.g., other hallways are made much cheaper to increase the likelihood that hallways will lead into each other.

The grid also helps to keep items equally spaced and easily accessible to the code and other scripts given its use of discreet[[3]](#footnote-3) values.

## Room Placement

Room prefabs are placed at random coordinates on the grid. These coordinates only include integer XY positions thus meaning that rooms need to occupy even amounts of space to keep the room allocation accurate across the entire dungeon.

Room allocation is the process of marking all positions within a room as a room to tell the Pathfinder to steer clear of those positions. This process is trivialized with the use of the Unity RectInt library as it can return a list of all the positions inside that space to be processed efficiently.

While a room is being placed a few things need to successfully occur; a random rotation must be performed on the room[[4]](#footnote-4), the entire room must fit inside the grid’s designated space, and it must not encroach on another room’s space. If any of these conditions fail the room is reset and the process restarts.

Once a room is placed a buffer is placed around it to give space for hallways and to present the other rooms with a reference to its presence in the general area, thus ensuring a gap between the rooms.

## Delaunay Triangulation

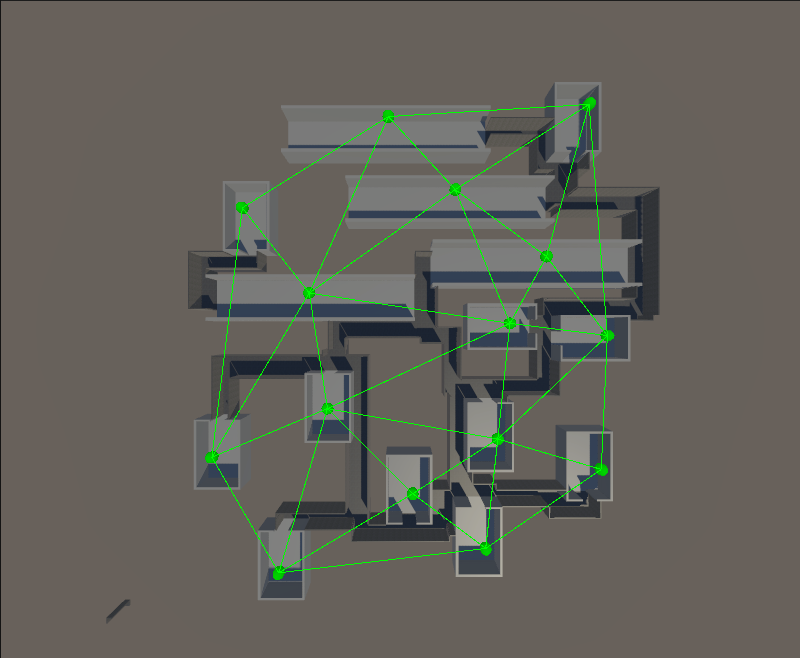
Delaunay Triangulation is the process through which a series of points relate to edges to create a mesh composed only of triangles. These triangles must not intersect with each other’s space; the connected points must be the most /immediate neighbours to the points on the triangle.

Figure : Triangulated Mesh Using Room Poitions

### Bowyer-Watson Algorithm

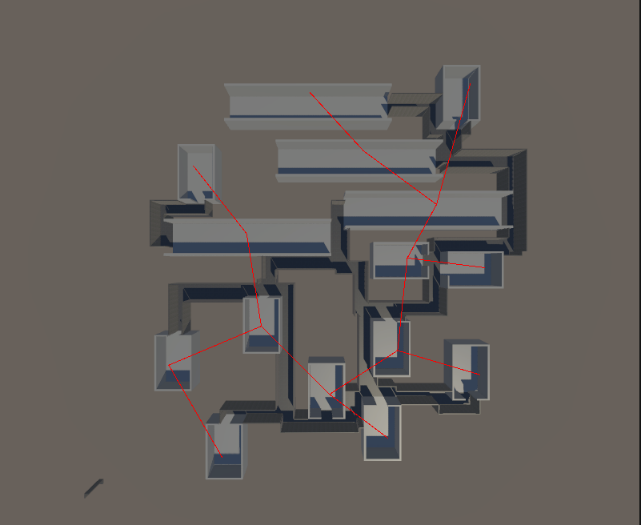
The Bowyer-Watson Algorithm builds the triangulation in such a way that when a new point is added to the mesh, the mesh is not made to recalculate itself completely but only the parts of the mesh that would be affected by that new point. Below is the MST Generated from reducing the Delaunay Mesh through Prim’s Algorithm.

Figure Result of MST connecting Rooms

## Minimum Spanning Trees and Prim’s Algorithm

A minimum spanning tree is used to connect every point on the Delaunay Mesh with the most efficient path possible given the positions of the points in relation to one another. The tree must include all points on the mesh and thus ensures that all rooms placed in the dungeon are included in the level.

## A\* Pathfinding

A\* is a pathfinding Algorithm that expands on Dijkstra’s Algorithm to be more efficient in its path choices as well as more receptive to being manipulated by the data in the space.

A\* requires a start point, endpoint and a cost function to carry out its duties as a path-finding algorithm. The cost function uses the distance between the start and the endpoints as well as the grid data to calculate the viability of a given direction. This function is run for each orthogonally adjacent point to the current position. For this use case, the cost function reacts to the type of cells these adjacent points are as seen in Figure 2.

The cell with the lowest cost is chosen to be expanded on as it is either in the ideal direction or is a desirable cell type. The cost function is what allows A\* to be manipulated to follow soft rules[[5]](#footnote-5) and achieve a more desirable outcome.

## Hallway Generation and Building

After the grid has been correctly populated with hallway, room and buffer cells, the walls are marked and then the hallway mesh is printed out in a tile[[6]](#footnote-6) fashion, then the individual tiles are combined into a single mesh.

### Wall Marking

The grid is iterated row by row, comparing each cell to the cells around it, should the cell and its neighbours match certain criteria it will become a wall facing either east, west, north or south. The grid is iterated through again this time focusing on walls and the marking of hallway corners.

### Hall Printing and Combination

Now that all walls and corners are marked on the grid, it is iterated once more to place the corresponding tiles on the correct grid coordinates. As the hallway at this point is a collection of individual objects that are sitting next to one another, the next step is to combine each of the meshes into a single mesh while removing the original objects.

This creates a single cohesive object that reaches all the rooms that have been placed while keeping the number of objects in the scene to a minimum.

# References

D.Kalpanadevi. (2013). Effective Searching Shortest Path in Graph Using Prim's Algorithm. *International Journal of Computer & Organization Trends*, 310-313.

Rebay, S. (1993). Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm. *Journal of Computational Physics*, 125-138.

Vazgriz. (2019, November 18). *Procedurally Generated Dungeons*. Retrieved from VAZGRIZ: https://vazgriz.com/119/procedurally-generated-dungeons/

Xiao, C., & Hao, S. (2011). A\*-based Pathfinding in Modern Computer Games. *International Journal of Computer Science and Network Security*, 125-130.

1. The cost function refers to the values A\* gives to each cell it evaluates as it chooses its path. [↑](#footnote-ref-1)
2. The inclusion of upper and lower floors in the dungeon generation process. [↑](#footnote-ref-2)
3. Discreet values refer to the use of integers over decimal value numbers. [↑](#footnote-ref-3)
4. The range of rotations is 0, 90, 180, and 270 degrees for simplicity. [↑](#footnote-ref-4)
5. Soft rules are guidelines that change the behaviour of an algorithm as it is allowed to do some things more often than others. [↑](#footnote-ref-5)
6. Floor and wall prefabs are placed on their corresponding spaces on the grid. [↑](#footnote-ref-6)