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 (Has become 6)**

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 (Shifted to accommodate revised timeline)**

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

* Character Controller (WIP)
  + Capable of combat, motion, and interaction
  + Using custom animations and character model
  + Player Stats and Manager
    - Experience Handling
    - Skill Progression
* Level Rules (Pushed to after Crit 2)
  + Establish failure/Success conditions.
    - Ideas for having different goals for different runs.
      * Time Trial
      * Key Hunting
      * Clearance
* Enemy AI (WIP)
  + 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 (Pushed until AI shell is complete)
  + 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 1 Vazgriz Dungeon Layout and Action



A screenshot of a video game

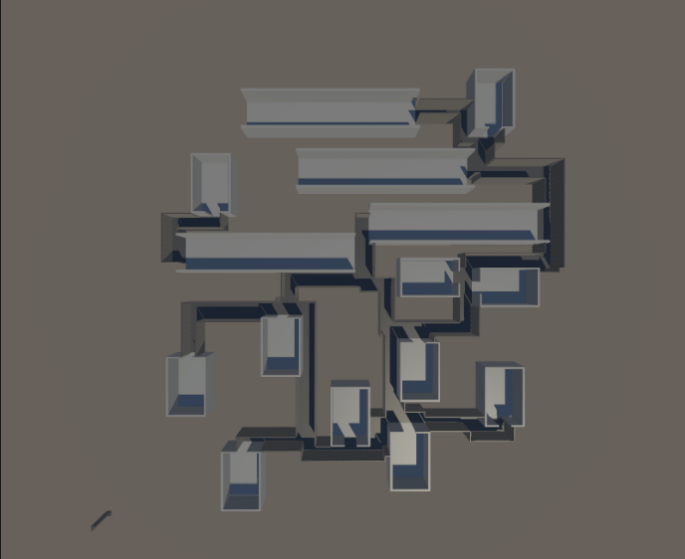
Description automatically generatedAs seen above, the generated paths in blue are one unit wide (1m in the Unity Engine) and pass through rooms to reach the room’s centre, the demonstration is meant to showcase the designation of space on a grid to mimic a possible layout for a dungeon, thus it shows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

Figure 2 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

Rooms are placed at random positions on the grid. Rooms are referred to as objects known as variants. Each variant stores the empty room prefab, an integer reference to an enemy formation, the room’s nominal value, the room’s bias value, and the shifted bias value.

The nominal value is calculated from the enemy formation, using the total health and damage of the enemies in the given formation.

A room’s bias value determines how likely it is for that room to appear in the dungeon layout with that particular formation and is calculated using the nominal value. It is calculated below.

A room’s bias always lies between 0 and 1. The function above is known as a Gaussian Curve or a Bell Curve. The number of runs completed shifts the peak of said curve so that as the player completes more runs, the more valuable the rooms that appear are. From here the shifted bias is calculated, this is simply the running total of room biases as the code iterates through the sorted array.

Array[0].shiftBias = array[0].bias

Loop (array[1] -> array[last], int i)

Array[i].shiftedBias = array[i].bias + array[i-1].shiftBias

Float lineLength = array[last].shiftedBias

The above pseudocode calculates the shift biases of each room variant and thus effectively maps the variants onto a line of length *lineLength.* Each room is seen as a segment of this line, and to choose a room, a random number is generated between 0 and *lineLength* and then the array is iterated through to find a variant with a greater shifted bias than the generated number.

Once a room has been chosen the prefab is placed, the occupied space of the room is checked for other rooms that may be present. If there are none, the room is finalized by spawning the enemies in the chosen formation, and the room is allocated its space on the grid.

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 and to inform other rooms that the space is occupied already. 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

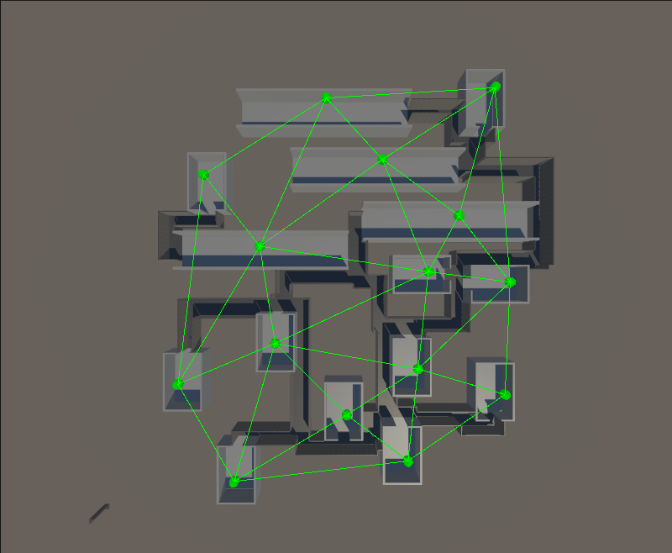
Delaunay Triangulation is the process through which a series of points relate to edges to create a mesh composed only of triangles. These triangles are such that no point can exist inside the circumcircle of any other triangle in the mesh.

Figure 3: Triangulated Mesh Using Room Positions

### 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. This algorithm is an expansion of the Incremental algorithm, meaning that vertices are introduced to the mesh one by one.

The algorithm begins with a super triangle whose circumcircle encompasses all vertex positions. As a vertex is added to the mesh, the triangles which the point interferes with undergo a flip[[5]](#footnote-5), which could sometimes lead to the entire mesh being recalculated, however, the more vertices in the mesh, the less likely this will occur. Once all points have been included in the mesh, the super-triangle is removed, and the complete mesh is left.

Delaunay Triangulations can be performed beyond the second and third dimensions.

## 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 ensure that all rooms placed in the dungeon are included in the level.

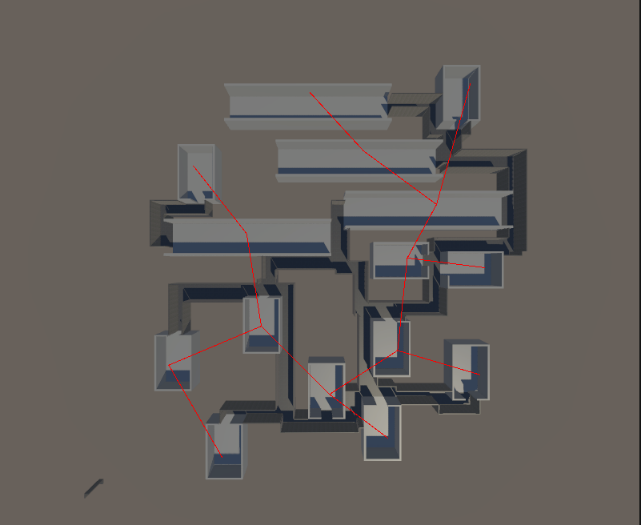
Minimum Spanning Trees ensure that there are no loops in the generated path while connecting all points with the lowest distances. This makes them useful for telecommunication applications.

Figure 4 Result of MST connecting Rooms.

### Prim’s Algorithm

Prim’s Algorithm is initialized with a single point in a mesh. It then weighs all edges connected to that point and chooses the cheapest one – in this case, the shortest connected. This is then repeated until all points in the mesh are included in the tree. Edges that connect a point already included in the tree are not included as loops would return an infinite tree weight.

## 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[[6]](#footnote-6) and achieve a more desirable outcome.

For this instance of the algorithm, the cost function evaluates the type of cell that its neighbours occupy, meaning it assigns values to the cells based on the distance between the cell and the end, and the type of cell. To force the algorithm to move around rooms, cells occupied by rooms are made extremely expensive, and to create connected hallways, cells marked as halls are made extremely cheap. A\* is customized primarily through this function, by adding weighted positions to the grid, the algorithm can produce connection nodes where all hallways meet.

As A\* is usually used for AI navigation through a space, this implementation is fairly unique in its use case.

## 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[[7]](#footnote-7) 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.

# Player and Enemy Design

## Player (WIP)

The player’s character possesses 4 main stats: Health, Strength, Vitality, and Endurance.

### Damage – Strength

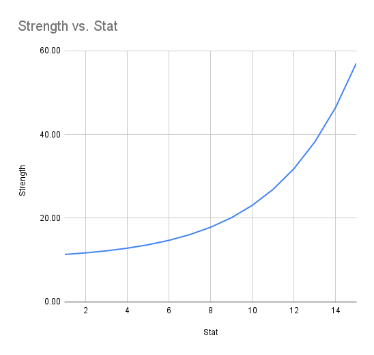
Strength affects the amount of base damage the player does to enemies through attacking. Strength scales exponentially with its level, and its level increases through the total amount of damage the player has dealt over their career in the game.

Figure 5: Graph showing the graph of Strength vs Damage Output

The damage dealt scales exponentially with the player’s strength level to allow for a late-game snowball effect as the player increases their strength level – leading to a greater return on investment in terms of player power.

### Defence – Vitality

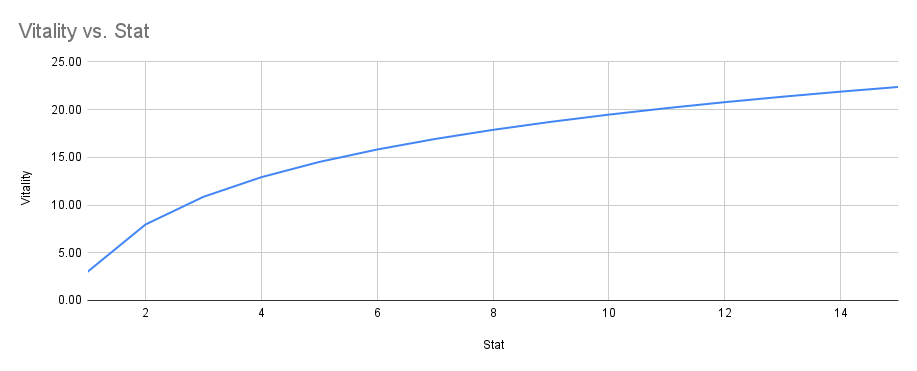
The player’s defence stat is subtracted from incoming damage, the stat is scaled logarithmically with the Vitality level. A logarithm was chosen as the model due to how the curve flattens as the value of the level increases, if a player’s defence stat were to exceed a given amount, then player skill would not matter. As skill is a crucial element, the player must not be able to negate most damage.

Figure 6:Function showing Vitality vs Defense

A logarithmic scale will give diminishing returns as the player levels up their Vitality, until hitting a “soft cap”, Vitality is meant as an aid to struggling players, but must not become a crutch for unskilled players to enable them to play mindlessly.

### Stamina - Endurance

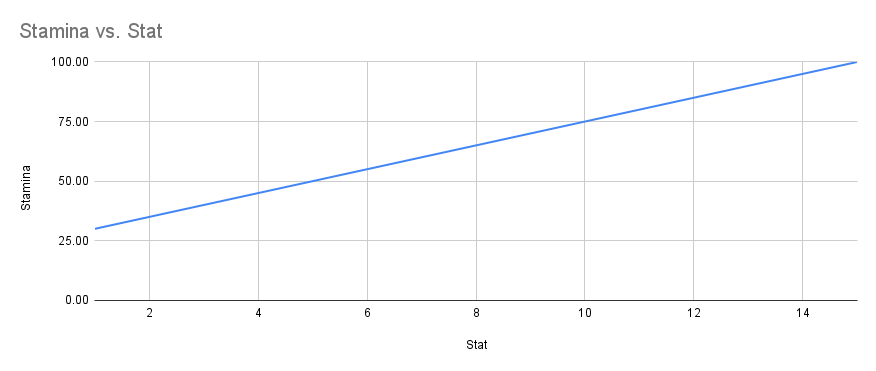
Endurance can be thought of as the player character’s fitness level, and like any fitness, it must be trained. Endurance affects the amount of stamina the player has and how quickly that stamina regenerates over time. Stamina is needed for the player to sprint, dodge and perform special attacks. The player’s endurance level increases as they use stamina points.

Figure 7: Function showing Endurance vs Stamina.

### Health

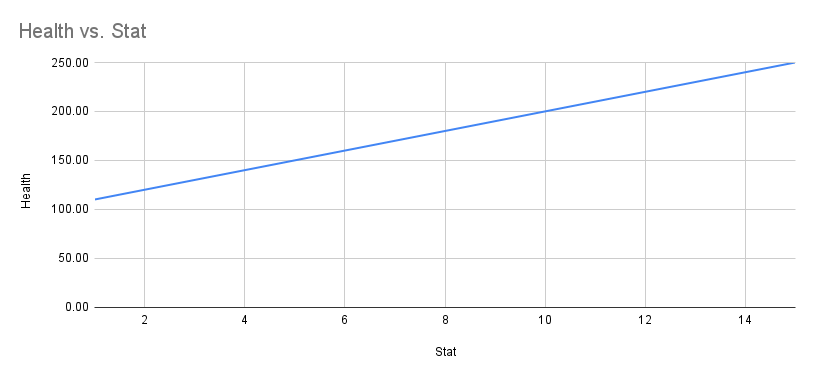
Health affects the number of health points the player has. It scales linearly with its level which increases with the total number of levels across Strength, Vitality, and Endurance. As such the player’s overall, experience is indicated through this stat.

Figure 8:Function of Health Level vs Health Points

Once the player’s health is reduced to zero, the run ends and is marked as a failure, thus not contributing to the dungeon’s curve – this is meant as a subtle message that the player is not ready to move on to the next “stage” as the pool of available rooms to the dungeon will not have changed.

## Enemy (WIP)

As it stands there are three enemy types, creeps, ranged, and brutes. Creeps are the typical melee enemies; they will attempt to overwhelm the player with numbers. Ranged enemies are similar in that they are not threatening on their own but in numbers or when paired with other enemies, they serve to deal damage to the player from a distance. Brutes are tanks, slow hitting and sponges for damage. Their goal is to draw the player’s attention as the main threat in the encounter while creeps and ranged enemies take advantage of the distraction.

### Enemy Stats

Enemies have 3 stats, health, strength and aggression. Enemies do not scale to meet the player’s level as that feature is handled through the dungeon generation process. Health and strength are like the respective stats of the player character, and aggression represents the likelihood that an enemy will decide to attack a given tick[[8]](#footnote-8).

# Reflections

## Crit 1 to Crit 2

Following Crit 1, the goal had become to populate the dungeon with content as well as to allow the dungeon to serve a purpose outside of being generated by allowing the player to run through it. However, that plan had to change due to a lack of design for the game itself.

The game had not taken shape on paper yet, and so the implementation of the game had been more difficult than it needed to be. The goal then, before making the dungeon playable, is to decide how the dungeon would take shape and how I wished it to respond to the player’s progress. Player progress is measured by the number of runs the player has managed to complete (i.e., how many times the player has cleared the dungeon) and so the dungeon needs to become more difficult as the player grows more experienced – thus resulting in the dungeon difficulty being directly proportional to player progress, this curve would need to be achieved by altering the rooms that appear as the player completes more runs – allowing higher value (more difficult) encounters to appear as the player’s perceived strength increases.

Through this goal, I learned about bias curves and how I could map the value of a given room onto a curve which is then treated as a segment of a line. This approach allows for the exclusion of rooms that are not suited for the player’s current progress. The use of a Gaussian curve then allows for the exclusion on both sides – very low and very high-value rooms are excluded from being generated in the dungeon.

From this point, it was simple to implement the updates to the dungeon algorithm to follow this structure and meet the desired output. However, now with the dungeon behaving, the enemies inside needed to be able to navigate, meaning the navigation data for each room would have to be generated after the dungeon had been built – base Unity does not have a way to bake Nav Meshes at run time, however, an experimental package allows for the generation of said data. A small challenge was that the generation process left artefacts of the rooms generated and deleted which affected the navigation data – this was fixed by accident during some “advanced debugging”.

Next was bringing the player into the dungeon to finally make it interactable. This included a character controller camera rig and spawn point. The spawn point is easy enough to implement since It just needed to be included into whatever room prefab the player was starting in, The controller is simple for now with camera-relative WASD controls, a sprint, and dodge, all physics-based to allow tuning of timings and distance. The camera is hand-built due to all the overhead that Cinemachine comes with and uses a simple SphereCast to make it collide with the dungeon walls.

After implementing a usable player in the dungeon, I found the dungeon was too narrow to be comfortable, to solve this I made the dungeon corridors 2 units wider, and this has indirectly solved many issues with the dungeon itself. Entrances are not open to the outside of the walls, empty spaces where there should be walls do not occur anymore due to specific cases. The dungeon feels much better to move through now, however, the lighting in the HDRP causes very intense shadows to occur for the scene's main light, this will be softened and eventually entirely replaced by lighting inside the dungeon corridors.

## Crit 2

* Brendon
  + Focus on Character controller and combat.
  + Work to better convey theme and intention.
  + Think of deepening combat abilities and player engagement.
* Tim
  + Get Priorities Straight
  + Focus on system building and then dolling up the game to make it pretty.
    - For a technically focused project like this, the systems come first.
* Kirsten
  + Define everything better (even if it seems self-explanatory)
    - This makes the purpose of that item more prominent.
  + Get Priorities straight.

From all the feedback from Crit 2 there was a definite push that I focus on building the systems first before trying to do work I am not particularly comfortable with. This means abandoning making my own animations for now until such a time that there is nothing else left to do. I was planning to get a premade character with animations and rigging already sorted like I have for the AI, anyway, but the fact that everyone said that was the best way forward, really means that I should take it seriously and build the prototype as quickly as possible.

I think that the lecturers see the potential of the project and delivering it fully realized is going to be a real challenge since simple questions are catching me off-guard and if those questions are left unanswered, that means there are massive holes in the vision I have for this game. However, I will rise up to meet it in time, and hopefully, it can reach it potential.

First is building and fleshing out the character controller. Given the sleek high-intensity combat plan that I have in mind, the controller needs to be responsive, fast and satisfying. Responsive that the player’s input must be reflected soon, and fast so that the player can be all over the place without the controller or enemies breaking and satisfying as a combination and balance of both aspects. In essence, the combat must tread a line of challenging as well as rewarding. I believe that reward can be found in the precision of certain actions like a perfect dodge or a parry in other games (this will have to wait until the enemy Ais can do more than think about a lack of thoughts).

Then enemy AIs, I think I will need to scale up all of the rooms in the dungeon to accommodate the sprawling, far-reaching combat. But I will be starting with ranged enemies since they will need the most attention given their tendency to stay out of the player’s range.

There is plenty of work to be done, but the engagement and interest from lecturers have made the work far more exciting to meet. My goal for the next Crit is to have player’s completing runs and hopefully have started with playtesting to collect that information – and make more work for myself after Crit 3!

# 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. If the sum of two opposite angles in a quad is greater than 180, the edge that splits the quad is made to join the opposite angles. [↑](#footnote-ref-5)
6. 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-6)
7. Floor and wall prefabs are placed on their corresponding spaces on the grid. [↑](#footnote-ref-7)
8. Tick – this occurs after a set amount of time in seconds repeatedly. [↑](#footnote-ref-8)