THE DEVELOPMENT OF PROCEDURAL MAP GENERATION WITHIN GAMES

by

Arran Smedley

40406581

Dissertation submitted in partial fulfillment of the degree of
BSc (Hons) Games Development

Edinburgh Napier University

Word Count: 14,194

Authorship Declaration

I, Arran Smedley, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed.

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained **informed consent** from all people I have involved in the work in this dissertation following the School's ethical guidelines.

Signed:

Arran Smedley

Date: 20/04/2021

Matriculation no: 40406581

Data Protection Declaration

Under the 1998 Data Protection Act, The University cannot disclose your grade to an unauthorized person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below one of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

Arran Smedley

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

The aim of this project is to produce and compare three procedural map generation algorithms for a two-dimensional dungeon roguelike game. In turn, deriving assumptions from these algorithms which in turn can be used to produce detailed comparisons of performance.

The outcome was a successful prototype that had three different map generation algorithms consisting of Cellular Automata, Perlin Noise and Delaunay Triangulation. The player had control over movement to obtain results on ease of: exploration, caves that were not accessible and more.

With further work, the final application could be aimed at a variety of different fields and educational purposes such as creating a library of procedurally map generated algorithms, or a fully-fledged game, utilizing the algorithms already in place.

Contents

Figures

Tables

Acknowledgements

I would like to thank my supervisor, Dr Simon Wells for his support, guidance, advice and ideas throughout this project and my final year at Edinburgh Napier University.

I would also like to thank my second marker, Babis Koniaris for his support.

I would also like to thank Dr Thomas Methven for his continued support throughout my time at Edinburgh Napier University.

Finally, I would like to thank the school of computing for providing equipment, environment and knowledge to enable me to produce this body of work and make the most from my education.

1 Introduction

The following chapter briefly describes this study's background and context whilst also outlining the aims and objectives needed to create a successful game/system that accurately represents the procedurally generated algorithms implemented.

1.1 Motivation

Games are becoming increasingly complex and ever more demanding to develop as new hardware technologies continually enhance the prospects and expectations of what games are currently capable of. In turn, this can run the risk of developers going hugely over their budget to push the boundaries and surpass their consumers' expectations. Leading to heavily graphic-intensive games coinciding with the overly complex algorithms implemented.

The motivation to research this topic derives from a personal interest in procedural generation (proc-gen). The development and potential of procedural generation will be great in the next few years, with complex algorithms currently being developed and discovered every day.

Games development has always been a passion of mine - to create and explore - so researching this topic will be a fantastic learning experience.

1.2 Problem

The issue that surrounds procedural generation is not so much the algorithms themselves but more how they are compared, with few good ways of comparatively evaluating them. With Procedural generation algorithms all having different aspects of randomization, optimization, and program structure, creating a valid form of comparing how these elements affect the content, or game map being generated would be an asset to anyone looking to utilize any of these algorithms for their game or software.

1.3 Background

Procedural generation is the method of generating data algorithmically rather than manually.

What is meant by this is the content currently being developed is made by the computer itself (via an algorithm) rather than a person editing and creating each piece of content introduced to a game/system.

With a combination of assets acquired or made and commonly used algorithms with processing power and computer randomness, procedural generation can efficiently produce fascinating results. In the games industry, procedural generation is programmed to generate: maps, terrains, textures, 3D models, game levels, sounds, and animations. Procedural content generation (PCG) is described well in the article Togelius et al., 2011, p2, where they state "procedural content generation (PCG) in games refers to automatically creating game content using algorithms."

Procedural generation was first introduced in the late 1970s / early 1980s, where it was primarily used for roguelike games, which were directly inspired by the famous role-playing game Dungeons and Dragons (D&D). In this game, the “dungeon master” illustrated paths, enemies, and terrain by following the game’s user guide, which would be initiated using die that players would roll for the dungeon master to generate their path. In turn, this provided a foundation for which a procedural, computer-generated algorithm could develop within the video game industry.

The first roguelike games that introduced this procedural generation included Beneath Apple Manor (1978) and Rogue (1980), which implemented dungeon and game level generation. The type of proc-gen used within these games would create dungeons in ASCII, defining rooms, monsters, hallways, and treasures to produce challenges for the player, much like the tabletop game D&D.

The evolution of both hardware and software technologies within computers have allowed for increased capabilities through components such as: RAM, CPU, and GPU power being improved heavily. The algorithms used within the early days of procedural generation have been significantly expanded upon because of the capabilities now possible. In addition to this, the graphical improvement of games developing from 2-dimensional (2D) to 3-dimensional (3D) algorithms have been adapted to evolve with 3D video games.

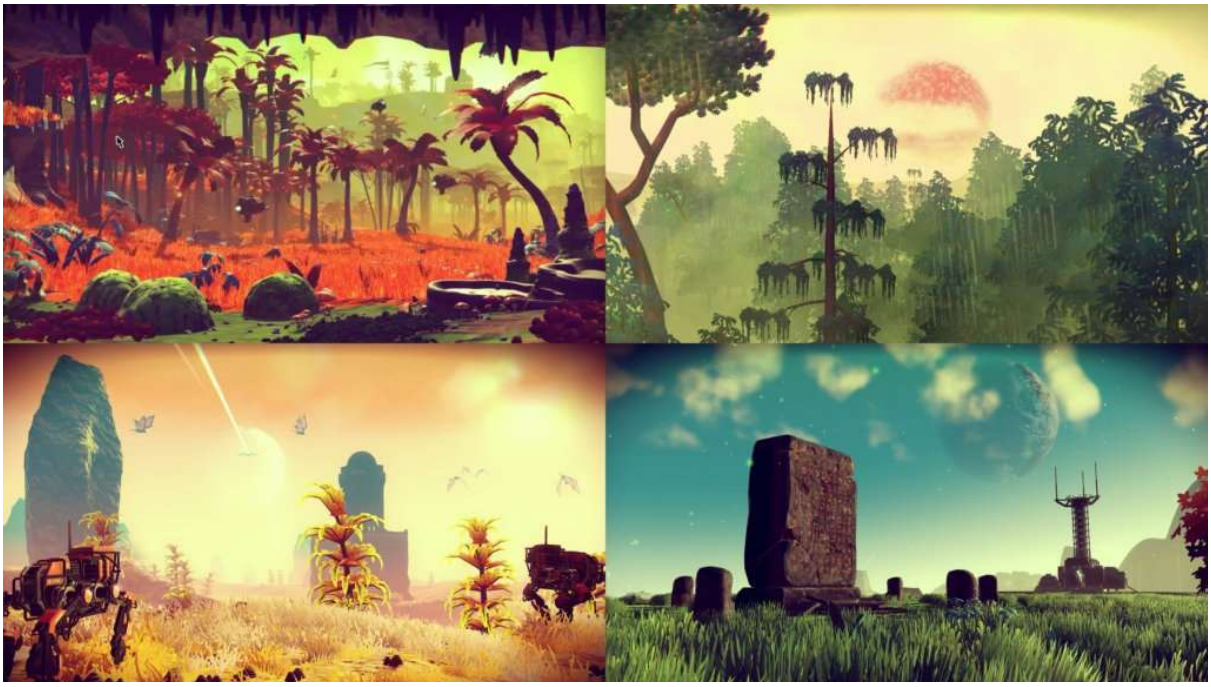


Figure 1 A representation of No Man’s Sky’s different procedurally generated worlds.

A key modern-day example of high-end procedural generation in a video game is No Man’s Sky (No Man’s Sky, n.d., Can be seen in Figure 1.) – developed by Hello Games.

No Mans Sky generates millions of planets within its game instantaneously, of which no two are the same. All of this is generated through L-Systems, a procedurally generated algorithm with a set of grammar-like substitution rules that are applied recursively to get “organic” like results, allowing the user to feel like they are traveling and exploring across a diverse galaxy (Fornander, 2013).

1.4 Aim and Objectives

This research thesis aims to explore the different procedural generation techniques and algorithms used from the past and present and illustrate their uses within a game/system. It is also be used to write a literature review based on the algorithms used to provide insight into how much these have evolved and adapted into our modern-day procedural generation techniques and algorithms.

The project objectives are as follows:

1. Research early and modern-day procedural generation techniques and choose multiple valid algorithms that can be implemented into a 2D dungeon crawler-themed game.

2. Implement these algorithms into a game/system to demonstrate the differences between each proc-gen technique.

3. Compare and contrast the different proc-gen algorithms from the early and modern-day and eras and make assumptions for each one.

4. Identify which algorithm provides the best efficiency for generating such components as: dungeons, hallways, enemies, treasures etc.

The project will aim to answer:

1. How have the different procedural generation algorithms developed over time?

2. What task-specific, procedurally generated algorithms applied in the dungeon-generated games?

3. Which procedurally generated algorithm will provide the best map-generated results (map accessibility, ease of exploration, can the player complete the level?).

1.5 Answering the question

The purpose of this project is to answer the question, "which procedurally generated algorithm will provide the best map generated results (map accessibility, ease of exploration, can the player complete the level?)".

The game will provide procedurally generated levels that can be tested for ease of exploration and, if the level can be completed. Providing results on the best algorithm for dungeon map generation.

1.6 Scope of Study

Whilst this project has the potential to be expanded to a significant degree in directions such as art/graphics, game narrative, animation, selection of algorithms, the prototype I plan to build needs to have a much narrower scope, designed to show the overall feasibility rather than every possible feature. As such, I plan to limit my work to creating a framework that provides a selection of three procedurally generated maps within a 2-Dimensional top down roguelike game that can then be given to play testers to evaluate how well each map performs in terms of randomization and other design aspects.

Figure 2 Example of Roguelike game (Legend of Zelda (1987))

The project will aim to have the three different PCG techniques referred to in the literature review. These techniques consist of Cellular Automata, Perlin noise, and Generative Grammars.

1.6.1 Boundaries

The final deliverable will not be a fully formed game title, as the project is solely focused on the procedural generation techniques implemented within the game itself. It will also not include elements such as a working user interface (UI) or extensive game art design.

1.6.2 Constraints/ Obstacles & Risks

The main obstacles and risks lie within the planning of the project, i.e., ensuring the project is kept to a tight and rigid schedule to ensure that the deadlines are met.

This project aims to investigate the most efficient procedural generation algorithm available at this current moment. In order to produce effective results, testing whether these algorithms can work and adapt in various scenarios is a must.

The potential risks that I could face include: spending too much time on a specific algorithm and therefore not spreading my time efficiently to learn enough about others to create a non-bias comparison.

Another significant risk alludes to code quality and errors that could arise within the code itself playing a significant factor in the randomness of the procedural generation techniques used. If these errors go undetected, they could produce skewed or results, leading to flawed comparisons.

An obvious but essential risk to note is game optimization - ensuring the game does not crash. As with any game, optimization is a crucial element and if not done correctly, will make it extremely hard to make any assumptions and / or comparisons.

1.7 Sources of Information

The research conducted for the literature review will be derived from various sources, both academic and non-academic. The reliability and transparency of this information will be assessed upon use. The seeds that will be used for this project will consist of:

- Books.
- Encyclopaedias.
- Magazines.
- Databases.
- Past Dissertations related to the field of research.
- Websites.

1.8 Chapter Outlines

1. **Introduction:** provides an overview of the entire project. Fundamentally bygiving a background context to: what procedural generation is, where it was derived, and how it is implemented within games. In addition to this, it will also contain a breakdown of the thesis' aim and scope.

2. **Literature Review:** covered in this chapter will be how procedural generation has evolved through time and illustrate the problems it has faced through its evolution. I will also provide an evaluation that will provide insight into the best procedural generation techniques being used now.

3. **Methodology/Approach:** a description of which algorithms were chosen for the game and how they were implemented.

4. **Procedural Generation Algorithm 1:** the analysis, design, implementation and testing of the algorithm used.

5. **Procedural Generation Algorithm 2:** the analysis, design, implementation and testing of the algorithm used.

6. **Procedural Generation Algorithm 3:** the analysis, design, implementation and testing of the algorithm used.

7. **Evaluation:** an evaluation of the algorithms used and underlying architecture.

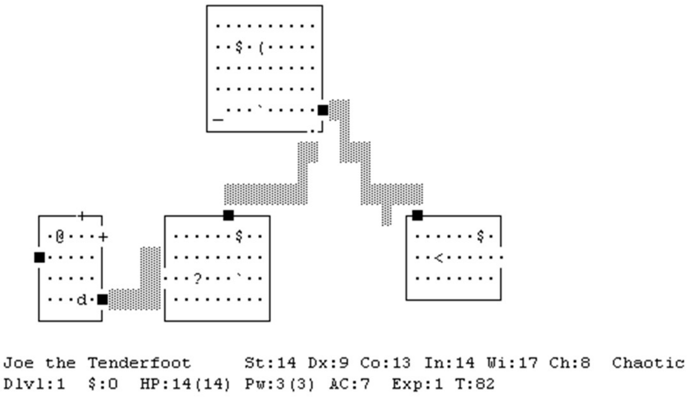
8. **Conclusion:** conclusion of the project.

2 Literature Review

2.1 What is Procedural Generation

Procedural generation has been around for decades and has developed multiple uses: level generation, terrain, modeling, and much more. Before procedural generation was first introduced in the late 1970s, content generation was done dynamically (De Carli et al., 2011).

In brief context, dynamic content generation is when all the content (graphics, modeling, and programming) is developed by the artists/developer manually. Early procedural generation within roguelike games used ASCII or regular tile-based systems to define different objects within a room or, in this case, dungeon, to present challenges to the player (Van Der Linden et al., 2014).



```
Joe the Tenderfoot     St:14 Dx:9 Co:13 In:14 Wi:17 Ch:8  Chaotic
Dlvl:1  $:0  HP:14(14) Pw:3(3) AC:7  Exp:1 T:82
```

Figure 3 A dungeon that is procedurally generated using ASCII in the video game NetHack.

Roguelike games were allegedly directly inspired by the Dungeons and Dragons (D&D) game previously mentioned. The direct comparisons can be clearly seen, by focusing on areas such as the unique rules and game patterns that D&D has inspired. D&D, in simple terms, is a tabletop, manually procedurally generated game that the dungeon master generates from the players' role of the die (Aycock, 2016).

The early motivations of procedural content generation (PCG) became apparent; with the primary goal of providing an efficient, replayable experience and allowing players to explore their creativity were common framings for a large amount of PCG research and practice.

A theme also continually emerged of hobbyists and artists using PCG as an algorithmic tool and an expressive medium, both analog and early digital PCG. What is meant by an expressive medium is that creating the generative systems can result in a reward itself for artists and designers. As many designers in the early days of PCG systems have the shared experience of spending hours just hitting the "generate" button to see the next set of unusual results would occur (Smith, 2015).

As time went on, more opportunities and hence, motivation arose for digital PCG, where the computer is generating the content for an analog game or is directly copied from one. Two bases emerged: the role of the computer as an unbiased agent (a computer program that performs various actions continuously and autonomously on behalf of an individual or an organization in an impartial manner) and the ability for computer to make the traditional multiplayer game accessible for individuals to play (Smith, 2015).

To understand the history that surrounds PCG, you must firstly comprehend the non-digital games that came before and influenced its creation.

Such as games like the tabletop D&D, which provided the platform for PCG to be created digitally and in-part, gave creators the knowledge that PCG is not just a complex algorithm but more like a set of predefined rules that can generate unusual and spectacular results in which nowadays, games can take real advantage of (Korn, 2017).

2.1.1 Overview of How Procedural Generation is Used

Within the initial design of a game, the decision to use PCG is dependent on the type of game being developed. Games like *Rogue* and *Spelunky* significantly utilise randomly generated content and rely heavily on PCG to deliver the core gameplay (Short & Adams, 2017).

When designing and implementing PCG algorithms into games, the advantages are focused upon first. Using PCG over the traditional dynamic designing and implementing every aspect of a game has come as a relief to many software engineers and designers as it reduces their workload significantly (Hendrikx et al., 2013).

Some of these advantages are conveyed by AAA games which have already been produced. Take No Man's Sky, for example, a game that's foundations are built upon procedurally generated algorithms and L-Systems (which I will focus on later). These games heavily utilize these PCG algorithms and creating extraordinary results (Short & Adams, 2017).

In contrast to this, we can look at CD Project Red's The Witcher 3: Wild Hunt, which was done primarily manually - leading to a much higher level of expenditure being used (Barriga, 2019).

The main point here is that PCG is a money and time saver, as many AAA titles take millions of dollars to develop.

To illustrate this, a good example would be if we were working on a game title that required 100-floor textures, commonly, an artist/developer would be hired to create each floor texture, and while the quality will be maintained throughout, this will cost a lot of time and money to be developed (Barriga, 2019).

Alternatively, PCG algorithms can be used here. The artist would only have to create a handful of textures, and the PCG algorithm would then look at the original creation and generate however many resources needed for the title, hence saving a lot of time and money. PCG also offers an increase in gameplay variety and opportunity (Togelius et al., 2013).

As with any dynamically developed game, the experience will be fixed. The player will be collecting the same items with the same terrain being generated each time – no matter how many times they re-load the game. Meaning the overall experience will be the same every time.

In contrast, PCG offers that variety and a sense of the unknown to the game, ensuring something new behind every door. A game's replayability is essential to keeping the consumer drawn in, as many story-driven games are very linear. Once the game is completed, it will never be touched again.

PCG offers more of a challenge to the player, ensuring that the game is replayed by posing a new challenge to the player everytime – creating a cycle.

2.1.2 Advantages and Disadvantages of Using Modern Day Procedural Generation

PCG does come with its potential drawbacks, such as project risk. It is essential to understand the different issues a developer can be exposed to when implementing PCG and understand the ramifications and impacts poor implementation of PCG can have across a project. I have highlighted some of these risks below:

- Quality Assurance (QA): As QA is a fundamental part not just within the games industry but the software industry as a whole. It ensures that the PCG algorithms are implemented correctly with little to no bugs when the games are initially presented to potential investors and finally to the consumer. Since PCG is built on randomness, this will come with some potentially severe bugs, and tweaking this to get the "correct randomness" implemented can take a lot of time (Togelius et al., 2013).

- Time Restrictions: PCG is perceived as a method that saves time when implementing certain aspects of a game/level. Although there is no guarantee that this is true. Due to the fact that the complex algorithms' design and implementation may prove to cost more time than initially budgeted (Togelius et al., 2013).

- Story-Driven Games: Games driven by a story usually intend the player to encounter various parts of the game linearly. These games benefit a lot less from the use of PCG and may even harm the game's storyline if the content does not fit neatly to the authored experience the developer has in mind (Togelius et al., 2013).

- Multiplayer: The overall control of game balance in PCG is complicated when it comes to multiplayer games. In particular, the likes of maps in real-time strategy games can be a complex challenge to developers. The many components that are operating within the game, such as: starting positions, resources, threats, and topography, affect how each player performs. A small difference such as where a player spawns within a PCG Map, could allow that player to gain certain advantages, which would be unacceptable in competitive games (Togelius et al., 2013).

With risks comes reward and logical reasons behind the use of PCG over manual game design. The ability to create massive game worlds by yourself that would usually require a team of people to produce can make it very enticing prospect. Such factors allow for this:

- Reduces Time Costs: When generating different PCG algorithms, the amount of time it takes to produce the content manually is longer in comparison. Games with heavily graphic and player-interactive-intensive maps are an excellent example, as generating a map using PCG can be a lot less time-consuming than manually making every texture such as characters, the landscape, buildings, etc (Togelius et al., 2013).

- Replay-ability: Due to algorithms generating random positions for map content or generating an entirely new map every time you play, this allows for much more replayable content, where the user can finish the game and still go back to an entirely different game world providing them with a completely different experience (Togelius et al., 2013).

- Code Reusability: With PCG algorithms generating random content within a game, code reusability such as different landscapes can be used across the board with minor tweaks made to the algorithms to change landscapes such as mountains instead of desert plains (Togelius et al., 2013).

2.1.3 Dynamic VS Static Procedural Generation

Dynamic and Static Procedural Content Generation (PCG) are systems that produce different advantages and disadvantages when developing a game/system.

Dynamic generation is a type of PCG that occurs during the game's execution. An excellent example of this is Minecraft. For example, when the player moves to different regions within the game, the regions are generated in real-time when they come within specific range of the player (Freiknecht & Effelsberg, 2017).

Static generation is the opposite of this in the way that it occurs prior to the game's active use by the player. The output of the generation could then be loaded at run-time (while the game is running) or be incorporated in some other way into the code or system (Bontchev, 2016).

When analysing these PCG types, the term procedural generation within games typically refers to dynamic generation, whereas PCG is predominantly in film, where the stories are static (Bontchev, 2016).

Most approaches to video games incorporate both dynamic and static elements. For instance, *Spelunky* dynamically stitches together pre-built parts to create dungeons.

Another example includes the Perlin noise algorithm - which we will go into more detail later - incorporates a statically computed table that dramatically improves its efficiency.

A final example, Speed tree, is a modeling system used to create static tree meshes using procedural generation techniques.

Considering we are building a system in which at run-time can generate an entire populated world. The advantages to using dynamic generation in these circumstances include:

- *Infinite variation*: The game will never run out of generated worlds as (like Minecraft) worlds will be randomly generated within a certain range of the player’s position (Bontchev, 2016).

- *Infinite size*: The game will be able to make worlds that go on forever (Bontchev, 2016).

- Infinite detail: The game will add ever-increasing amounts of detail to all aspects of a world (Bontchev, 2016).

- *Fast and efficient*: With the world only generating textures / constructs within a certain perimeter of the player, it will run efficiently (Bontchev, 2016).

- *Player input*: The player's input can alter the world's styles as everything runs in real-time (Bontchev, 2016).

In conjunction with our advantages, dynamic generation does have some disadvantages. These include:

- *Computation complexity*: If the game is to generate more exciting worlds, the player will have to wait a lot longer (Bontchev, 2016).

- *Limited control over output*: With everything being computed within the game's execution, incorporating a designer into the game can only be done at the algorithm design stage (Bontchev, 2016).

Static generation is where the generated world has already been designed and implemented with each part of the world being manually done to fit the playing experience. This is a very common way of map generation that can be seen in the company Rockstar Games with titles such as Read Dead Redemption 2 and Grand Theft Auto 5 (Freiknecht & Effelsberg, 2017).

Figure 4 Grand Theft Auto 5 Map (Grand Theft Auto 5, n.d.)

The advantages include:

- *Combination of Multiple Tools*: The generation can be used as part of a general content-creation process.

- *Computational complexity does not affect the player*: Many computational resources can be used (obviously as much as can be afforded).

The disadvantages include:

- *Data extensive*: With everything being made before run-time, the pre-generated worlds require a lot of disk space.

- *Limited to finite quantity*: With everything being generated before run-time, the worlds will be limited to a certain number of pre-generated worlds, the amount of detail also affecting the data, so you are naturally bound to determining this too.

- *Cannot respond to dynamic events*: Static content cannot be reactive to player input.

Overall, dynamic generation has the most benefits from a development standpoint, although being able to implement both in unison can also be beneficial instead of just sticking to one.

2.2 Map Generation Algorithms

Map generation (a labyrinth of different compartments and connected rooms) is one of the most common procedural generation features within video games. It is also the most researched concept in procedural generation. With an abundance of algorithms to choose from, more comprehensive / complex algorithms will yield better results. An example of some of the simpler algorithms include:

- *Simple maze generation algorithms based on girds:* These are the most straightforward and simplistic algorithms to implement. A graph with equal nodes representing spaces on a grid can create a maze-like structure that explores all the different nodes and connects them as if there is a path from one to another. Some of these algorithms go by the name of: *the Sideway algorithm*, *the recursive backtracker, the binary tree algorithm, Eller's Algorithm, the growing tree algorithm, Kruskal's algorithm, the hunt-and-kill algorithm, Prim's algorithm, Aldous-Broder algorithm, the recursive division algorithm,* and many more. With so many options to choose from, many different results can be found.

- *Binary Space Partition:* Very similar to *the recursive division algorithm* but does not work with a grid, so the outcome has a bit more freedom in terms of how the map looks. A video on how this is done is very well explained by Nathan Williams (Williams, 2014a).

- *Delaunay Triangulation:* With the rooms having to be randomly generated beforehand, working with Delaunay triangulation to create the corridors and a good map is the idea behind this method.

As we go into depth on different algorithms, we will focus on four main concepts: Cellular Automata, Perlin Noise, Generative Grammars and Delaunay Triangulation.

2.2.1 Cellular Automata

Cellular Automata (CA) was first introduced in the 1940s by John von Neumann and Stanislaw Ulam at Los Alamos National Laboratory. This was firstly represented as a two-dimensional array of cells that "evolve" step-by-step by taking the values of neighboring cells and specific rules that are put in place that depend on the simulation (Federale & Losanna, 2015).

In simple terms, cellular automation is a group of "coloured" cells on a grid of specified shapes that evolve through a collection of discrete time steps, according to a set of rules based on neighboring cells' states (Wolfram, 2006).

A massive boost to the popularity and traction to CA came from John Conway's highly addictive "Game of Life" presented in Martin Gardener's October 1970 column in *Scientific American.* With that popularity, CA still lacked some key components such as depth, analysis, and applicability and could not be presented as a scientific discipline (Schif, 2006).

As time went on, CA got a significant improvement in the 1980s when physicist Stephen Wolfram in the seminal paper, “statistical mechanics of cellular automata,” began the first serious research study of CA. In that study, Wolfram began producing some of the most iconic images found within CA's research topic (Wolfram, 2006).

Conferences and meetings were urgently formed, and people from various establishments and scientific backgrounds were being drawn into the field. It has now developed into an established scientific discipline with applications found in many science areas. Wolfram has reported having counted more than 10,000 papers referencing his original works on the subject, and the field of CA has taken an eternal life on its own (Schif, 2006).

Let us look at how Cellular Automata (CA) works. Let d be a positive integer. A d-Dimensional cellular space is Z^d. Elements of Z^d are called cells. Let S be a finite *state set*. Elements of S are referred to as states. A configuration of a d-dimensional CA with state set S is a function:

$$c: Z^d \rightarrow S$$

That will assign a state to each cell. The state of cell $\mathrm{n} \in \mathrm{Z}^d$ is $\mathrm{c}(\sim\mathrm{n})$. A configuration should then be understood as an instantaneous description, or a snapshot, of all the states in the system of cells at some moment in time. Most frequently, we consider one- and two-dimensional spaces in which cases the cells form a line indexed by Z or an infinite checkerboard indexed by Z^2 respectively (Janssens, 2007).

Next, since we are in need to count the neighboring cells, we need to establish a neighborhood vector: $\mathrm{N} = (\sim\mathrm{n}1, \sim\mathrm{n}2, \ldots, \sim\mathrm{nm})$ in which will be a d-dimensional neighborhood vector.

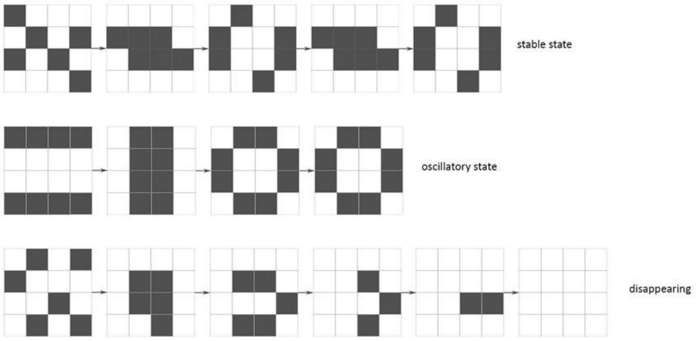


Figure 5 Sample Game of Life Objects (Cazzaro, n.d.)

As previously mentioned, John Conway’s Game of Life is a two-dimension cellular automation created in the 1970s. Its evolution is only determined by its initial state, following three rules (Figure 5):

1. Each counter with two or three neighboring counters survives for the next generation;

2. Each counter with more than three or less than two neighbors dies (is removed);
3. A new counter is placed, at the next move, on each empty cell adjacent to exactly three neighbors.

Figure 5 illustrates three different schemes that lead respectively to a stable state, oscillatory state, and all counters disappearing.

In summary: To specify a CA, one needs to specify the following items (some of which may be clear from the context):

- The dimension $\mathrm{n\ d}\ \in\ \mathrm{Z}$,
- The finite set S,
- The neighborhood vector $\mathrm{N}\ =\ (\sim\mathrm{n1}, \sim\mathrm{n2}, \ldots, \sim\mathrm{nm})$, and finally
- The local update rule $\mathrm{f}:\ \mathrm{S}^{m}\ -\!\!\rightarrow\ \mathrm{S}$

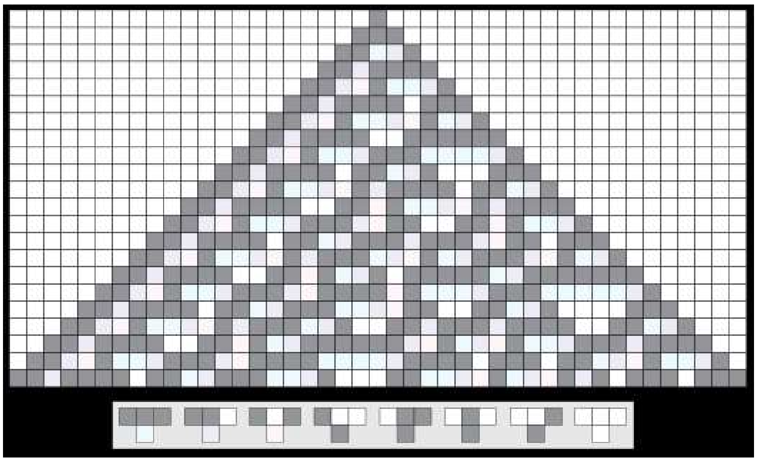


Figure 6 Rule 30 (Federale & Losanna, 2015)

Represented in Figure 6 is Rule 30. Rule 30 is elementary cellular automation introduced by Stephen Wolfram in 1983. This Rule is essential in understanding within cellular automation as it produces complex, random patterns from simple, well-defined laws that will be used within the implementation of this project. The rules are represented under the image within the appendix (Federale & Losanna, 2015).

Cellular automation within a game consists of an array, usually infinite in extent, with a discrete variable at each cell. The state of cellular automation is specified by the values of the variables at each cell. Cellular automation evolves in discrete time steps. The variable's value at one site is affected by the values of variables at cells in its "neighborhood" on the previous time step. The variable at each cell is updated simultaneously, based on the importance of the variables in their neighborhood at the preceding time step and according to a definite set of "local rules" (Wolfram, 2006).

2.2.2 Perlin Noise

Perlin noise is a type of gradient noise first introduced by Ken Perlin in 1983. As a result, his first interpretation of the look of computer-generated imagery was too “machine-like”, therefore developing Perlin noise. He formally described his findings in a SIGGRAPH paper in 1985 called an image Synthesizer (Arttu Marttinen, 2017).

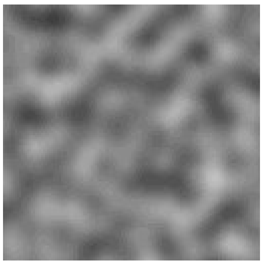


Figure 7 Example of Perlin Noise

Shown in Figure 7 is an example of what Perlin noise looks like in its raw form. Perlin noise can be used for game development for any wave-like, undulating material or texture. An excellent example of this is Minecraft in where the terrains are generated using Perlin noise, as mentioned before, using a dynamic generation (Chapter 2, Dynamic vs. Static Generation).

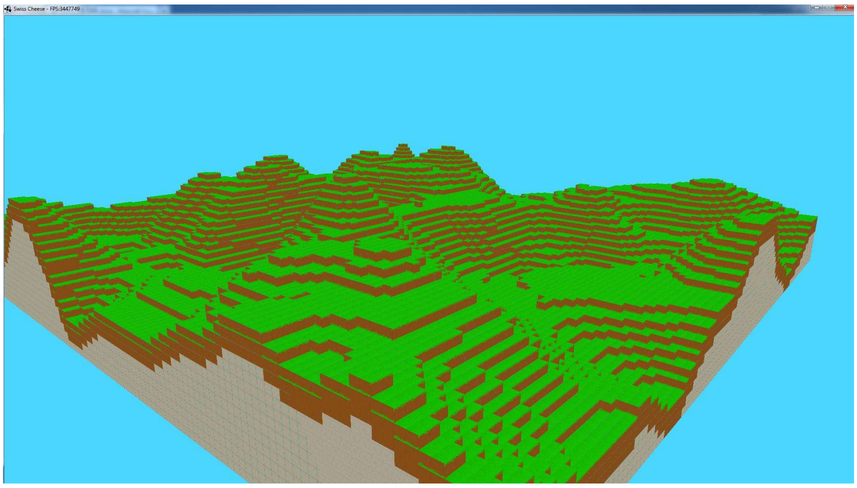


Figure 8 Perlin Noise Terrain 3D (Minecraft, n.d.)

Figure 8 represents a terrain in which Perlin noise is used to generate. Perlin noise is a function for generating coherent noise over a space. Coherent noise means for any two points in the area, the value of the noise function changes smoothly as you move from one end to the other.

Perlin noise can additionally be extended across several dimensions, as it can also be animated. 2D Perlin noise is usually interpreted as terrain, but 3D Perlin noise can be interpreted as undulating waves in an ocean sea (Erstu et al., 2012).

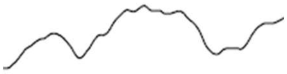
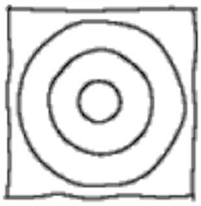
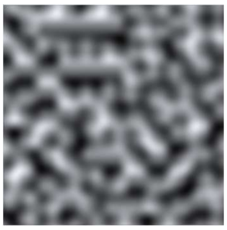
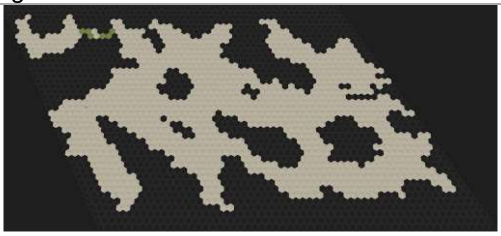
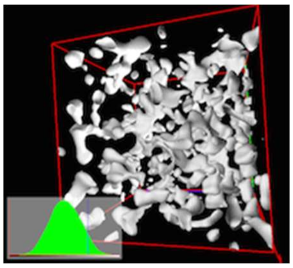
Noise Dimension	Raw Noise (Grayscale)	Use Case
1		 Using noise as an offset to create handwritten lines.
2		 By applying a simple gradient, a procedurally generated map for a dungeon game can be created.
3		 Perhaps the quintessential use of Perlin noise today, terrain can be created with caves and caverns using a modified Perlin Noise implementation.

Table 1 Perlin Noise Examples

Above is table 1 representing the different dimensions of Perlin noise and the various situations it can be used for (Erstu et al., 2012).

Since the system being developed is in 2D, I will explain in greater detail how Perlin Noise works in a 2D environment. The approach to applying the Perlin Noise is that of the same in the one-dimensional case:

- The generation of a finite sample of random values.
- The generation of a noise function that interpolates smoothly between these values.
- A sum that calculates together various octaves of this function by scaling it down by factors of 1/2, to then apply a dampening persistence value to each successive octave, so that high frequency variations are diminished.

To understand this to a greater degree, we need to investigate the finer elements. Assuming we have a *nxn* grid of unit squares (Figure 7), for a relatively small number n (e.g., n might range from certain values). For each vertex *[i,j]* of this grid, where *0 < i,j < n* , the generation of a random scalar value will be necessary *Z[i,j]*.(These values are interchangeable and are not too important to the actual algorithm. In Perlin's implementation of the noise function, these values are all set to 0, and it still produces a vibrant looking noise function.) Compared to the one-dimensional case, it is convenient to have the values wrap around, which we can achieve by setting *Z[i,n] = Z[i,0]* and *Z[n,j] = Z[0,j]* for all *i* and *j* (Mount, 2018).

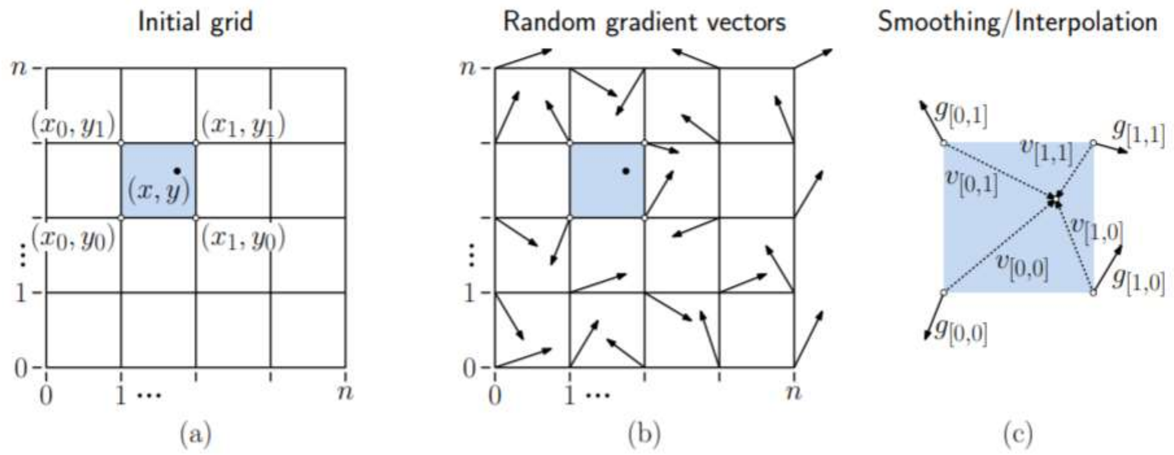


Figure 9 Generating 2D Perlin Noise (Mount, 2018)

A more straightforward approach would add smoothing to the random values at the grid points. However, the finished result would produce a very rectangular look (as every square would suffer the same variation). Alternatively, Perlin came up with a way to have every vertex behave differently by creating a random gradient at each grid's vertex (Mount, 2018).

2.2.3 Generative Grammars

Generative Grammar’s (GG) are a model for generating the syntactically correct sentences in a language using rule replacements. Noam Chomsky developed GG in the mid-1950s. According to Noam Chomsky himself, GG is “a precisely formulated set of rules whose output is all (and only) the sentences of a language—i.e., of the language that it generates” (Chomsky et al., 2019) .

In computing terms, GG is used in PCG as phrases or symbols representing actions the player must do, such as “fighting the main boss,” “pick up an item,” and so forth (Chomsky et al., 2019).

A finite state machine (FSM) is a good representation of what a generative grammar can look like for a game. FSM is a model of computation based on a hypothetical machine made of one or more states. Only a single state can be active at the same time, so the machine must transition from one state to another in order to perform different actions (Bevilacqua, 2013).

FSMs are mainly used to represent an execution flow, which is very useful to implement Artificial Intelligence within games. The “brain” of an enemy or individual within the game for instance can be implemented using a FSM: every state represents an action such as “attack” or “evade” (Bevilacqua, 2013).

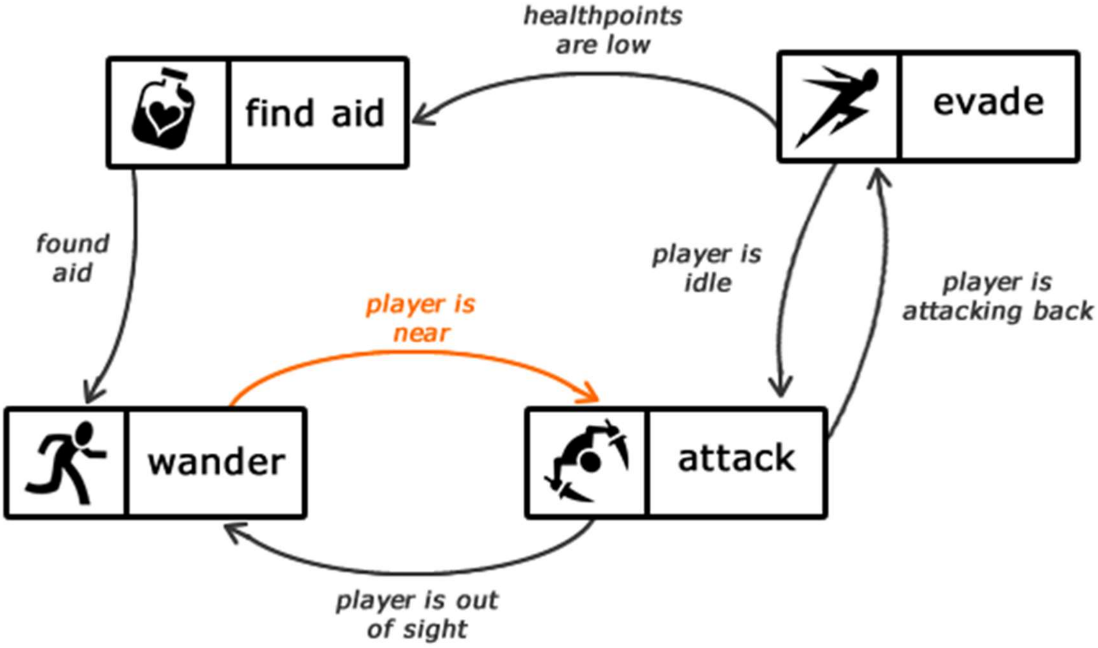


Figure 10 FSM representing the brain of enemy (Bevilacqua, 2013)

GGs are an easy and efficient way of showing both the game missions and game spaces combined to create maps and levels. For games such as RPG and roguelike, which consist of dungeons with assignments containing locks and keys, treasures, monsters, and more. A graph can be constructed through the GGs, which will provide the level in a very readable and sequential manner (White, 2019).

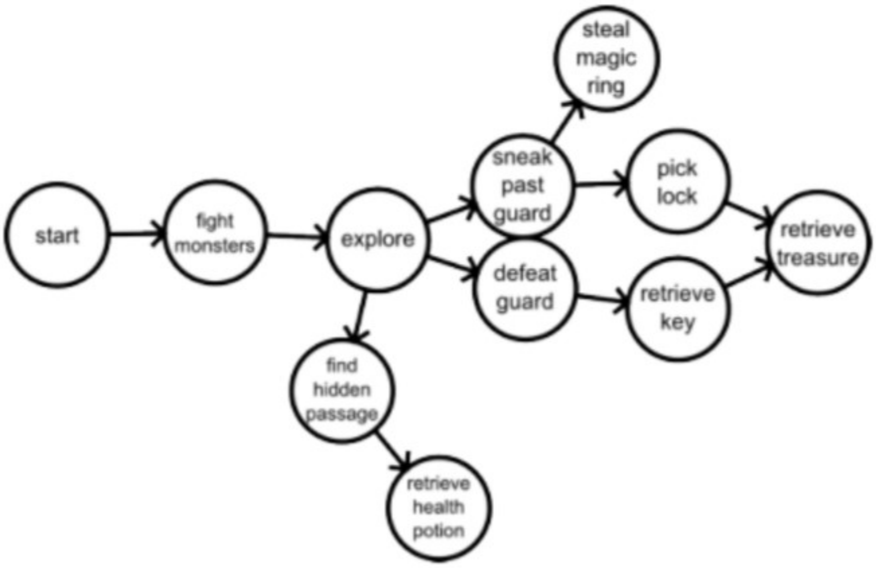


Figure 11 An Example of Level Graph Grammar (White, 2019)

Like Generative Grammar's, Graph Grammar's first requires a set of rules with which it can work with. The following (Table 2) is an example of what these rules could look like.

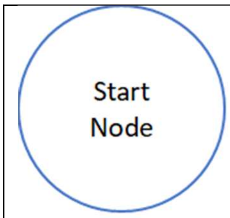
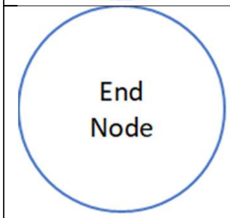
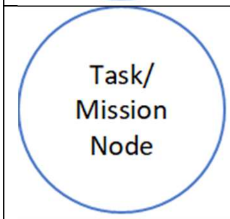
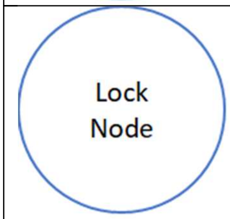
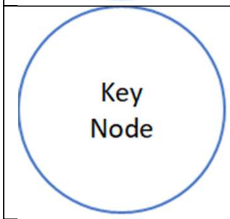
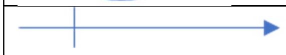
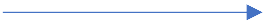
	The Start Node is the initial node in which the grammar generates from.
	The End Node is the goal in which the player can achieve to complete the level.
	The Task/ Mission node is the tasks or missions in which the player must complete. Such as killing monsters, retrieving treasure etc.
	The lock node requires a key in which the player can retrieve from the within the game map.
	The key node can be found within the map for it to be used on a certain lock node.
	The unlock edge in which specifies corresponding locks and keys.
	An edge that defines the connecting nodes

Table 2 Nodes and Edges

With this basic set of nodes and edges, a procedural generation can then be put in place to construct a set of basic rules in which the GGs will follow (White, 2019).

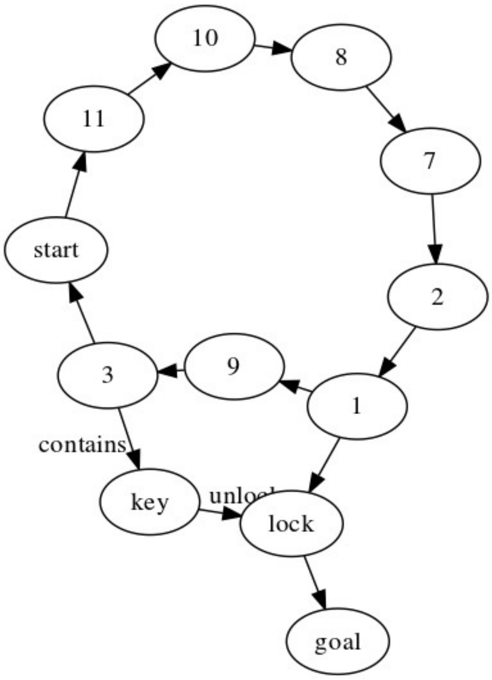


Figure 12 Set of rules and Task (Lavender & Thompson, 2016)

Figure 12 represents what a game loop looks like when using the graph grammars. As you can see, an edge connects each node; some edges have barriers or items that the player must have previously to access the next node, such as a key to unlock a door. How this result is produced, the designer must first construct the rules he would like to have, such as where the start goal and end goal are. The designer must next create specific tasks that the player can perform, such as getting to a particular waypoint or picking up an item. The charges must later be defined, meaning when a player has completed such an event, the tasks are linked to some sort of end node that will complete that task. Then events such as keys and unlocking doors are added, as shown in figure 8; the only way to achieve the level is by retrieving the key, getting to the door, and unlocking the door (Lavender & Thompson, 2016).

By partitioning the generation into multiple different stages, the production and time cost becomes a lot less and provides a much more straightforward approach to developing a procedural generation algorithm. Having such an ability to simply tweak or change a rule entirely without altering the algorithm ultimately accelerates the workflow of designing a game (Sportelli, 2014).

Next, when thinking about level generation having the documents and graphs representing the level is all good and well; creating that mission within a level space in which a player can perform the tasks is another issue. There are three methods of doing this, the first being a transformation from mission to space, the second is the creation of a set of instructions or process that can be used to represent the game space, and the third is to build the level geometry to determine a better representation of the area in which will generate the mission itself (Sportelli, 2014).

1) Mission to Space Transformation:
   Once the detail of the space can be represented through a graph such as Figure 8, the level can further be translated using an automated graph layout algorithm to position the nodes of the graph; then, the output can then be sampled to a tile-map. The whole level can then be represented in a space in which the player can access (Middag, 2016).

2) Creation of an Instruction Set to Represent a Space:
   This requires taking the missions from the graph and forming them into a set of instructions that are then used to build the space that meets the predefined requirements. An example can be seen below:
   - Begin Rule (x1)
   - Add task (x15)
   - Add Boss (x1)
   - Define task (x15) (Locks & Keys, Treasure, Side Quests, Enemies, etc.)

These instructions can then be transformed into a tile-map or shape grammar, making this a pretty simple way of using GGs. However, the results of this are likely always to be linear and are less likely to have multiple paths leading to the same goal, which is suitable for games such as platformers or story-driven games (Middag, 2016).

3) Generating with Shape Grammars:
   In simple terms, shape grammars very similar to any other grammar uses a set of rewritten rules in which are used to manipulate and transform existing shapes (Middag, 2016).

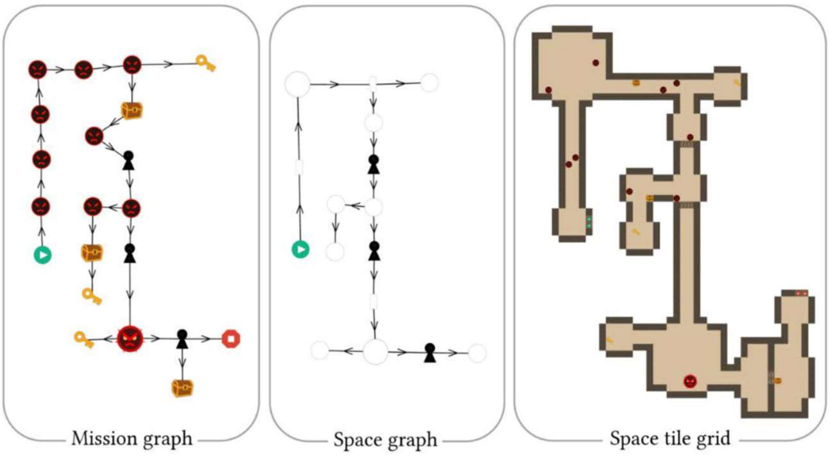


Figure 13 Step by step transformation from mission, to space, to tile grid (Middag, 2016)

When generating space with shape grammars first, each mission must be represented as a symbol; these can then be mapped to each Rule in the shape grammar. A good shape grammar algorithm is Dormans (2010). It encompasses a variance of difficulty using dynamic parameters in which can select rules with specific qualities based on the intended problem—applying probability to every task within the level itself (Middag, 2016).

2.2.4 Delaunay Triangulation

The Delaunay Triangulation (DT) of a set of points is one of the classical computational geometry problems. They were discovered in 1934 by the French mathematician Boris Nikolaevich Delaunay. DT was primarily used in the past for dividing a set of scattered points into uneven triangular grids. The algorithm has since been improved and refined for specific Procedurally Generated tasks. In 1985, Guibas and Stolfi proposed a Divide and Conquer algorithm for triangulations in two dimensions (Razafindrazaka, 2009).

Nowadays, DT is used for modeling terrain, or other objects are given a set of sample points. The DT gives a nice set of triangles to use as polygons in the model. In particular, the DT avoids narrow triangles (as they have large circumcircles compared to their area) (Razafindrazaka, 2009).

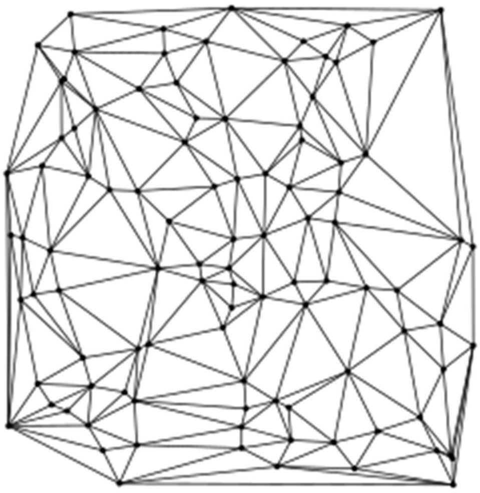


Figure 14A Delaunay triangulation of a random set of 100 points (Wikipedia)

Above (Figure 14) shows what a Delaunay Triangulation looks like for a random set of 100 points. The Delaunay Triangulation is assembled by introducing each point, one at a time, into an existing Delaunay triangulation which is then updated. The process is started by selecting three points to form a 'Super Triangle' that ultimately encompasses all data points triangulated.

When a new point P is introduced into the triangulation, we first find an existing triangle that encloses P and forms three new triangles by connecting P to each of its vertices. After the new point P has been inserted, the existing triangulation is updated to a Delaunay Triangulation using a swapping algorithm such as Lawsons. In this procedure, all the triangles adjacent to the edges opposite P are placed on a last-in, first-out stack (a maximum of three triangles are placed on the stack initially). Each triangle is then unstacked, one at a time and a check is made to determine if P lies within its circumcircle. If this is the case, the triangle containing P as a vertex and the adjacent triangle form a convex quadrilateral with the diagonal drawn in the wrong direction. It must be replaced by the alternative diagonal to preserve the structure of the Delaunay Triangulation. The swapping procedure will then replace two old triangles with two new triangles with no net gain in the total number of triangles.

Once the swap is completed, any triangles which are now opposite P are added to the stack (there is a maximum of two). The next triangle is then unstacked, and the whole process is repeated until the stack is empty, which results in a new Delaunay Triangulation containing the point P (Lee & Schachter, 1980).

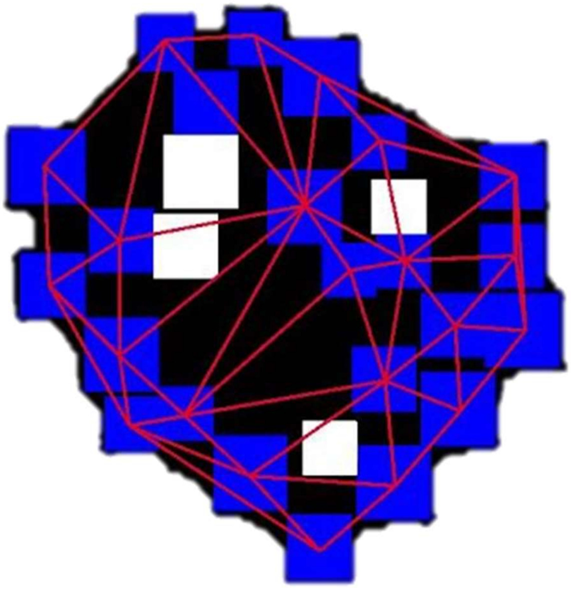


Figure 15Dungeon Representation Of Delaunay Triangulation

Above (Figure 15) shows how Delaunay Triangulation is used for dungeon maps in 2D. A video by Nathan Williams (Williams, 2014a) shows this in greater detail on how each room is generated and how they connect.

2.3 Summary

There are various and multiple aspects of the research that have inspired the plan of my prototype design. The choice of three algorithms that are not too similar such as original algorithms that have been enhanced or altered. The prototype focuses on map generation rather than content generation algorithms, such as the work done on Generative Grammars by Kane White on Dormans (White, 2019). The map generation may oppose more of a challenge than initially thought of. Converting content generation algorithms to a map generation algorithm could take a lot more time as less research has been done on the subject.

The reasoning behind the choices of algorithms I have made lies within the similar traits they have with each other. The research done on cellular automata by Wolfram (Wolfram, 2006). Has explored the different aspects of generation for dungeons such as corridors, rooms, start and end positions, much like the research conducted by Nathan Williams for Delaunay Triangulation looking for the same map generated results (Williams, 2014b). With such elements, this will allow for comparative testing to then understand which algorithm is best suited for dungeon generation. Perlin noise is an algorithm more associated with terrain generation than a dungeon. With the research conducted by Dave Mount, I can develop an algorithm that'll provide the elements needed to be viable to compare to the other two (Mount, 2018).

I plan to evaluate each algorithm's viability by providing a start and end position with a similar distance to the player on each level. To then run the shortest distance pathfinder such as A* to test if the algorithm can provide a map accessible to the player and efficient to navigate, answer the third research question (Chapter 1, Section 1.3).

3 Requirements and Analysis

3.1 Requirements

This project aims to provide three different procedurally generated maps using different PCG algorithms for the player to explore. It will investigate and compare each algorithm against each other, such as ease of use, exploration, and then deciding which algorithm is the best to use for dungeon generation. The map can also contain some elements the player can interact with though it isn’t a must.

The following requirements include those identified during the research phase, even if they are not going to be implemented within the project.

ID	Requirement	Category	Subcategory
1a	Menu Screen (Choice of Algorithm)	Game Design	System Design
1b	Making map generation customizable	Game Design	System Design
1c	Localisation of Game	Game Design	System Design
2a	Randomly Generated Map (Cellular Automata)	Level Generation	Game Design
2b	Randomly Generated Map (Perlin Noise)	Level Generation	Game Design
2c	Randomly Generated Map (Delaunay Triangulation)	Level Generation	Game Design
3a	Quit Functionality	Game Design	System Design
3b	Control Scheme Information	Game Design	System Design
3e	Customisable Key Bindings	Game Design	System Design
4a	Single Player	Game Systems	Entity Behaviour
5a	2D game type	Game Design	World Design
6a	Enemies	Game systems	Entity Behaviour
6b	AI Pathfinding for Enemies	Game systems	Entity Behaviour
6c	Procedural Animation	Game systems	Entity Behaviour
7a	Procedural Content Generation (Keys, Vegetation, etc.)	Game bits	Game items
8a	Interaction	Game bits	Behaviour
9a	Music	Game Music	Sound Effects
10a	PC Build	N/A	N/A
10b	Web Build	N/A	N/A
10c	Mobile Build	N/A	N/A

Table 3 Potential Requirements

Using the MoSCoW method, these requirements have been filtered down to define the scope of the project:

Must-Have

- ID 1a: Have a menu screen that allows testing with any individuals simple and easy to allow for a good evaluation.
- ID 2a: A procedurally generated map using a cellular automata algorithm, for the player to explore and venture around.
- ID 2b: A procedurally generated map using a Perlin noise algorithm, for the player to explore and venture around.
- ID 2c: A procedurally generated map using a Delaunay Triangulation algorithm, for the player to explore and venture around.
- ID 3a: Functionality to allow the player to quit to the menu screen to be able to switch and test different algorithms.
- ID 4a: Ensure the player can control a character within the scene.
- ID 5a: Generate a playable game – 2D chosen due to the simpler nature of 2D map generation.
- ID 10a: PC build is necessary for the game to be tested and evaluated.

Should Have:

- ID 1b: Algorithm Customisation such as map size and different toggles would be useful but not essential. As it could allow for better comparisons when it comes to evaluation.
- ID 3b: Control Scheme Information would provide good information on how to control the character and other information if the player has no previous knowledge on common game controls.

Could Have:

- ID 6a: Enemies would be a good addition to make it more of a game rather than a showcase of algorithms.
- ID 6b: AI Pathfinding would be a cool addition but not mandatory and goes a bit off scope from the aim of the project.
- ID 10b: Web build would be good to allow people to test and give opinions for future research on the topic.

Will not Have:

- ID 1c: Localising the game to other languages will increase the audience however, I will be unable to do this myself as usually this task is carried out by specialized teams and is way beyond the scope of the project.
- ID 3e: Customizable key binding would be an excellent addition but not necessary and will increase the amount of time on unnecessary parts.
- ID 6c: Procedural Animation is not suitable for most 2D games, so it will not be included in this project.

- ID 8a: Interaction is more of a game element that is not necessary for this project type.
- ID 9a: Music and SFX are not needed for this project, as time can be spent elsewhere more directed to the project scope.
- ID 10c: Mobile build is unnecessary and would require many more controls and game mechanics, which would take the time that could be directed more at the scope of the project.

3.2 Development Methodology & Tools

I will be using a Waterfall development methodology. There are clear stages in which I will aim to complete for the project to follow a good workflow. These stages will include:

- **Requirements** - where we analyse business needs and document what software needs to do.
- **Design** - where we choose the technology, create diagrams, and plan software architecture.
- **Coding** - where we figure out how to solve problems and write code.
- **Testing** - where we make sure the code does what it supposed to do without breaking anything.
- **Operations** - where we deploy the code to a production environment and provide support.

Testing pans: The majority of testing will take the format of manually running scenarios to ensure each algorithm is producing the correct results until I can allow other users to test the game. There are several areas of both functional and non-functional testing I plan to include:

- Functionality Testing: Will confirm whether the end product works following the specifications. Will aim to hunt for generic problems within the game or its graphics & User interface.
- Ad Hoc Testing: Ad hoc testing, often referred to as ''general testing'' is a less structured way of testing, and it is randomly done on any section of the gaming application. Specifically, there are two distinct types of ad hoc testing. This kind of testing works on the technique called "error guessing" and requires no documentation or process or planning to be followed.

- Compatibility Testing: Compatibility testing aims to detect any defects in the functionality and show if the final product meets the software's essential requirements, hardware, and graphics. It is better to keep the game users happy, after all.

For projects that contain such procedurally generated algorithms, I would usually opt in to use Unit Tests. However, I am using Unity, Where I do not have any experience implementing unit tests in this environment and consider time constraints. I will conduct frequent manual tests to ensure each algorithm is working accordingly.

The main tools for developing the game I will be using consist of Unity, a widely used game engine for games development used for indie and AAA titles, and Mono Develop, the default IDE for programming in C#.

Version control such as GitHub will keep track of progress and ensure the project is backed up.

4 Design

The project structure, in simple terms, is made up of a menu screen with a selection of three algorithms to choose from. Once selected, the algorithm will then procedurally generate a map of the chosen algorithm, and then the player can explore the surrounding area.

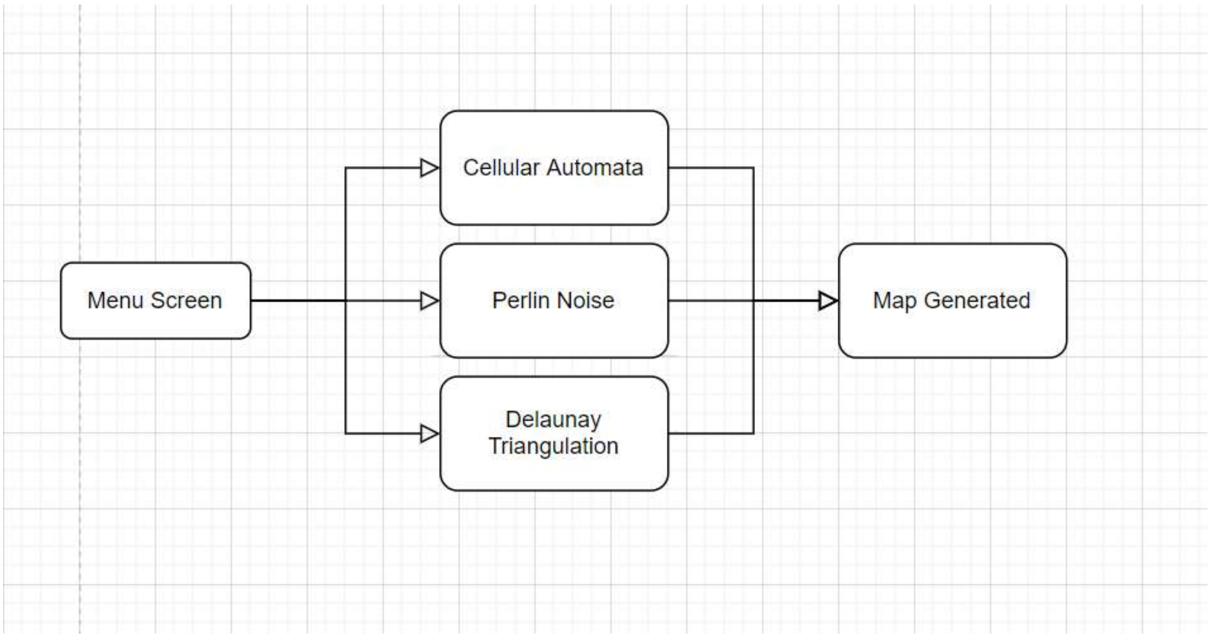


Figure 16 Overview of Proposed Project Structure

This will select the desired map generation algorithm and produce a playable map for the character to explore and make any judgments/opinions on the map generated.

4.1 Base Game Design

The game type will be a top-down 2D rogue-like dungeon crawler. With elements such as:

- The playable character: Which will be able to move 4 directions, no animation or effects will be added due to time constraints.
- A floor tile: The floor tile will be the tile in which the player can move around on and will allow for a base, bottom map to be generated of floor tiles.
- A wall tile: The wall tile will be the tiles in which the player can collide with and will be the most important part in terms off creating the caves and corridors.
- A key: The key will be a collectable in which the player can use to escape the cave generated map.
- Enemies: Enemies that will be in the form of officers that will follow the player around the game map.

- Obstacles: Obstacles such as doors, holes, and nets to push the player in different directions of the map.

- Collectables: Collectables such as health, speed, and invisibility to give the player advantages to venture around the map.

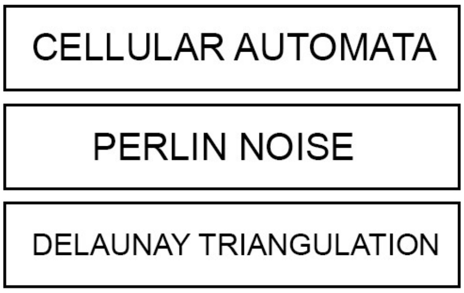


Figure 17 Wireframe of Menu UI

While enemies and collectibles such as keys and obstacles provide a more engaging and fun game to play, I opted only to include these features if I have time. As they are not essential features to the project's actual scope and only provide a better experience for the real game itself.

As we can see in Figure 17. This is a very basic wireframe of the menu screen being developed as I won't have a lot of time to spend on elements such as menu and UI/UX.

The game art itself is based on a previous game I have produced, as it allowed for less time to be allocated to elements that don't affect the project scope at all. The art style is a 2D Prisoner of war setting.

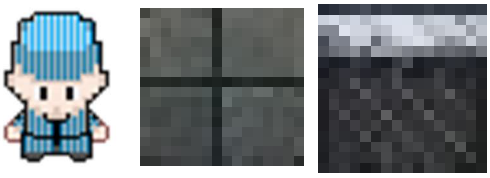


Figure 18 Game Art

Shown in Figure 18. Is the character, floor, and wall tile in the theme of a 2D Prisoner of War setting. All assets were designed by me so no copyright infringements have been committed.

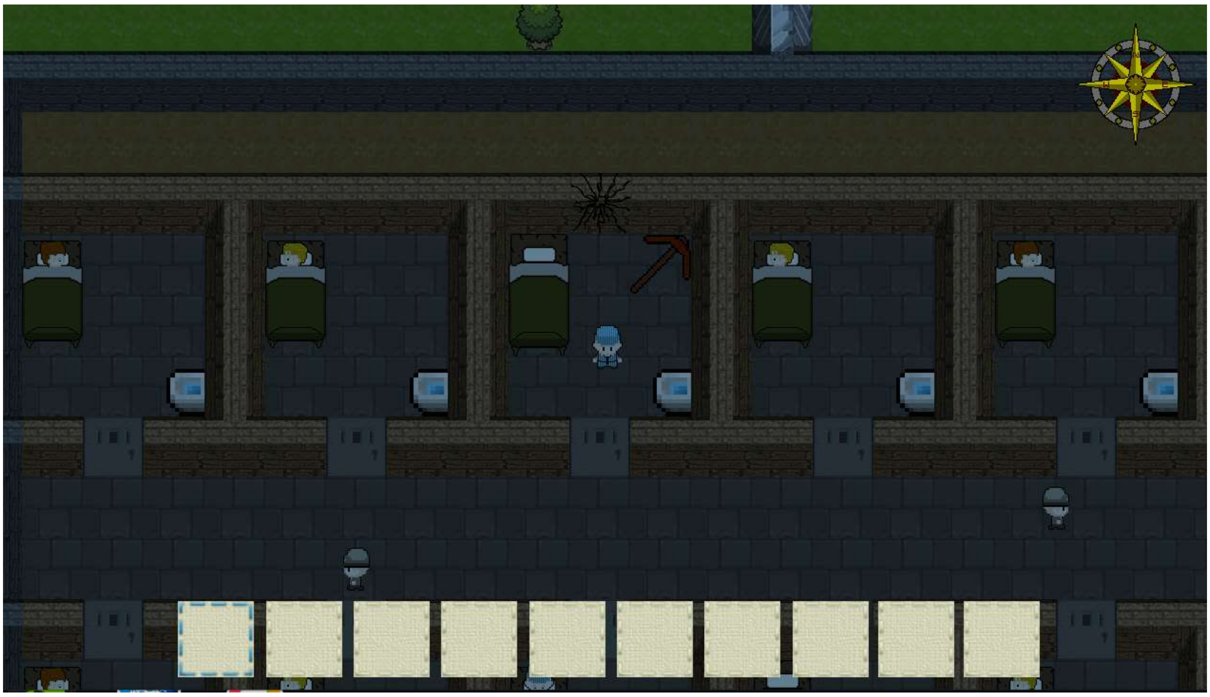


Figure 19 Prisoner of Way by Arran Smedley

Shown in Figure 19. Is the game I developed in late 2018 called prisoner of war where all the game art for this project will be inspired from.

5 Implementation

5.1 Proof of Concept

To understand how this project's aims could be achieved, I had to determine the feasibility of the planned application. The goal was to develop a simple prototype that demonstrated three procedural generation algorithms within the Unity Game Engine. The prototype should give the user a choice of three algorithms in which will be chosen, and a map will then be generated for the player to explore. As I had previous experience working in Unity and Visual Studio, I did not have to learn any new programming languages or how the engine works.

5.2 Cellular Automata

As discussed in Chapter 1, the first algorithm developed within Unity (C#) was cellular automata. I went with how Wolfram described each stage of Cellular Automata as mentioned previously within the literature review and built on that to produce a map generator (Wolfram, 2006).

The first thing when implementing this algorithm was to understand how to implement the cell states. The cell states are represented as a 0 or 1 and create and fill it based on their condition.

Cellular automata, as previously described by Wolfram, lives by a certain number of rules in which is applied to each cell in every step of the simulation:

1. If a living cell has less than two live neighbours, it dies.
2. If a living cell has two or three live neighbours, it stays alive.
3. If a living cell has more than three live neighbours, it dies.
4. If a dead cell has exactly three living neighbours, it becomes alive.

Though very simple can be altered to retrieve fascinating and strange outcomes to the map generation. These rules were used in the game to create interesting cave patterns.

The cellular grid is represented as an array of *ints*(Integers) in which takes the width and height from public *ints* decided on the unity UI.

```
private int[,] terrainMap;
int width;
int height;
```

Each one of these array positions represents one of the ‘cells’ in our cellular grid. Next is the initialisation to begin building these cellular maps.

Each cell will have the same random chance of being made alive by using Unity’s random system in which iniChance will decide the possibility of it surviving or being dead.

```
public void initPos()
    {
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                terrainMap[x, y] = Random.Range(1, 101) < iniChance ? 1 :0;
            }

        }

    }
```

If I were to run the code at this point, as seen in Figure 15, we would see many random cells, alive or dead, which would not be coordinated whatsoever. Next, I had to populate and grow them into caves.

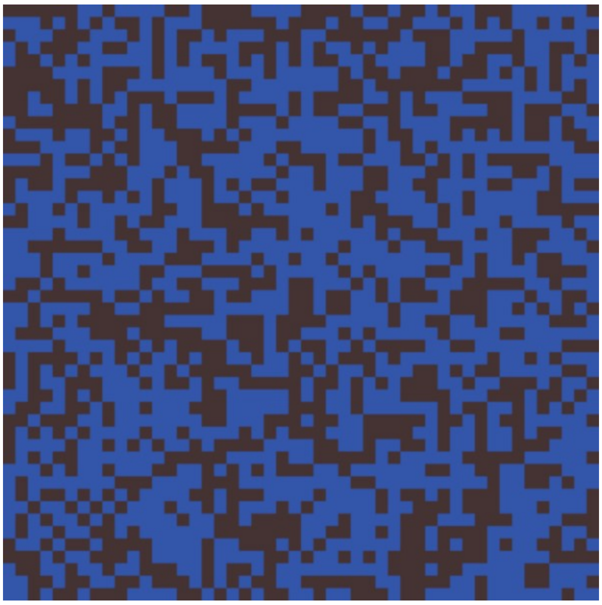


Figure 20 Random Cells

Referring to Conway's Game of Life, each time the simulation went ahead by one step, every cell would check Life's rules to see if it would change to being dead or alive. I decided to use the same regulations and idea to build the caves – in which I wrote a function that loops over every cell in the grid and applies some basic rules to decide whether it lives or dies.

We consider each cell in the grid in turn and count how many of its neighbours are alive and dead. I have made these calculations in a method called genTilePos in which is displayed below.

```
public int[,] genTilePos(int[,] oldMap)
    {
        int[,] newMap = new int[width,height];
        int neighb;
        BoundsInt myB = new BoundsInt(-1, -1, 0, 3, 3, 1);


        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                neighb = 0;
                foreach (var b in myB.allPositionsWithin)
                {
                    if (b.x == 0 && b.y == 0) continue;
                    if (x+b.x >= 0 && x+b.x < width && y+b.y >= 0 && y+b.y < height)
                    {
                        neighb += oldMap[x + b.x, y + b.y];
                    }
                    else
                    {
                        neighb++;
                    }
                }

                if (oldMap[x,y] == 1)
                {
                    if (neighb < deathLimit) newMap[x, y] = 0;

                        else
                        {
                            newMap[x, y] = 1;

                        }
                }

                if (oldMap[x,y] == 0)
                {
                    if (neighb > birthLimit) newMap[x, y] = 1;

                else
                {
                    newMap[x, y] = 0;
                }
                }

            }

        }
        return newMap;
    }
```

The idea of this function is that we want to look at all the neighbouring cells around a particular tile in the graph and decide whether the cell is alive or dead. Returning a new map of values to be simulated.

Finally, the central part of this cellular automation is the simulation, so I gave a doSim function to make a new grid in which updated cell values are stored into. To understand this part, remember that to calculate the unique value of a cell in the grid, we need to look at its eight neighbors, represented in the previously mentioned genTileMap function.

```
public void doSim(int nu)
    {
        clearMap(false);
        width = tmpSize.x;
        height = tmpSize.y;

        if (terrainMap==null)
            {
            terrainMap = new int[width, height];
            initPos();
            }


        for (int i = 0; i < nu; i++)
        {
            terrainMap = genTilePos(terrainMap);
        }

        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                if (terrainMap[x, y] == 1)
                    topMap.SetTile(new Vector3Int(-x + width / 2,
                    -y + height / 2, 0), topTile);
                    botMap.SetTile(new Vector3Int(-x + width / 2,
                    -y + height / 2, 0), botTile);
            }
        }


    }
```

Some tweaking and tuning were added to use the full power of the Unity engine by adding certain components to make the cellular automation more manageable these components consisted of:

- iniChance sets how dense the initial grid is with living cells.
- deathLimit is the lower neighbor limit at which cells start dying.
- numR is the upper neighbor limit at which cells start dying.
- birthLimit is the number of neighbors that cause a dead cell to become alive.

These variables are interchangeable and can change the simulation results entirely, though they can give excellent results for dungeon-generated maps that can accommodate the type of game being created.

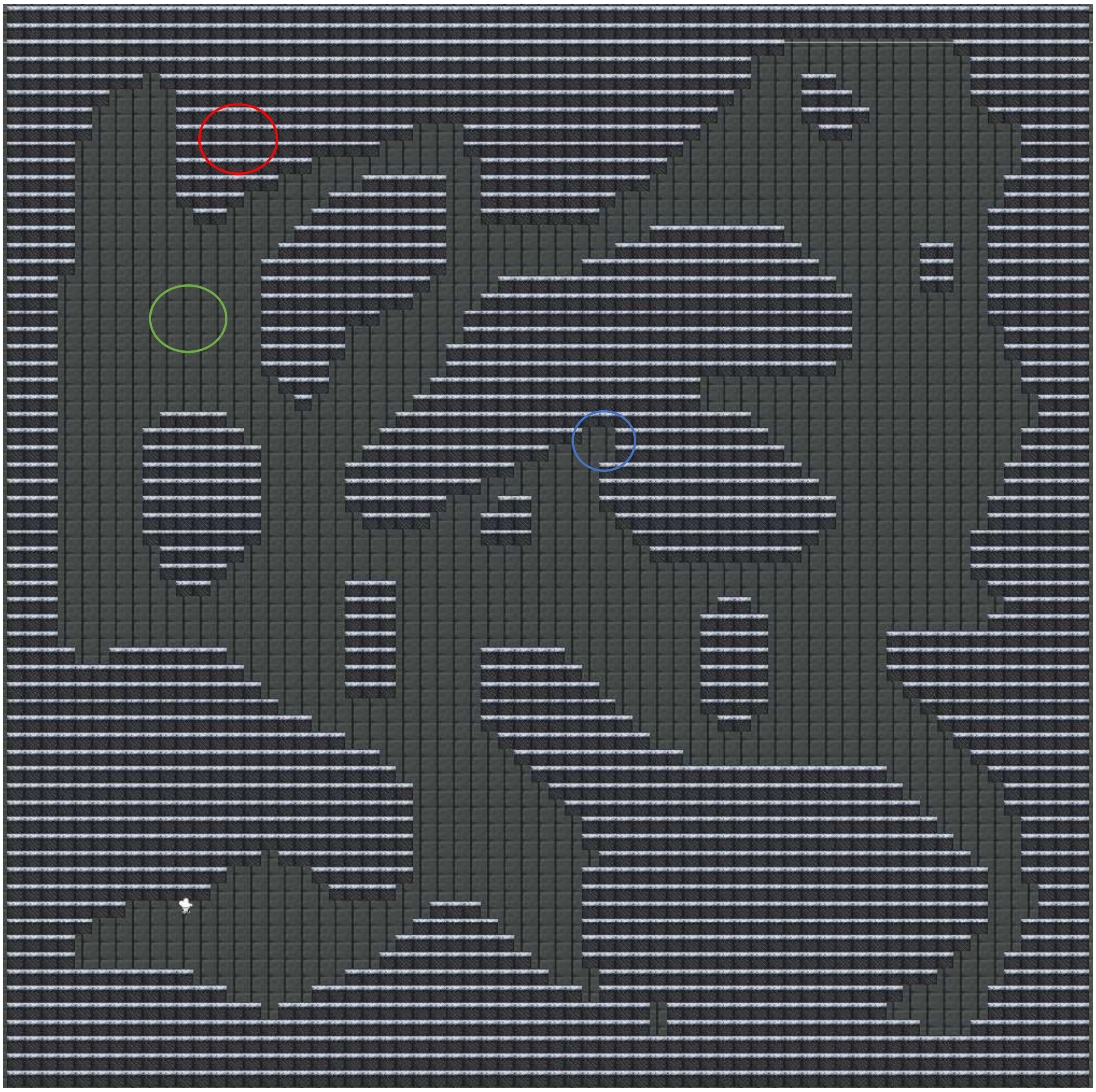


Figure 17 Cellular Automation

With everything implemented, shown in Figure 22. This is a map where the Initial Chance has been set to 30, the Birth and Death Limit have been set to 3 and the Number of Repetitions has been set to 10 on a 64x64 sized map. With these values, there is a good base of a dungeon crawler map there and ensures there are no areas where the player cannot access.

As we can see going back to the four rules, we installed into the cells by Wolfram we can see that they are following them nicely. With a good border around the map (Wolfram, 2006).

- Red: Rule 1
- Green: Rule 2
- Blue: Rule 4 & 3

With values such as the Initial Chance or Limits changed, different and exciting maps can be formed.

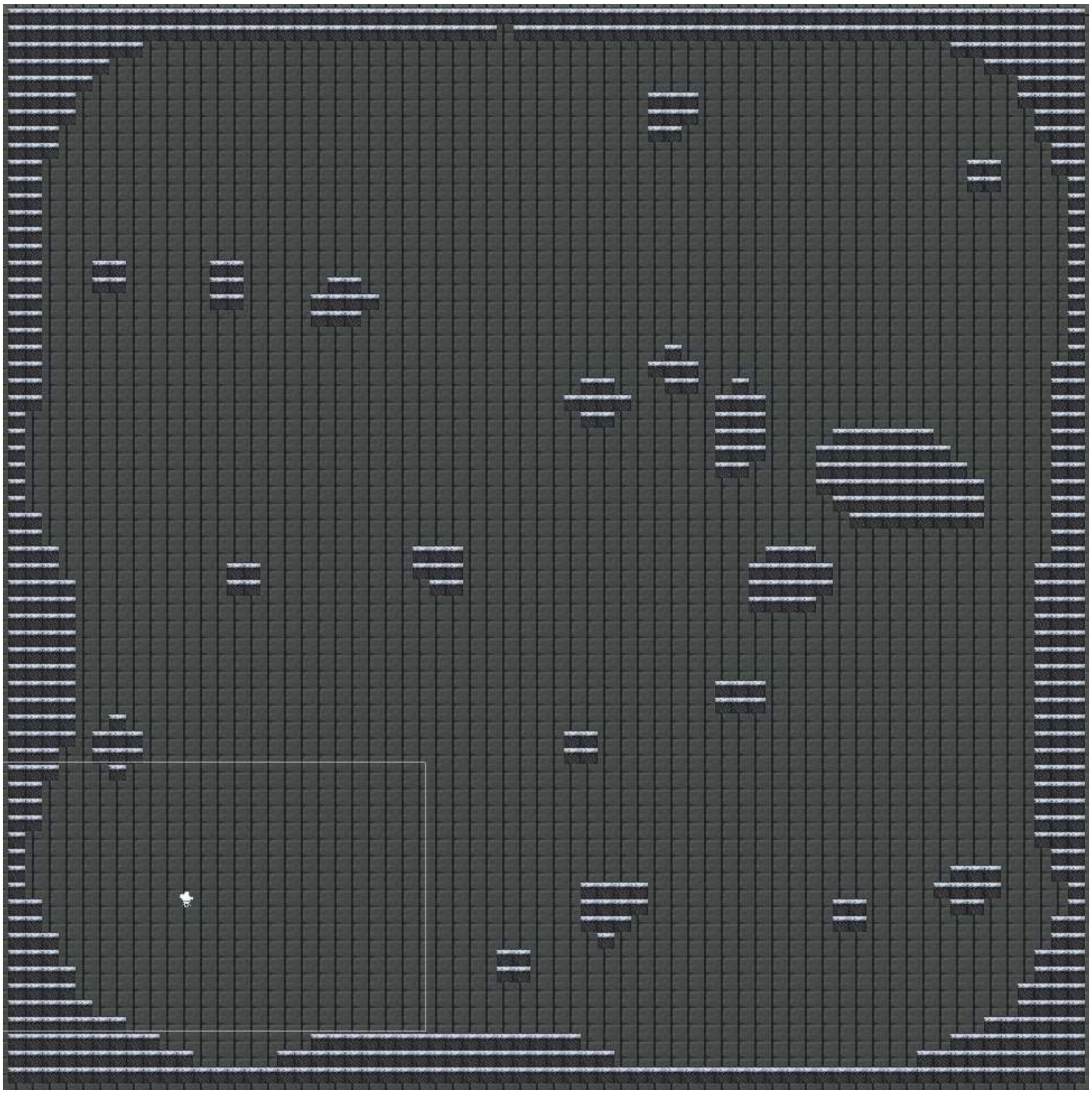


Figure 18 Cellular Automation (Initial Chance Altered)

Shown in Figure 23. Is when the Initial Chance has been altered to 20 rather than the 30 I had before. As we can see, we get an interesting map design that could represent a room rather than a full-on 2D-Dungeon Map.

When changing to limits, we start to get into more complicated map designs that become impossible to explore.

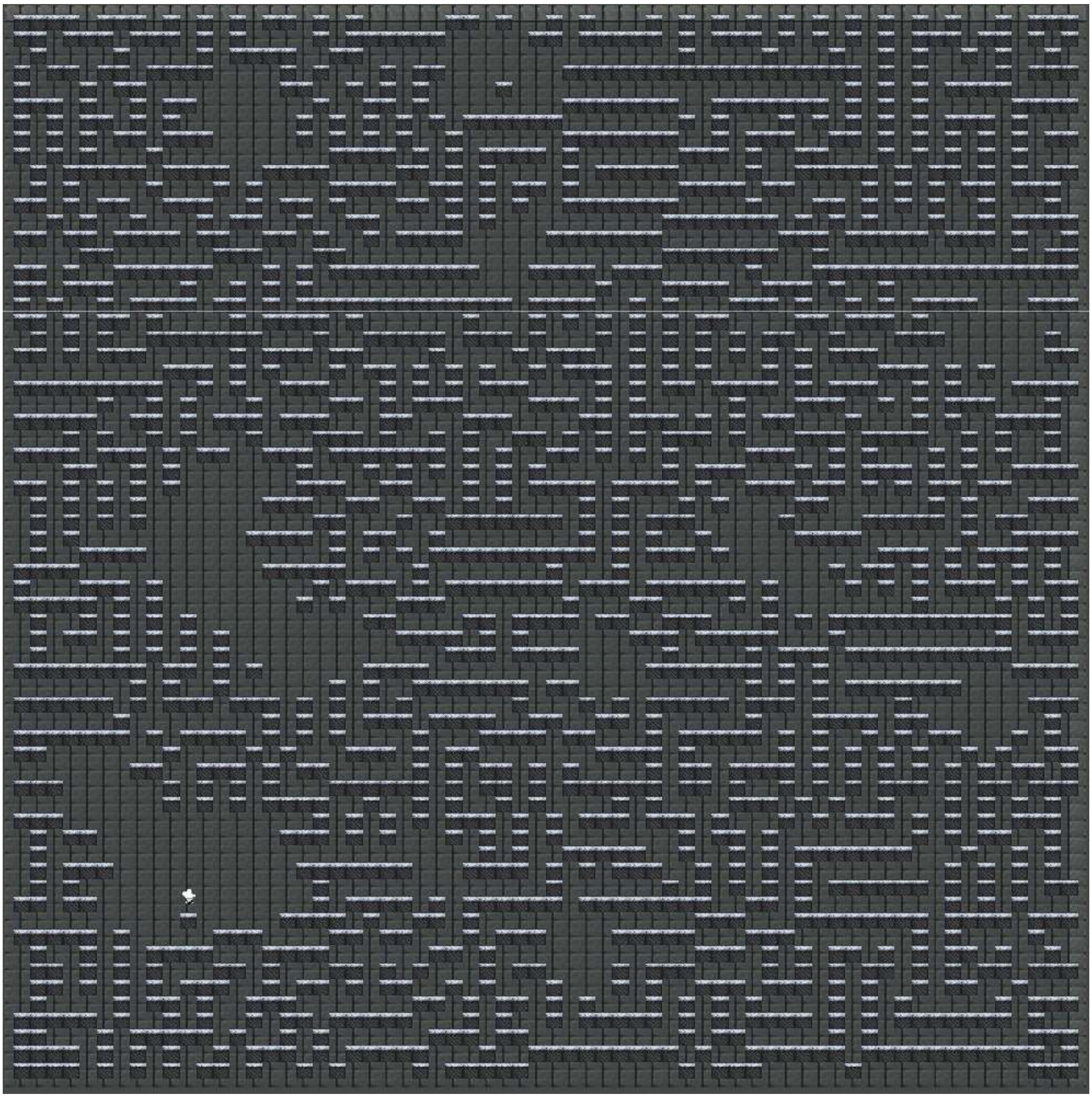


Figure 19 Cellular Automation (Altered Limits)

As we can see in Figure 24. With the altered limits, we get exciting but not very traversable map design. This would be a helpful map for making a maze game or some sort of puzzle game, but I do not believe it is valid enough for dungeon map designs.

With all the variables adjusted and tested, I believe the initial values shown in Figure 16 produce the best 2D-Dungeon Crawler type map. I will be taking these values into the User Evaluation.

5.3 Perlin Noise

The second algorithm done within a different scene of the Unity project with C# consists of the Perlin noise algorithm. Perlin noise, as mentioned before, is a procedural generation technique not just used within the game's development environment but also for movies, entertainment, etc. To understand the algorithm and how I have implemented it, we must first understand the critical components of how Perlin noise works. As not a lot has been done on Perlin noise for specific 2D-Dungeon Crawler Map generation, I decided to go with a modified approach to how Mount described Perlin Noise with the help of some of Unity's built-in function (Mount, 2018).

Embedded within the Unity engine is a useful function that helps with the Perlin noise generation, which has done many calculations for me.

```
Mathf.PerlinNoise(float x, float y);
```

A float value is returned from this function that can be used to map what looks like pre-generated Perlin noise to an x and y coordinate that is chosen.

```
var perlin = Mathf.PerlinNoise(i / 10, k / 10);
```

This is how I implemented this function. By playing around with the input values, different outcomes of the Perlin noise can be produced. Below are some examples of Perlin noise generated in Unity using other x and y inputs.

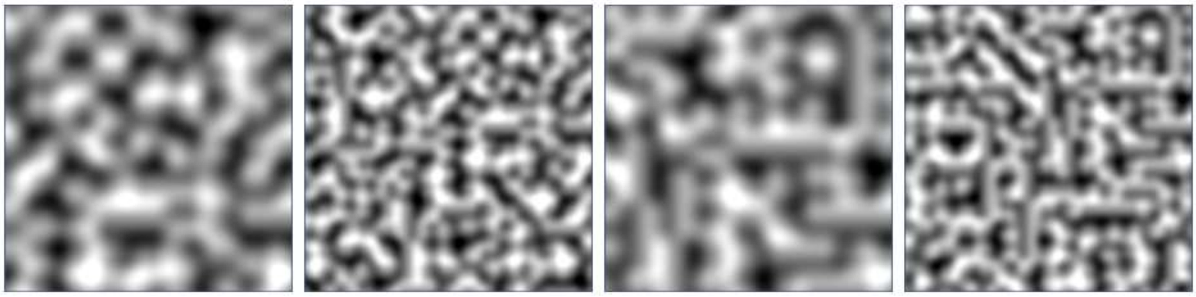


Figure 20 Different Perlin Noise outputs

By changing such values such as the offset and scale within the bounds defined, Perlin noise is an effortless way of generating the desired game map or terrain that is wanted. Focusing on a pattern within the Perlin noise itself can create a sound dungeon map generation as I have done. Below is an excellent visual representation of what I am describing.

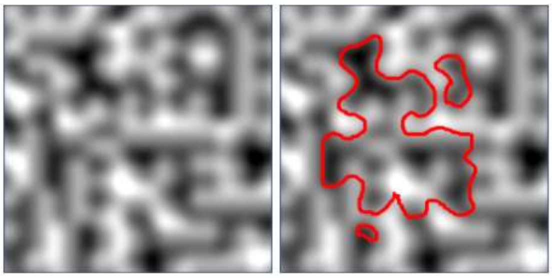


Figure 21 Perlin Noise Map representation

To understand the actual PerlinNoise function itself, I delved deeper into the Unity documents and how it was created. Below is how the noise itself is calculated using all code developed and done within the Unity Engine.

```
void CalcNoise()
    {
        // For each pixel in the texture...
        float y = 0.0F;

        while (y < noiseTex.height)
        {
            float x = 0.0F;
            while (x < noiseTex.width)
            {
                float xCoord = xOrg + x / noiseTex.width * scale;
                float yCoord = yOrg + y / noiseTex.height * scale;
                float sample = Mathf.PerlinNoise(xCoord, yCoord);
                pix[(int)y * noiseTex.width + (int)x] = new Color(sample, sample, sample);
                x++;
            }
            y++;
        }

        // Copy the pixel data to the texture and load it into the GPU.
        noiseTex.SetPixels(pix);
        noiseTex.Apply();
    }
```

This calculates that any plane point can be sampled by passing the appropriate X and Y coordinates. The exact coordinates will always return the same sample value. Still, the plane is virtually infinite, so it is easy to avoid repetition by choosing a random area to sample from.

Bellow is the Perlin noise generator class developed by myself; this class creates the map itself using Unity's Perlin Noise function as the central part:

```
public class Generator : MonoBehaviour
{
    public GameObject dirtPrefab;
    private GameObject C;
    private float maxX = 320;
    private float maxY = 320;
    private int seed;
    void Start()
    {
        Regenerate();

    }

    private void Regenerate()
    {
        float width = dirtPrefab.transform.lossyScale.x / 5;
        float height = dirtPrefab.transform.lossyScale.y / 5;
        for (float i = 0; i < maxX; i++)
        {
            for (float k = 0; k < maxY; k++)
            {
                var perlin = Mathf.PerlinNoise(i / 10, k / 10);
                if (perlin > .5f)
                {
                    C = (GameObject)Instantiate(dirtPrefab, new Vector3(i * width, k * height, 2), Quaternion.identity);
                    SpriteRenderer Sr1 = C.GetComponent<SpriteRenderer>();
                    Sr1.color = new Color(0, perlin, 0);
                }
            }
        }
    }
}
```

Most of the code is positional data of what part and where the Perlin noise will be within the game map and gives the noise a value such as the dirtPrefab game object to create the dungeons and rooms.

From using this Perlin noise algorithm, I managed to get similar simulations to what previously had been shown for cellular automata:

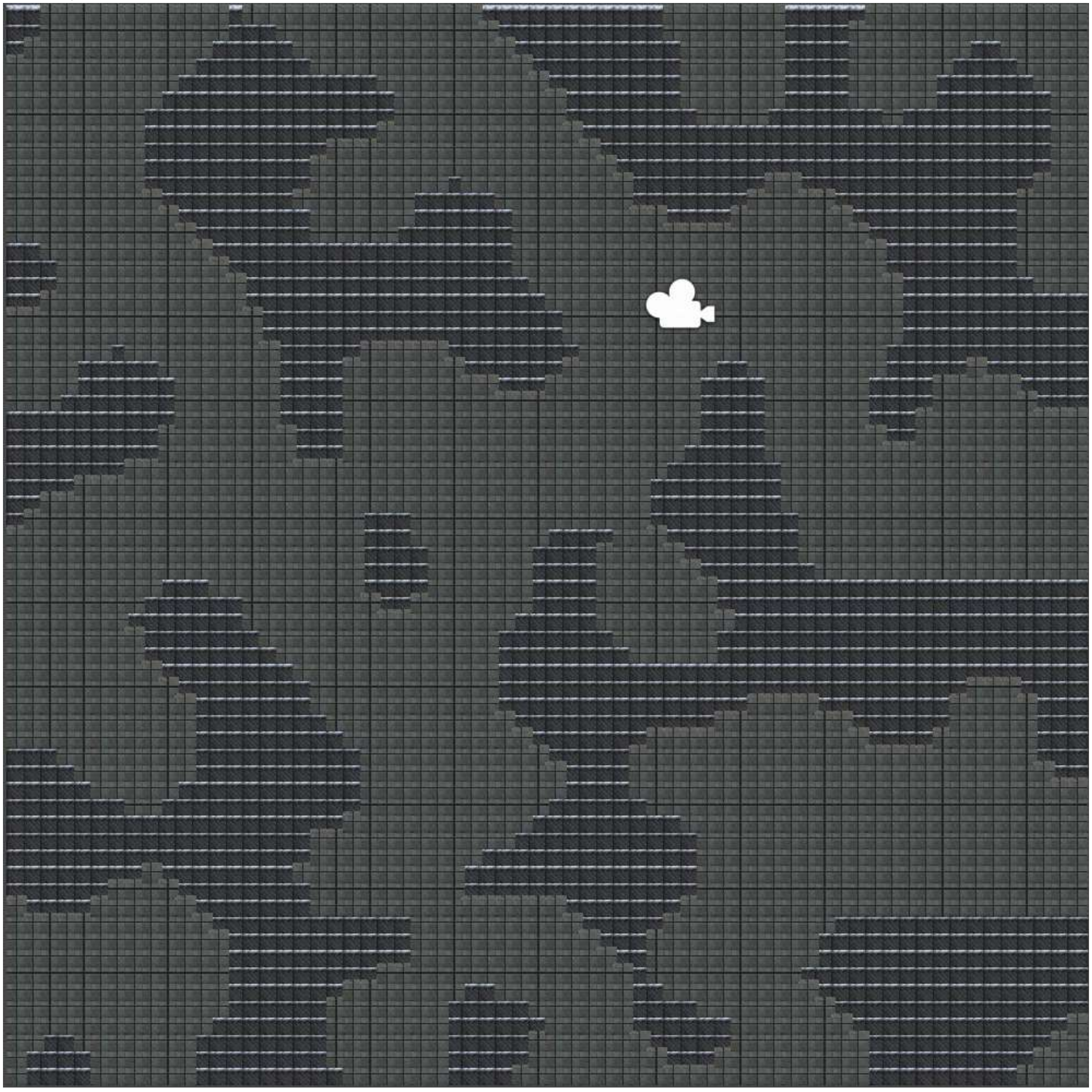


Figure 22 Perlin Noise Map Generation Outcome

As shown in Figure 27. We get an exciting map design. This map, like previously in Cellular Automata, is also 64x64, but as we can see, there are many differences. To briefly cover what I mean, which will be in more detail in the evaluation, many areas cannot be reached, making parts of the map utterly inaccessible.

I began to experiment further and decided to increase the map size to 256x256 to see if there were still many areas in which the player cannot access.

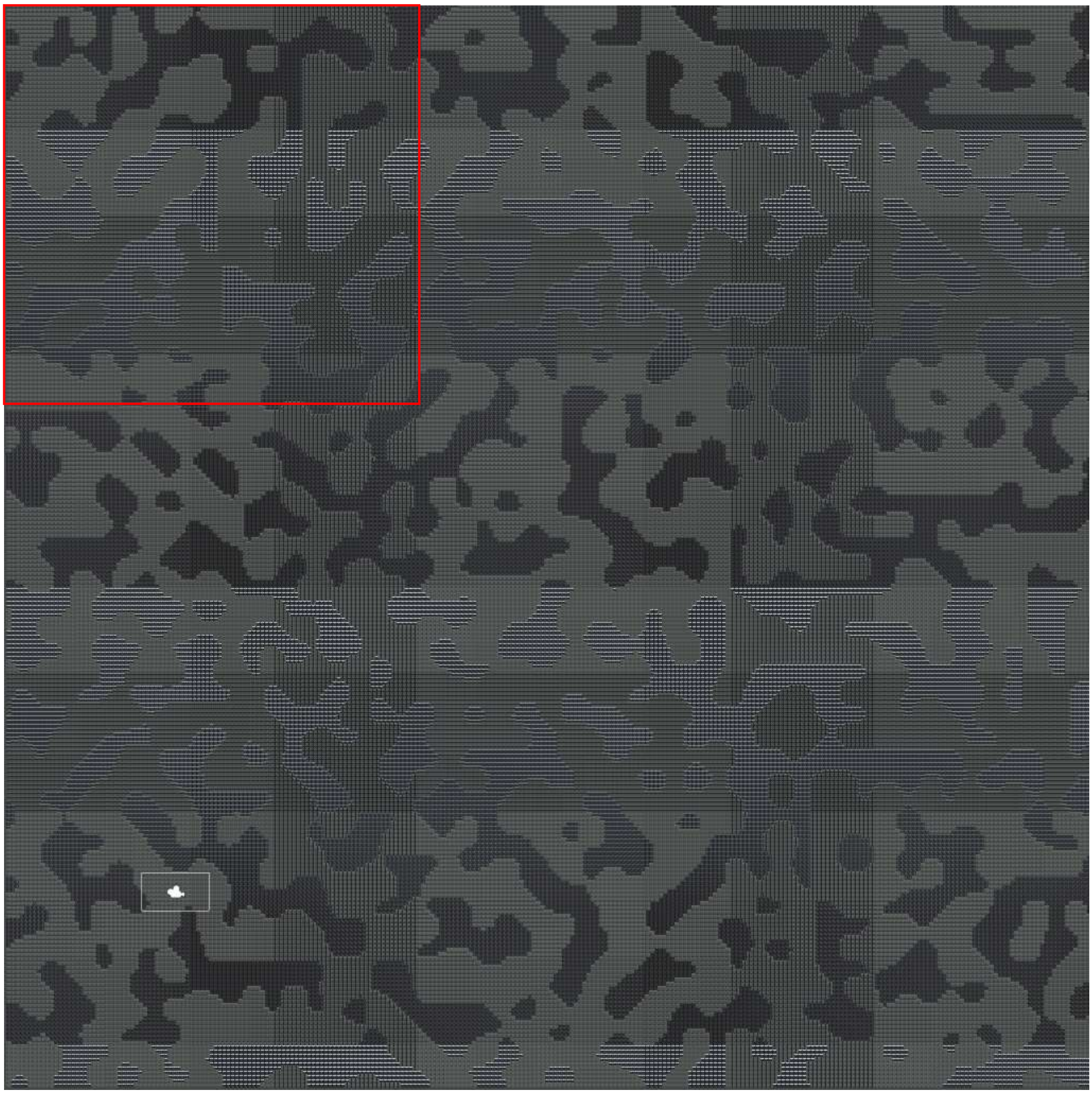


Figure 23 Perlin Noise Generation (Bigger Map)

As shown in Figure 28. The areas before that we couldn’t access are now accessible, providing a more complete user experience. Shown in the red is what a Perlin Noise map could look like when initially generating it. Going back to the start of the chapter if you compared that red square to the initial Perlin Noise images they are very similar in pattern.

I decided to go with the 256x256 sized map for the user evaluation. It provided a better experience than the 64x64 allowing the player to access those caverns through different corridors and rooms.

5.4 Delaunay Triangulation

The third algorithm done within a different scene of the Unity project with C# consists of the Delaunay Triangulation algorithm. As previously mentioned in Chapter 2, Delaunay Triangulation is a procedural generation technique of a set of points that is one of the classical computational geometry problems. The way in that I have developed this algorithm into a 2D Dungeon Crawler type game is as follows.

So the central concept that I followed consisted of the following:

1. Create cells and distribute them randomly.
2. Separate those cells to make sure there's no overlapping cells.
3. Filter the rooms based on size and then triangulate between the ones kept.
4. Create a spanning tree between them to make sure we can reach all cells in an out generator.
5. Strict the spanning-tree into horizontal and vertical lines to make paths.
6. Visualise the results.

By going through this concept, I was able to create a good Delaunay Triangulation Algorithm. It was first developing the actual cells and distributing them randomly.

```
void CreateCells()
        {
            RandomFromDistribution.ConfidenceLevel_e conf_level =
            RandomFromDistribution.ConfidenceLevel_e._80;

            int numberOfCells = levelStats.numberOfCells;
            float roomCircleRadius = levelStats.roomCircleRadius;
            percFromGraphToPaths = levelStats.percFromGraphToPaths;
            mainRoomMeanCutoff = levelStats.mainRoomCutoff;

            float cellMinWidth = levelStats.cellMinWidth;
            float cellMaxWidth = levelStats.cellMaxWidth;
            float cellMinHeight = levelStats.cellMinHeight;
            float cellMaxHeight = levelStats.cellMaxHeight;

            for (int i = 0; i < numberOfCells; i++)
            {
                float minWidthScalar = cellMinWidth;
                float maxWidthScalar = cellMaxWidth;
                float minHeightScalar = cellMinHeight;
                float maxHeightScalar = cellMaxHeight;

                GeneratorCell cell = new GeneratorCell();
                cell.width = Mathf.RoundToInt
                (RandomFromDistribution.RandomRangeNormalDistribution
                (minWidthScalar, maxWidthScalar, conf_level));
                cell.height = Mathf.RoundToInt
                (RandomFromDistribution.RandomRangeNormalDistribution
                (minHeightScalar, maxHeightScalar, conf_level));


                Vector2 pos = GetRandomPointInCirlce(roomCircleRadius);
                cell.x = Mathf.RoundToInt(pos.x);
                cell.y = Mathf.RoundToInt(pos.y);
                cell.index = i;
                cells.Add(cell);
                widthAvg += cell.width;
                heightAvg += cell.height;
            }

            widthAvg /= cells.Count;
            heightAvg /= cells.Count;
        }
```

Next, I needed to separate those cells to make sure there are no overlapping cells.

```
void SeparateCells()
        {
            bool cellCollision = true;
            int loop = 0;
            while(cellCollision)
            {
                loop++;
                cellCollision = false;
                if(debug)
                {
       //                     Debug.Log("Loop " + loop);
                }

                for (int i = 0; i < cells.Count; i++)
                {
                    GeneratorCell c = cells[i];

                    for (int j = i + 1; j < cells.Count; j++)
                    {
                        GeneratorCell cb = cells[j];
                        if(c.CollidesWith(cb))
                        {
                            cellCollision = true;

                            int cb_x = Mathf.RoundToInt((c.x + c.width) - cb.x);
                            int c_x = Mathf.RoundToInt((cb.x + cb.width) - c.x);

                            int cb_y = Mathf.RoundToInt((c.y + c.height) - cb.y);
                            int c_y = Mathf.RoundToInt((cb.y + cb.height) - c.y);

                            if (c_x < cb_x)
                            {
                                if (c_x < c_y)
                                {
                                    c.Shift(c_x, 0);
                                }
                                else
                                {
                                    c.Shift(0, c_y);
                                }
                            }
                            else
                            {
                                if (cb_x < cb_y)
                                {
                                    cb.Shift(cb_x, 0);
                                }
                                else
                                {
                                    cb.Shift(0, cb_y);
                                }
                            }
                        }
                    }
                }
            }
        }

```

Furthermore, I needed to Filter the rooms based on size and then triangulate between the ones kept.

```
void PickMainRooms()
{
    foreach (GeneratorCell c in cells)
    {
        if(c.width * mainRoomMeanCutoff < widthAvg ||
        c.height * mainRoomMeanCutoff < heightAvg)
        {
            c.isMainRoom = false;
        }
    }
}

void Triangulate()
{
    List<Vector2> points = new List<Vector2>();
    List<uint> colors = new List<uint>();

    Vector2 min = Vector2.positiveInfinity;
    Vector2 max = Vector2.zero;

    foreach (GeneratorCell c in  cells)
    {
        if (c.isMainRoom)
        {
            colors.Add(0);
            points.Add(new Vector2(c.x + (c.width / 2), c.y +
            (c.height / 2)));
            min.x = Mathf.Min(c.x, min.x);
            min.y = Mathf.Min(c.y, min.y);

            max.x = Mathf.Max(c.x, max.x);
            max.y = Mathf.Max(c.y, max.y);
        }
    }

    Voronoi v = new Voronoi(points, colors, new Rect(min.x, min.y,
    max.x, max.y));
```

```
        delaunayLines = v.DelaunayTriangulation();
        spanningTree = v.SpanningTree(KruskalType.MINIMUM);
}
```

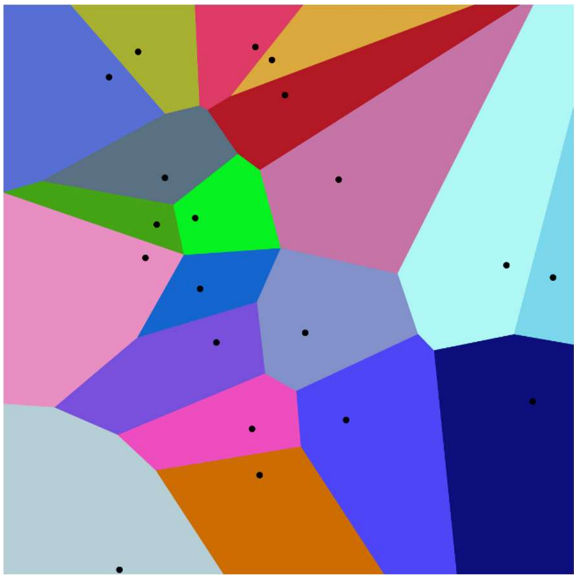


Figure 24 Voronoi Diagram (Wikipedia)

The Delaunay triangulation, as shown above, uses a method called Voronoi. Briefly what Voronoi diagrams are. In mathematics, a Voronoi diagram is a partition of a plane into regions close to each of a given set of objects. In the simplest case, these objects are just finitely many points in the plane, which will help us greatly with the creation of the rooms in our generation.

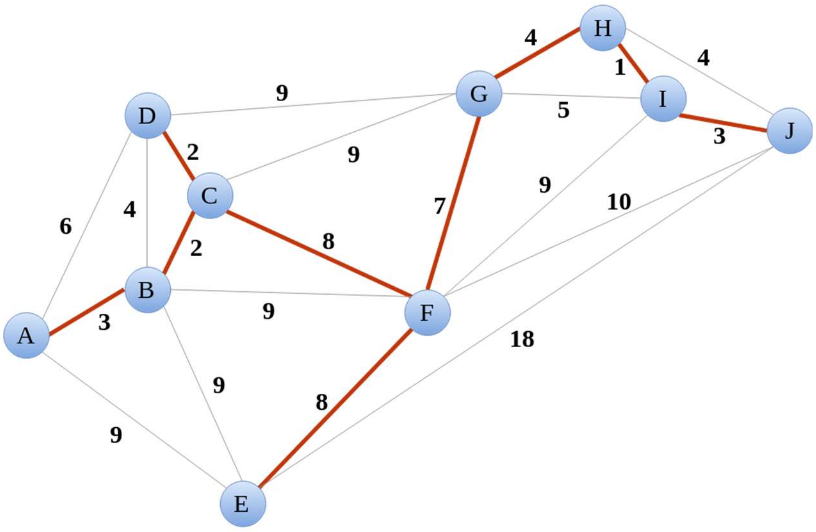


Figure 25 Minimum Spanning Tree (MST)

The next stage was to create a spanning tree between them to make sure we can reach all cells in out generator. A spanning tree is a sub-graph of an undirected connected graph, which includes all the graph vertices with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree. The edge may or may not have weights assigned to them. The spanning-tree algorithm I used was Kruskal's Minimum Spanning Tree Algorithm. To explain the algorithm, a minimum spanning tree (MST), connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. A spanning tree's weight is the sum of weights given to each edge of the spanning tree. The MST has a (V – 1) edges where V is the number of vertices in the given graph.

Finally with everything in place I could produce a map:

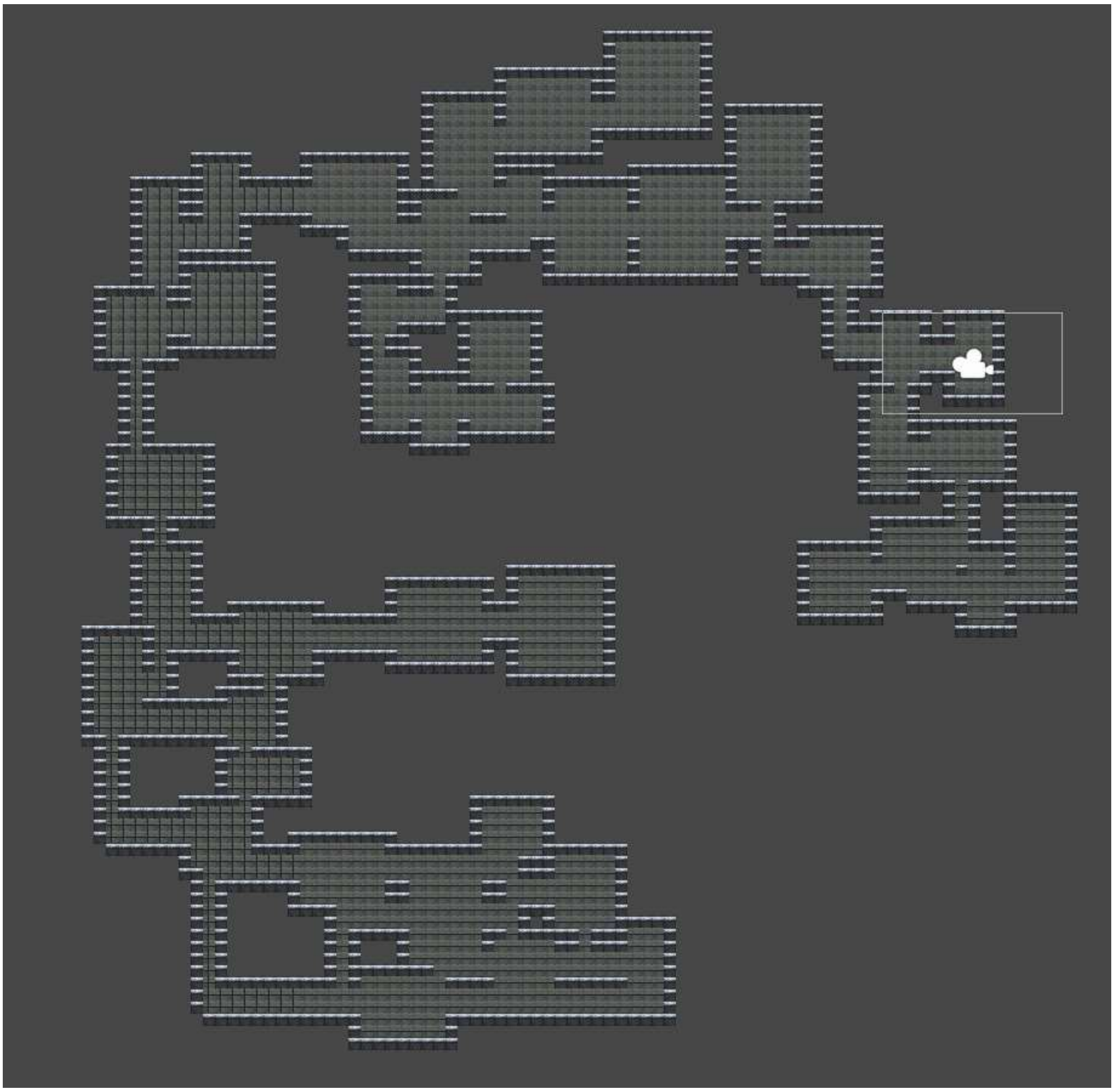


Figure 26 Delaunay Triangulation Generation

As we can see in Figure 31. The map is very different to the two other algorithms, since this algorithm is more optimized, more modern and aimed at 2D-Dungeon Map generation it creates a very nice map that you would see in a classic Legend Of Zelda Game.

5.5 Final Maps

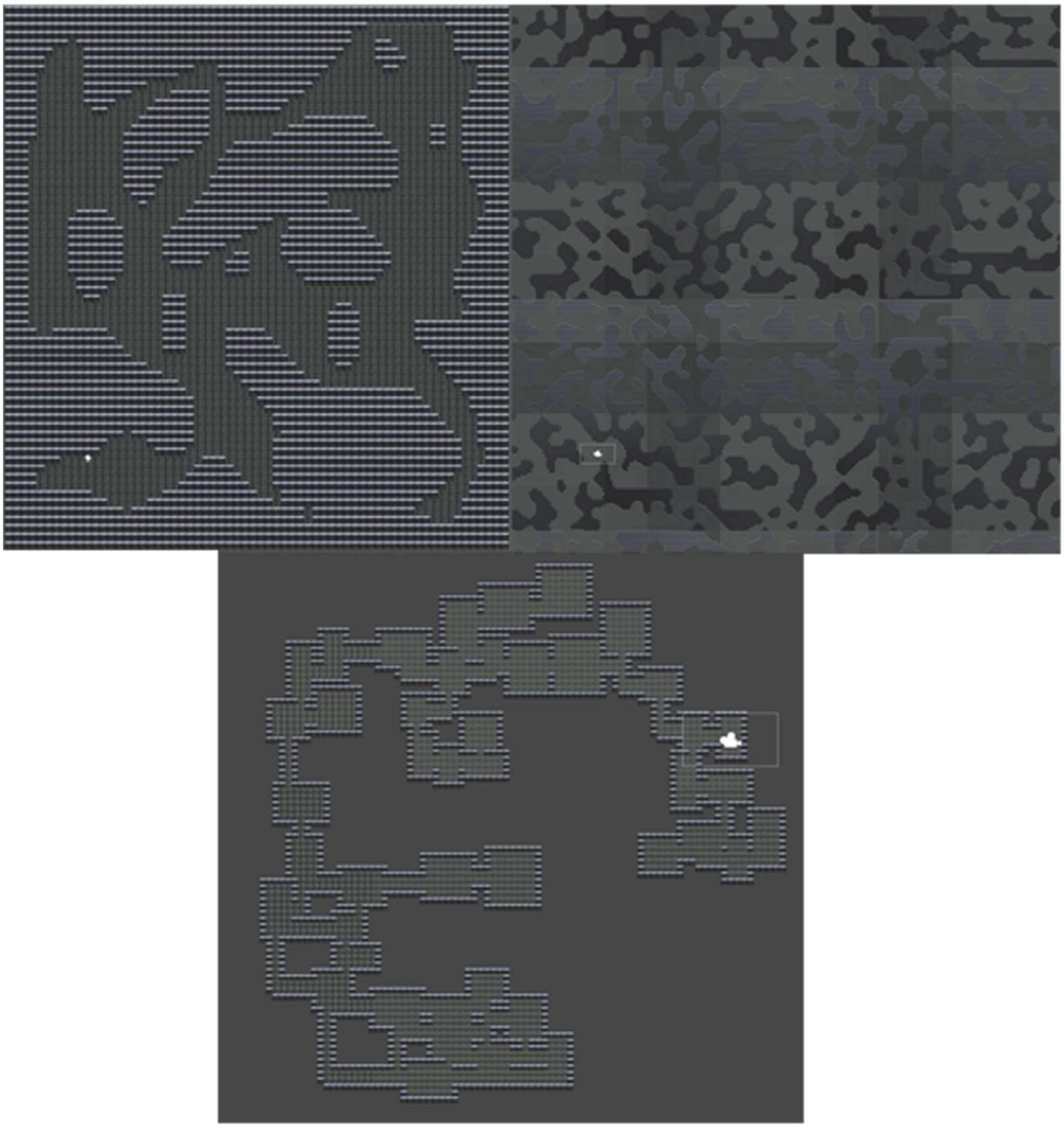


Figure 27 All three maps

As we can see shown in figure 32. All three maps are very different in how the map is generated. Perlin Noise and Cellular Automata look somewhat similar in how some of the corridors and caves look though Perlin Noise is too much bigger in scale. Delaunay Triangulation looks the most 2D-Dungeon like map generation with the map not having a massive surrounding area and each cave having different routes.

6 Testing

Testing was done manually alongside the implementation of the game itself. As each function and algorithm was implemented, it was tested for everyday use and edge-case scenarios before progressing.

Within the testing phase, I managed to get friends and family to help identify bugs or any other defects within the game itself. For the most part, the game was in its final state, and I got them to run multiple tests to see if it was possible to break or alter the map generation in any way. As they were new to the game, it provided good unbiased feedback compared to my little feedback of being the game's creator. The following bugs were found:

- Cellular Automation
  - Sometimes players spawned in wall areas making it impossible to access the entire map. It can be fixed by just going to the menu and back into the game until spawned in a good area.
  - There was a fair amount of collision issues; when the player collides with a tile corner, the screen can be seen as rotating.
- Perlin Noise
  - Some minor Collision issues.
- Delaunay Triangulation
  - Problems with the player not being visible on map creation can be fixed by just loading the map multiple times until the player is visible.

As this was a game I wasn't going to be releasing to the general public and only used to compare some PCG algorithms, I didn't think it was essential to fix these bugs. As the bugs are not game-breaking and can easily be fixed within the game itself.

Testing	Outcome	How it was done
Manual	Player Movement is in four directions.	Running the game and choosing a map generation ensures the player can move in 4 different directions.
Manual	Player Collision	Running the game, choosing a map generation ensures the player can collide with particular area walls.
Manual	Cellular Automation Algorithm generates map correctly	Running the game and choosing Cellular Automata ensures the map has developed an excellent map using the mouse scroll to view the entire map.
Manual	Perlin Noise Algorithm generated map correctly	Running the game and choosing Perlin Noise, ensuring the map has generated a good map using

		the mouse scroll to view the entire map.
Manual	Delaunay Triangulation generated map correctly	Running the game and choosing Delaunay Triangulation ensures the map has generated a valid map using the mouse scroll to view the entire map.
Manual	Ensure every segment of the dungeon's corridors in each algorithm connects correctly, allowing the player to move around.	Creation of a miniature pathfinder to ensure paths can be seen.
UI testing	Ensure the buttons on the menu screen work correctly.	Run the game and launch one of the three maps to see if they take you to the correct map.

6.1 User Testing and Evaluation

For the user evaluation, I investigated previous projects that have been done in general for games development/software development. As this isn’t so much of a complete game and only showcasing the three procedurally generated maps, I could only partially take parts from project focuses and get the evaluation to be aimed at the advantages and disadvantages of using such algorithms, comparing them to previous games, etc.

Finally, for the evaluation, referring back to the interim meeting can be found in the appendix. I decided to go with qualitative over quantitative, as talks of a quantitative approach were discussed, such as developing an A* algorithm to identify successful paths. With more time, this approach would have been perfect though I decided not to go with that approach due to the time constraints.

The Qualitative approach would essentially be a survey that gathers opinions about the resulting maps (whether it is fun or not? Long/short? Easy/hard?). Due to the software not being so much a level and more of a showcase, the selection of questions will be a bit limited and more aimed at exploring the map, such as the corridors, and now easy it is to venture around the entire map. Such questions can be answered using this way of evaluating the software:

- What was the user’s first impression of each procedurally generated map?
- How adequate was each map's randomization?
- Was the map impeding any exploration, such as closed-off walls?
- How was the load times for each map?

7 Results

7.1 User Evaluation Findings

By sending my evaluation questionnaire out, I managed to get a possible 7 out of 8 replies. The 7 signed informed consent can be found in the Appendix that follows the university guidelines of regulations.

7.1.1 User Qualitative Feedback

The individuals were asked a range of questions to understand which procedural generation algorithm is the best and which one comes with significant limitations. This project's age range was early adults to late adults as dungeon crawler games were introduced within the late 1970s as mentioned in the Introduction; this allows for the feedback to come from a range of different backgrounds and sources.

1. *What was your first Initial thought of each procedurally generated map? (Write a brief paragraph)*

This question was to gauge an understanding of everyone's first impressions on each map when firstly spawning into the map itself. It allows me to get information on what sort of feel each map provides and gives essential information to find which map is the best.

Participant Feedback:

- For Cellular automata, almost everyone said it was a very open space though; one participant went into greater detail, saying, "*The formation resembles islands or archipelagos, sparsely spread open spaces leave a lot for potential objectives without it feeling all too barren. Nooks and crannies do not feel too tight and leave the entire map very easily navigable. The enclosure of the area being generated makes the map look polished when zoomed out. Compared to Perlin, there are the occasional enclosed spaces, but few and far between relatively. Sizing of areas makes the level feel like a bonus level, medium sizing. Large contiguous areas look smooth and blobby, friendly and natural."* This gives a great understanding of the feeling cellular automata gives.
- For Perlin Noise, everyone agreed that it was a massive open map that provides some cave-like structures that were a bit tighter than cellular automata.
- For Delaunay Triangulation, some people said that it was a little too enclosed off and others gave feedback such as *"Feels almost like a room layout, a la Among Us/Out of Space. Could easily be developed for a dungeon crawler. Very familiar feeling, would certainly appeal to old school gamers".*

2. *Which map provided a complete experience (Ease of exploration, Interesting dungeon patterns)? (Provide a best, 2nd best, and most minor).*

This question gave me a good understanding of which map provides the user the best experience in terms of what you would see in a final 2D dungeon crawler game.

Participant Feedback:

1 Best 2 Second 3 Least

	User 1	User 2	User 3	User 4	User 5	User 6
CA	2	2	3	2	3	2
PN	3	3	2	1	1	3
DT	1	1	1	3	2	1

Table 4 Participant Feedback 1

As we can see, many people thought Delaunay Triangulation provided a complete experience in terms of exploration and exciting dungeon patterns. Though 2 out of the six people thought Perlin noise gave the best experience. Cellular automata got no votes for providing the best experience.

3. *Which map had the longest load time? (Provide a longest, 2nd longest and fastest)*

This question depends on the type of PC you are running, so it didn't provide the expected results. However, it did allow for people with a computer with lower specs to give some interesting results.

Participant Feedback:

	User 1	User 2	User 3	User 4	User 5	User 6
CA	1	2	2	1	n/a – Said all were instantaneous	1
PN	3	3	3	3	n/a	3
DT	2	1	1	2	n/a	2

Table 5 Participant Feedback 2

As we can see, the results are a little scattered through 3 out of the 6 had the same results may be indicating that cellular automata provide the fastest load time.

4. *Which map resembles a dungeon crawler game the best? (Tick the selected map)*

This question is a significant one and is essential to understanding which algorithm is the best in representing a 2-Dimensional dungeon map.

Participant Feedback:

	User 1	User 2	User 3	User 4	User 5	User 6
CA						
PN				x		
DT	x	x	x		x – Provided it gives an “Enter the gungeon” feel.	x

Table 6 Participant Feedback 3

As we can see from the six people, only one person disagreed and said Perlin noise provided the best dungeon type generation. Other than that, everyone agreed that Delaunay Triangulation gave the best dungeon crawler type experience, as it is the most developed and most modern algorithm for the dungeon generation. I believe this was correct.

5. *Which map provided the best randomisation? (Number of Corridors/Rooms)*

This question was to grasp which map provides the best randomization aspect rather than the actual dungeon itself. As randomisation is one of the most critical parts and as the maps get bigger the randomization aspect can be a definitive point in terms of map flow and replayability.

Participant Feedback:

	User 1	User 2	User 3	User 4	User 5	User 6
CA						
PN	x		x	x	x	
DT		x				x

Table 7 Participant Feedback 4

As we can see, four out of the six believe Perlin noise provided the best-randomized results as one participant wrote for Perlin noise *"I believe this was the most random out of the three, there were many different corridors and routes to explore on each time I launched this compared to other maps."* This gives good results as randomization, and the actual exploration of the map can be compared.

6. *In your own words, what could you do to improve each map? (More rooms, walls around map, more accessible areas)*

This question was to understand each map's advantages and limitations as the more you would need to improve on a map such as the actual map generation, the more it needs to be adjusted and improved.

Participant Feedback:

- For Cellular Automata, most participants gave the same feedback saying that the rooms were way too big, and one participant described it as "never-ending."
- For Perlin Noise, every participant enjoyed how "Swift" the map was to navigate and agreed that it was less open than cellular automata, which provided a better dungeon-like experience.
- For Delaunay Triangulation, everyone agreed that it was maybe too small in terms of corridor width and height. This would be easily adjusted, was myself (the developers) error for making the map tiles too small.

The main points to take away from the user evaluation is that each map provides a different experience in terms of exploration and ease of use. With Delaunay Triangulation providing the best in terms of 2D-Dungeon Crawler themed map generation and Perlin Noise surprisingly 2nd in terms of map structure and randomness of corridors and caves.

7.2 Goals Achieved

I believe the project successfully achieved the overall goal, which was to represent three procedurally generated maps and to be able to compare them against each other, with the added player control to be able to explore each map and assume on each one. The functionality and number of bugs are evident within the software itself, though with the amount of time available to represent the three algorithms sufficiently, I am happy with it. The addition of more game-like features such as enemies, keys, and bosses would have made the prototype more complete, though it would have left me with less time for the actual algorithms themselves.

7.3 Future Work

With all the bugs fixed and more functionality within the software itself, a lot of work can be done to make this project more of a game rather than a showcase of algorithms.

Firstly, I thought about making the variables within the map generation algorithms changeable within the software itself, allowing the player to create their own wacky interesting maps within the three algorithms. This would make the software more of a map generation game rather than a roguelike 2D dungeon crawler.

Other game-like elements such as enemies, keys, and bosses could've also been added, allowing for a complete game experience and also would possibly qualify for better assumptions and opinions on the maps as more of the map would be getting explored and would tie into more of what was described in the literature review on the games design elements explained by Short & Adams (Short & Adams, 2017).

Lastly, some Artificial Intelligence and Pathfinding were discussed but not further implemented also due to time constraints. These elements would have provided a more Quantitative approach to the evaluation stage. It would allow for an A* Pathfinder to find the shortest most successful paths in each algorithm to compare them to see if any maps were unable to provide a good approach.

8 Evaluation of End Product

8.1 Project Strengths

The most significant benefit of this project was implementing three algorithms rather than two as it allowed for a better comparison between each one. With two algorithms, I feel there would not have been enough to write about contrast. With Cellular Automata, Perlin Noise and Delaunay Triangulation being so different from one another, it allowed for a solid comparison. Also, all three algorithms work efficiently to generate the map, though having some bugs with collision and some player movement features that were not within the scope, to begin with.

The differences between each algorithm are noticeable and comparable such as the number of rooms and corridors generated—also the randomness of each algorithm and how it fits within a dungeon crawler theme.

8.2 Project Limitations

The most significant limitation within this project was the optimization of the program. As one of the eight original individuals that evaluated struggled to run the program due to the machine the individual was using. This indicates that there are required specifications the device must have for the maps to be able to be loaded. Other limitations include minor issues such as it not being a game and more of a showcase of software, as initially, I wanted to have a game to show at the end.

Another limitation I was dealt with happened in the three algorithms' original choice, as initially, I was going to use Generative Grammars instead of Delaunay Triangulation. But due to Generative Grammars being more of a content generation than a map generation algorithm, it is more complicated. If I managed to get Generative Grammars working within the program itself instead of Delaunay Triangulation, the maps' comparison would be even better.

8.3 Project Improvements

Firstly, further work should be done to fix the bugs such as collision and other small details such as image quality for the tiles and spawning issues. None of these issues mainly broke the software or made it impossible to make a fair evaluation, though it did not help explore the map itself.

The software delivers on being able to generate three maps through a selected procedural generation algorithm and displays each algorithm within a menu screen to be chosen. Though improvements such as:

- Improved Menu UI screen: Animations, Control's button/menu to allow for the player to know how the player moves and any other relevant information.
- More Algorithms: More algorithms such as Generative Grammars, Binary Spaced Partitioning and more could be added within the same art style to allow for a much better comparison. To then be a library of procedural generation algorithms.

- Game Elements: More game elements could have been added to allow for a more game-like experience, such as:
    - Enemies to dodge/avoid.
    - Keys and doors.
    - Items such as increased speed and more.
    - Level system.
- AI Pathfinding: An Artificial Pathfinder could have been added to allow for a more Quantitative Evaluation rather than the Qualitative that I went with. As it would find if a path could be created from one end to the other end of a map within the map generation itself.

An improved menu user interface such as animations like you would see within a fully produced game such as animated prison bars to fit the theme would provide a better User Experience for the player. Adding more algorithms within the game would make it more of a library of algorithms that could be used for educational materials in schools/colleges to teach how procedural generation algorithms work and how they differ between each other. More game elements are a must for this project to be seen as a game rather than an artifact, as enemies, keys, and doors would provide a fun and excitable experience to the user. With the procedurally generated maps, the replayability is endless. With enemies being added an A* pathfinder could be implemented for them to follow the player so that the player has a sense of urgency to complete the level and collect anything they need to within the map itself.

There is also a question of how random the algorithms are. With the algorithms being developed by myself with a few resources to aid in developing, the question of how random the three algorithms are will always be a factor, and to identify any clear patterns is way beyond the scope of this project but would be an exciting part to do more research into.

9 Conclusion

The project's outcome was a working piece of software that provided three different procedurally generated maps using the algorithms provided on the menu screen (Cellular Automata, Perlin Noise, and Delaunay Triangulation). The software had incorporated:

- Cellular Automata Map Generation
- Perlin Noise Map Generation
- Delaunay Triangulation Map Generation
- Menu Screen
- Player Control around the map
- A map viewer
- Collide-able walls

During the development and evaluation phases, I managed to identify some key benefits and find the answer that I was looking for when initially researching this thesis:

Firstly, using a cellular automata approach to generating maps within a 2D-Dungeon Crawler game is not as effective as initially thought. Though cellular automata is a fascinating algorithm and provide decent results on generating maps, the caverns and corridors that it presents are too open for a dungeon environment. They do not give an entire dungeon-like experience for the user. Cellular Automata is not out of the picture for 2D-Dungeon Crawler games, though. Since it is not as effective for a full-on map generation, it could be utilized for a single room. It provides randomness and does not isolate any parts where the individual cannot reach a particular area. Cellular Automata also having the ability to change certain variables such as the "Death Limit" and "Birth Limit," as mentioned in the implementation. Its definitely an algorithm that has a lot of depth and can be further researched for other purposes as this project used a similar modified method as described in Conway's Game of Life (Schif, 2006).

Next, using the Perlin Noise approach to generate maps within a 2D-Dungeon Crawler game was very effective and provided excellent results based on map size and how it was generated. Perlin Noise is undoubtedly the most straightforward algorithm to implement. I was not expecting outstanding results as previously I had used Perlin Noise for terrain generation and other aspects but never 2D-Dungeon Map generation. With the feedback from mostly everyone within the user evaluation giving praise on how well the corridors and rooms are generated, it made me think as initially I did not believe it was a good map generation algorithm. Though the algorithm is relatively easy to implement within the Unity engine, it comes with limitations. As we used Mount's theory on generating the Perlin Noise, the map is being taken from a Perlin image, making the randomization aspect very limiting as the same map is being generated each run (Mount, 2018).

Finally, using the Delaunay Triangulation approach to generating maps within a 2D-Dungeon Crawler game was the best method. The map size was excellent, and the type of rooms and corridors generated. This method is the best as it is the most up to date out of the three and aimed towards 2D Procedural Map Generation. With the corridors being nice small areas in which a player can then find themselves venturing into small rooms, it provides the complete dungeon experience for map design. The randomness of how the level is developed, every single run being completely different every time, also gives the player a replay-able experience they will not want to miss out on.

Overall, I feel this project was a success and achieved all the key aims and objectives that I set out initially:

Objectives:

1. *Research early and modern-day procedural generation techniques and choose multiple valid algorithms that can be implemented into a 2D dungeon crawler-themed game.* This was provided through the literature review by going through the background of procedural generation and the different algorithms involved within the sector, current day, and the past.

2. *Implement these algorithms into a game/system that clearly shows the differences between each proc-gen technique*. A system was created containing three different procedural generation algorithms such as Cellular Automata, Perlin Noise, and Delaunay Triangulation.

3. *Compare the different proc-gen algorithms from the early and modern-day and make assumptions for each one.* Early algorithms such as Cellular Automata and Perlin Noise were compared against a more modern approach like Delaunay Triangulation, currently used for many Dungeon Crawler Games.

4. *Identify which algorithm provides the best efficiency for generating dungeons, hallways, enemies, and treasures.* An algorithm was specified for the best map generated, but elements such as enemies and treasures were not implemented with the amount of time given.

The project also answered the research questions of:

5. *How have the different procedural generation algorithms developed over time?*
   This question was answered within the literature review going over the background, and modern-day procedural generation uses.

6. *What tasks-specific procedurally generated algorithms are applied in dungeon generated games?*
   This was also answered within the literature review going over static and dynamic procedural generation and procedural content generation.

7. *Which procedurally generated algorithm will provide the best map-generated results (Map accessibility, ease of exploration, and the player can complete the level)?*
   This was answered within the user evaluation of the software itself, providing good results on which procedural algorithms provided the best maps and randomness.

As the software was not fully completed into a game, it lacked many qualities that could make it fun and engaging to the user, such as enemies, keys, doors, narrative, dialogue, and much more. However, these aspects can be added for future work, expanding on the game elements, and possibly improving the map generation algorithms. The user evaluation gives a significant insight into what improvements can be made and what can be adjusted within the software. Aside from this, the project could be used to understand how each procedural map generation technique works and what can be produced of that chosen algorithm.

I believe the work here shows the advantages and disadvantages of using different procedural map generation techniques for 2D-Dungeon Crawler Rogue-like games, since this project aims at old and modern-day game developers. There are many avenues to take a project like this to suit a variety of different audiences.

10 References

Arttu Marttinen. (2017). Procedural Generation of Two-Dimensional Levels. *Metropolia*, *October*.

Aycock, J. (2016). Procedural Content Generation. In *Retrogame Archeology*. https://doi.org/10.1007/978-3-319-30004-7_6

Barriga, N. A. (2019). A Short Introduction to Procedural Content Generation Algorithms for Videogames. *International Journal on Artificial Intelligence Tools*, *28*(2). https://doi.org/10.1142/S0218213019300011

Bevilacqua, F. (2013, October 24). *Finite-State Machines: Theory and Implementation*. https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

Bontchev, B. (2016). Modern Trends in the Automatic Generation of Content for Video Games. *Serdica Journal of Computing*, *10*(2), 133–166.

Cazzaro, I. (n.d.). *John Conway's Game of Life is a two-dimensional cellular automaton... | Download Scientific Diagram*. Retrieved April 20, 2021, from https://www.researchgate.net/figure/John-Conways-Game-of-Life-is-a-two-dimensional-cellular-automaton-created-in-1970-Its_fig2_319561154

De Carli, D. M., Bevilacqua, F., Pozzer, C. T., & D'Ornellas, M. C. (2011). A survey of procedural content generation techniques suitable to game development. *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, *September 2018*, 26–35. https://doi.org/10.1109/SBGAMES.2011.15

Erstu, E., Sell, J., & Valli, S. (2012). *Perlin Noise Generator*.

Federale, P., & Losanna, D. I. (2015). *École Polytechnique Fédérale De Lausanne*. *April 2013*, 2013–2014.

Fornander, P. (2013). *Game Mechanics Integrated with a Lindenmayer System*. 33.

Freiknecht, J., & Effelsberg, W. (2017). A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*, *1*(4), 1–34. https://doi.org/10.3390/mti1040027

Grand Theft Auto 5. (n.d.). *Rockstar Games - Grand Theft Auto V*. Retrieved April 20, 2021, from https://www.rockstargames.com/V/restricted-content/agegate/form?redirect=https%3A%2F%2Fwww.rockstargames.com%2FV%2F&options=&locale=en_us

Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications and Applications*, *9*(1). https://doi.org/10.1145/2422956.2422957

Janssens, K. (2007). Cellular automata. *Computational Materials Engineering*, 109–150. https://doi.org/10.1016/B978-012369468-3/50004-6

Korn, O. (2017). Game Dynamics. In *Game Dynamics*. https://doi.org/10.1007/978-3-319-53088-8

Lavender, B., & Thompson, T. (2016). A generative grammar approach for action-adventure map generation in the Legend of Zelda. *AISB Annual Convention 2016, AISB 2016*, 2–6.

Lee, D. T., & Schachter, B. J. (1980). Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences*, *9*(3), 219–242. https://doi.org/10.1007/BF00977785

Middag, B. (2016). *Controllable generative grammars for multifaceted generation of game levels*.

Minecraft, M. (n.d.). *Generating Perlin Noise - Computer Science and Technology - Off Topic - Minecraft Forum*

- *Minecraft Forum*. 2011. Retrieved April 19, 2021, from https://www.minecraftforum.net/forums/off-topic/computer-science-and-technology/482027-generating-perlin-noise?page=4

Mount, D. (2018). *CMSC 425 : Lecture 14 Procedural Generation : Perlin Noise*. 1–10.

Razafindrazaka, F. (2009). *Delaunay Triangulation Algorithm and Application to Terrain Generation*. *May*, 2–12.

Schif, J. L. (2006). Introduction to Cellular Automata. *Seminar "Organic Computing,"* 19.

Short, T. X., & Adams, T. (2017). *Procedural Generation in Game Design* (T. X. Short & T. Adams (eds.)). http://www.ghbook.ir/index.php?name=&option=com_dbook&task=readonline&book_id=13629&page=108&chkhashk=03C706812F&Itemid=218&lang=fa&tmpl=component

Smith, G. (2015). An Analog History of Procedural Content Generation. *Foundations of Digital Games*, *Fdg*, 0–5. http://sokath.com/main/files/1/smith-fdg15.pdf

Sportelli, F. (2014). *A PROBABILISTIC GRAMMAR FOR Francesco Sportelli Giuseppe Toto Gennaro Vessio*. *July*. https://doi.org/10.13140/2.1.3820.4163

Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss, M., & Stanley, K. O. (2013). Procedural Content Generation : Goals, Challenges and Actionable Steps. *Artificial and Computational Intelligence in Games*, *6*, 61–75. http://drops.dagstuhl.de/opus/volltexte/2013/4351%5Cnhttp://drops.dagstuhl.de/opus/volltexte/2013/4336/

Van Der Linden, R., Lopes, R., & Bidarra, R. (2014). Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(1), 78–89. https://doi.org/10.1109/TCIAIG.2013.2290371

White, K. (2019). *Implementation of a Procedural Level Generation Engine for Developing Rouge-like Dungeons Faculty of Science and Technology BSc (Hons) Games Programming May 2018 Implementation of a Procedural Level Generation Engine for Developing Rouge-like Dungeons B*. *January*, 0–48.

Williams, N. (2014a, June). *(227) Binary Space Partition Dungeon Generation - YouTube*. https://www.youtube.com/watch?list=UU4i6i-HBjQzBC_pC7l8GB1A&v=AUJx3xYM4n4&feature=youtu.be

Williams, N. (2014b, June 4). *Delaunay Triangulation Dungeon Generation - YouTube*. https://www.youtube.com/watch?v=CaI6edoGbFY

Wolfram, S. (2006). Cellular Automata. *18*(3), 28–30.

Appendices

A Project Overview

Initial Project Overview

SOC10101 Honours Project (40 Credits)

Title of Project: The Development Of Procedural Generation Within Games

Overview of Project Content and Milestones

The aim of this research of thesis is to explore the different procedural generation techniques and algorithms used from the past and present, and to be able to show this through a game/system. It is to also write a literature review based on the algorithms used that will provide insight into how much they have evolved and adapted into our modern-day procedural generation techniques and algorithms.

The Main Deliverable(s):

The creation of a demonstratable game/system that illustrates the proc-gen techniques from the past and present (within a time period), e.g. Show how the techniques have developed over a certain time period (using the literature to indicate when each technique was first introduced) and showing how the one demonstration system/game alters as each new technique is applied.

A literature review that covers the different aspects of procedural generation used from the past and present (within a time period), and to compare, analyse and demonstrate each technique to figure out which is the best.

The final dissertation with the literature review and all other relevant sources.

The Target Audience for the Deliverable(s):

The target audience will include students/teachers that wish to learn knowledge within the procedural generation sector within the games development environment.

The Work to be Undertaken:

The work to be undertaken for the literature review will include thorough research into the history of procedural generation from when it was first introduced to how it has evolved into the algorithms, we use now within video games.

Work within the programming and the construction of this game/system will include learning the algorithms used in past procedural generation techniques and how they have been developed and evolved into the algorithms we use now, by demonstrating the new techniques over time.

Additional Information / Knowledge Required:

Knowledge of programming within languages that are used commonly throughout the games industry such as C, C++, C# and Java.

Some knowledge of the history of the games industry such as commonly used algorithms for procedural generation and artificial intelligence.

Information Sources that Provide a Context for the Project:

- Books.
- Encyclopedias.
- Magazines.
- Databases.
- Newspapers.
- Library Catalog.
- Internet.

The Importance of the Project:

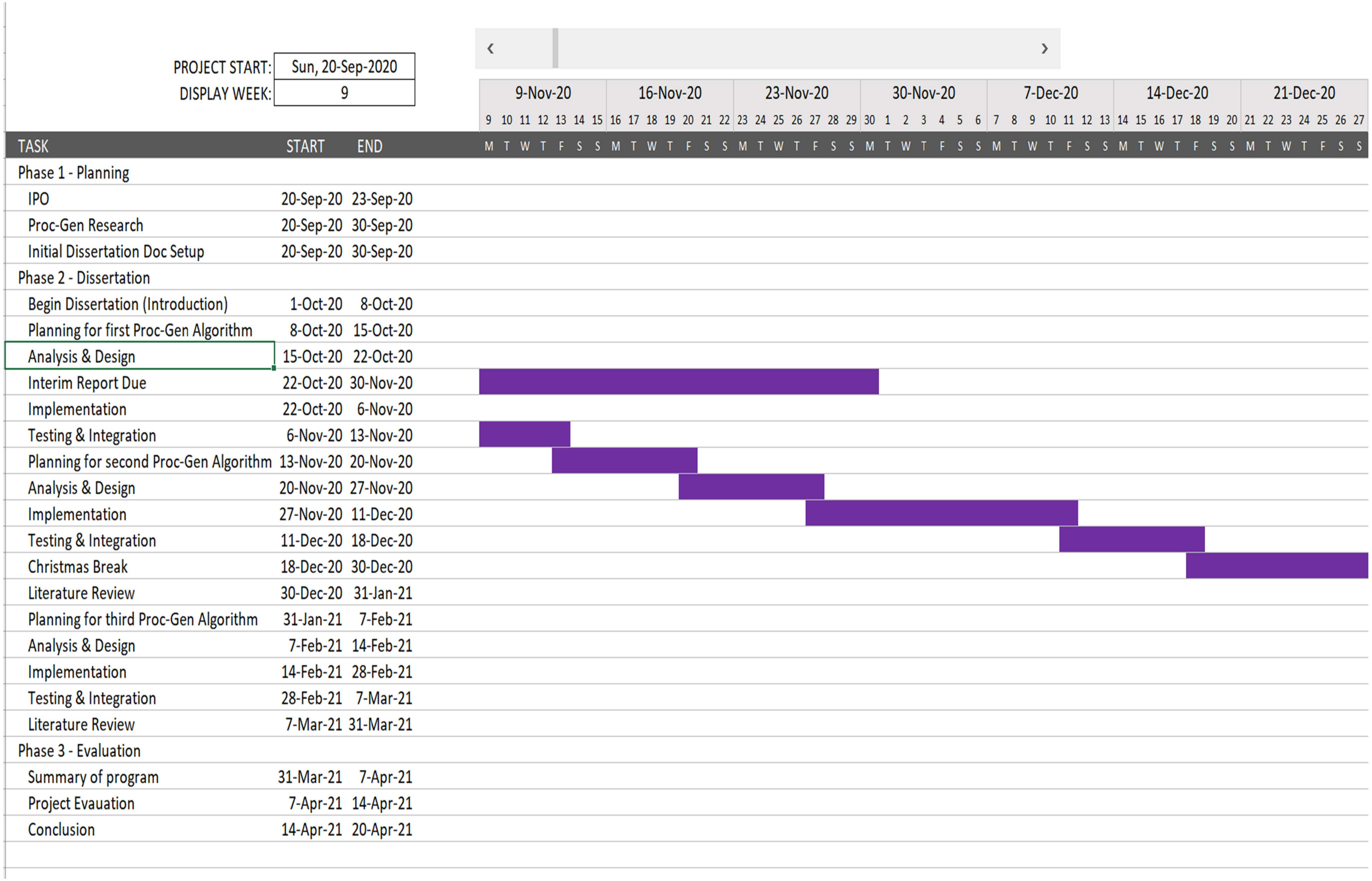
This project will provide insight into how far procedural generation has come within a certain time frame, allowing for any future developments made within this sector of the games industry to be comparable by its predecessors.

The Key Challenge(s) to be Overcome:

The key challenges to overcome when writing the literature review will be finding the source information of the most common procedural generation techniques used in the past and present, as a lot of these techniques will have been altered a lot over time, meaning finding the root source of each technique could appose challenging.

Another challenge will be writing the game/system itself; this will involve gathering information from books and the web to create an efficient game/system that covers the aspects of the procedural generation techniques used.

B Project Plan



PROJECT START: Sun, 20-Sep-2020

DISPLAY WEEK: 9

TASK	START	END
Phase 1 - Planning		
IPO	20-Sep-20	23-Sep-20
Proc-Gen Research	20-Sep-20	30-Sep-20
Initial Dissertation Doc Setup	20-Sep-20	30-Sep-20
Phase 2 - Dissertation		
Begin Dissertation (Introduction)	1-Oct-20	8-Oct-20
Planning for first Proc-Gen Algorithm	8-Oct-20	15-Oct-20
Analysis & Design	15-Oct-20	22-Oct-20
Interim Report Due	22-Oct-20	30-Nov-20
Implementation	22-Oct-20	6-Nov-20
Testing & Integration	6-Nov-20	13-Nov-20
Planning for second Proc-Gen Algorithm	13-Nov-20	20-Nov-20
Analysis & Design	20-Nov-20	27-Nov-20
Implementation	27-Nov-20	11-Dec-20
Testing & Integration	11-Dec-20	18-Dec-20
Christmas Break	18-Dec-20	30-Dec-20
Literature Review	30-Dec-20	31-Jan-21
Planning for third Proc-Gen Algorithm	31-Jan-21	7-Feb-21
Analysis & Design	7-Feb-21	14-Feb-21
Implementation	14-Feb-21	28-Feb-21
Testing & Integration	28-Feb-21	7-Mar-21
Literature Review	7-Mar-21	31-Mar-21
Phase 3 - Evaluation		
Summary of program	31-Mar-21	7-Apr-21
Project Evauation	7-Apr-21	14-Apr-21
Conclusion	14-Apr-21	20-Apr-21

C Interim Report

SOC10101 Honours Project (40 Credits)

Week 9 Report

Student Name: Arran Smedley

Supervisor: Simon Wells

Second Marker: Babis Koniaris

Date of Meeting: 18/11/2020

Can the student provide evidence of attending supervision meetings by means of project diary sheets or other equivalent mechanism? [✓ yes] no*

If not, please comment on any reasons presented

Please comment on the progress made so far

Progress is good so far, but the student needs to set a clear and limited scope in terms of work to be carried out (e.g. which methods to implement, to what end), that will assist with the rest of the project: the finalisation of the literature review, the development work and the evaluation of the results.

Is the progress satisfactory? [✓ yes] no*

Can the student articulate their aims and objectives? [✓ yes] no*

If yes then please comment on them, otherwise write down your suggestions.

The student can articulate their aims and objectives, which are the implementation and comparison of few procedural map generation methods, applied in the context of a very simple game framework.

Does the student have a plan of work? [✓ **yes**] **no***

If yes then please comment on that plan otherwise write down your suggestions.

The student does have a plan of work, regarding timelines of development, writing, etc, but there’s no plan yet for the evaluation, as that’s missing so far.

Does the student know how they are going to evaluate their work? **yes** [✓ **no**]*

If yes then please comment otherwise write down your suggestions.

The student did not have a concrete evaluation plan for the work. Evaluation can be quantitative or qualitative. Qualitative could be a survey for example, that gathers opinions about the resulting maps (is it fun or not? Long/short? Easy/hard? etc). Quantitative evaluation could be part of the map generation (given a start and end point, is there a valid path? Could use A* for that), or the survey (e.g. how long did it take players to complete the map? Did they give up at some point?). Survey design will take a bit of research and time, so the student needs to plan for that.

Any other recommendations as to the future direction of the project

The student has to tighten the scope of the project, and should focus on the map generation/procedural level design, instead of keys, enemies or health. Making a nice game out of it can come later on in spare time, and the priority is research, development and evaluation.

By identifying concrete research questions (e.g. “can I make fun procedural maps?”) and evaluation criteria (e.g. “what makes a map fun? Length? Variation? Challenge?”) the student can obtain clear requirements for the algorithms to implement, and requirements and constraints are great scope limiters which can help the student to focus on what’s **really** required, and navigate around distractions and nice-to-have features that will not be assessed.

Signatures: Supervisor Simon Wells Second Marker Babis Koniaris

Student Arran Smedley

The student should submit a copy of this form to Moodle immediately after the review meeting; A copy should also appear as an appendix in the final dissertation.

D Project Management

D.A Work Diary

Work Diary

Meeting 1 (Wed 30/09/2020 12:15 - 13:00)
Notes

- Discussed project plan.
- Discussed meeting times and suitability.
- General conversation about dissertation.

Meeting 2 (Wed 07/10/2020 12:15 - 13:00)
Notes

- General chat on dissertation ideas.
- Beginning of project plan.
- Early discussion of Initial Project Overview.
-

Meeting 3 (Wed 14/10/2020 12:15 - 13:00)
Notes

- N/A Didn't attend.

Meeting 4 (Wed 21/10/2020 12:15 - 13:00)
Notes

- Finished Initial Project Overview for submission.
- General chat about what algorithms I will be using.

Meeting 5 (Wed 28/10/2020 12:15 - 13:00)
Notes

- Introduction tuning and Start of literature review.

Meeting 6 (Wed 04/11/2020 12:15 - 13:00)
Notes

- Further work on Literature Review.
- Development on first algorithm (Cellular Automata).

Meeting 7 (Wed 11/11/2020 12:15 - 13:00)
Notes

- Further work on literature review.
- Not a lot of work done due to other coursework's.

Interim Meeting (Wed 18/11/2020 12:15 - 13:00)

Notes

- Development on first algorithm lateral stages small demo.
- Literature Review work in the background.

Meeting 8 (Wed 02/12/2020 12:15 - 13:00)

Notes

- Lateral stages of project plan.
- Beginning Initial Project Overview.

Meeting 9 (Wed 03/02/2021 12:30 - 13:00)

Notes

- Finished 2 algorithms after break (Cellular Automata and Perlin Noise).
- Rough draft literature review done.

Meeting 10 (Wed 17/02/2021 12:30 - 13:00)

Notes

- Demo of all three algorithms working.
- Clean up on literature review.

Meeting 11 (Wed 24/02/2021 12:30 - 13:00)

Notes

- Lateral stages of dissertation.
- General chat (problems with PC)

Meeting 12 (Wed 03/03/2021 12:30 - 13:00)

Notes

- More development, better collision and bug fixes.
- Still general problems with PC (Bought new one)

Meeting 13 (Wed 10/03/2021 12:30 - 13:00)

Notes

- Evaluation talk, user evaluation and code of ethics.

Meeting 14 (Wed 17/03/2021 12:30 - 13:00)

Notes

- Conclusion and Evaluation talk.

D.A Work Log

Work Log

(Wed 02/12/2020)
Notes

- Began work on Cellular Automata, learning the different ways it can be implemented into a 2D- Dungeon crawler type map.
- Looked at other algorithms briefly to accurately define a time when I can get all this done.
- Got basic WASD Player movement implemented.

(Wed 07/12/2020)
Notes

- Got basis of algorithm working with example tile map taken from the web some issues such as tile sizes.

(Wed 14/12/2020)
Notes

- Algorithm working accurately player can move around map.
- Looking onto Perlin Noise.

(Wed 21/12/2020)
Notes

- Having some issues with Perlin Noise generating a map. Player keeps falling under the map.
- Looking at implementing my own tile map within both algorithms.

(Wed 04/01/2020)
Notes

- Picking up after the Christmas holidays.
- Managed to fix the bug with Perlin Noise was to do with Layer Prioritization in Unity.

(Wed 11/01/2020)
Notes

- Started working on generative grammars and getting the map generation to work.
- Struggled to find any materials on map generation for generative grammars.

(Wed 18/01/2020)
Notes

- Still really struggling with generative grammars thinking of changing to an alternate algorithm such as Delaunay Triangulation.
- Emailed supervisor gave some good input.
- Generative Grammar map generation is very difficult and time consuming to implement I feel I wouldn't have a lot of time if I decided to look further into this problem.

(Wed 25/01/2020)
Notes

- Got my own custom tile map implemented within all three algorithms and produced a demo to supervisor.
- Looking at working on collision and any bugs.

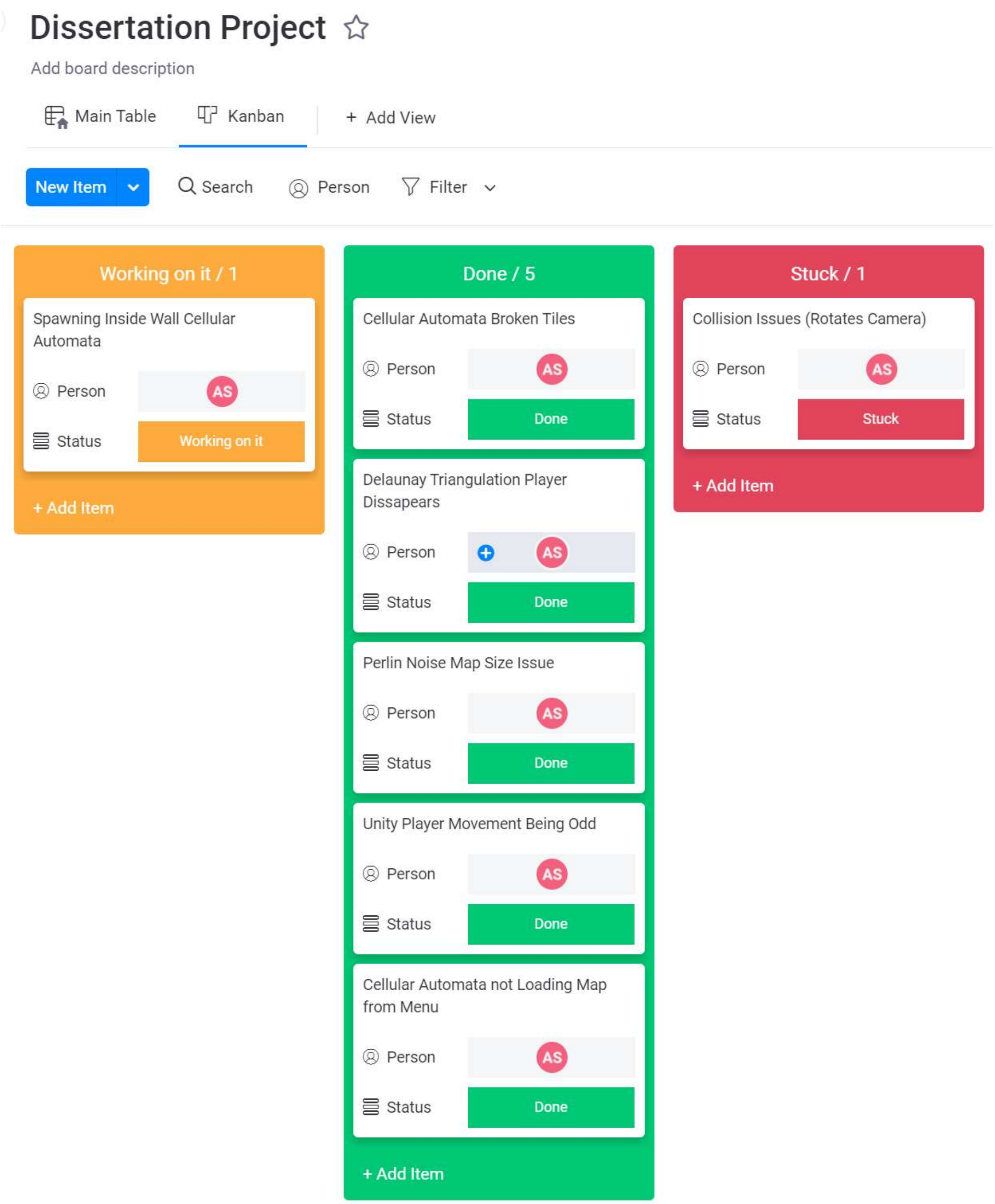
(Wed 03/02/2021)
Notes

- Menu screen added with three options for the three algorithms.
- Collision added and some bugs fixed.

(Wed 17/02/2021)
Notes

- Added back button functionality so player doesn't get stuck in map.
- Began User evaluation and end of product.

D.A Bug and Issue Tracking (Kanban Board)



Dissertation Project

Add board description

Working on it / 1

Spawning Inside Wall Cellular Automata
Person: AS
Status: Working on it

+ Add Item

Done / 5

Cellular Automata Broken Tiles
Person: AS
Status: Done

Delaunay Triangulation Player Dissapears
Person: AS
Status: Done

Perlin Noise Map Size Issue
Person: AS
Status: Done

Unity Player Movement Being Odd
Person: AS
Status: Done

Cellular Automata not Loading Map from Menu
Person: AS
Status: Done

+ Add Item

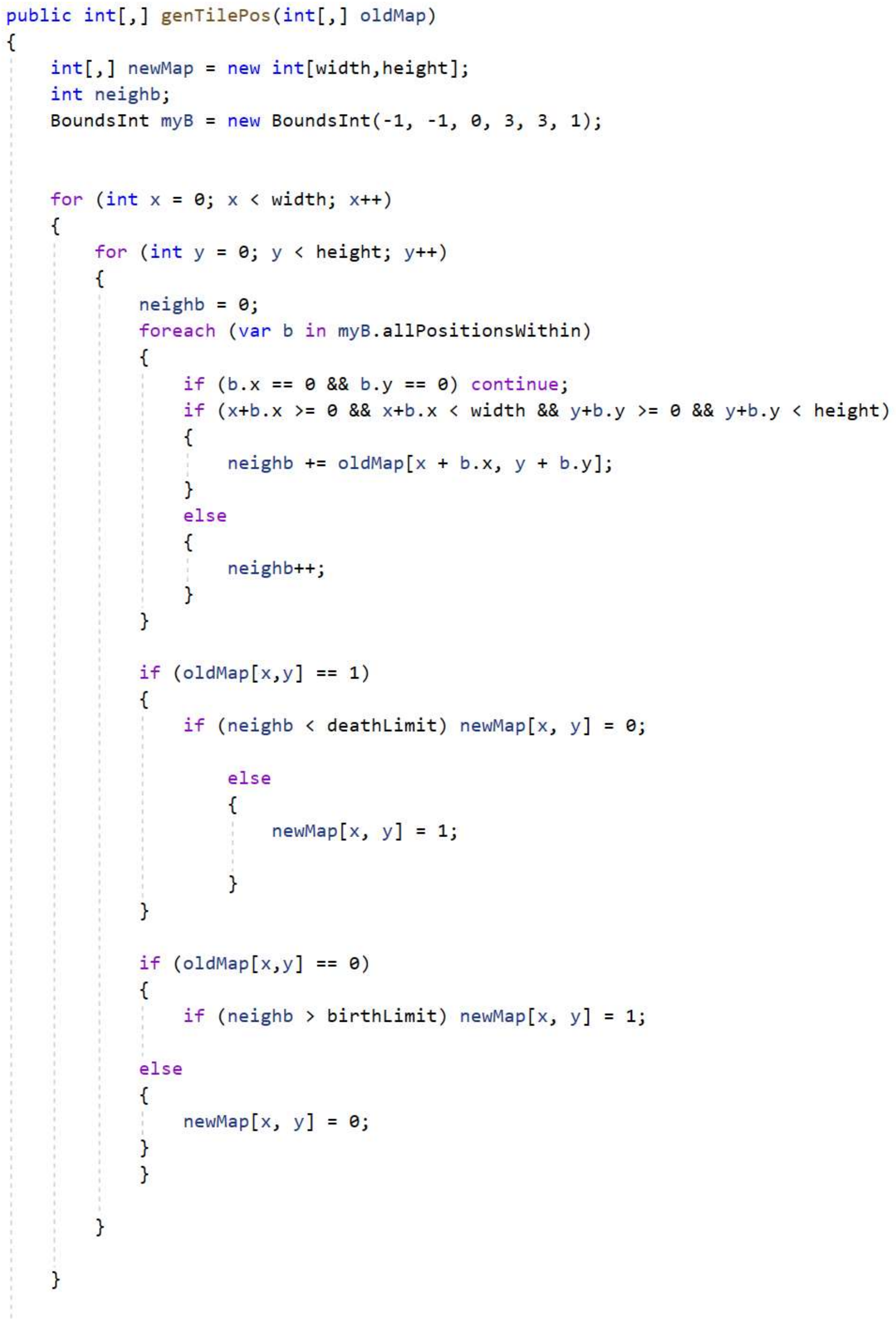
Stuck / 1

Collision Issues (Rotates Camera)
Person: AS
Status: Stuck

+ Add Item

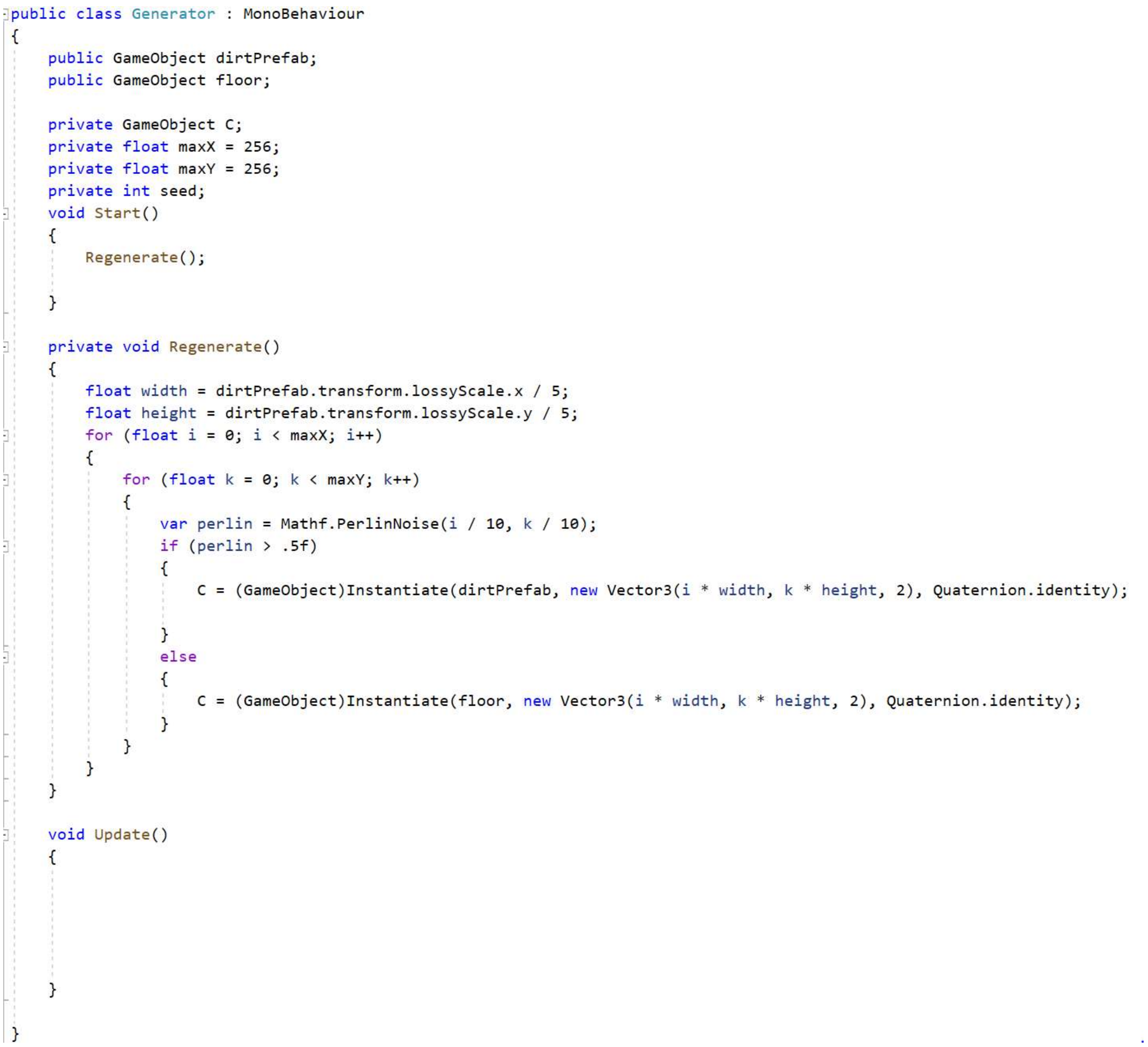
E Procedural Generation Code Files

E.A Cellular Automata



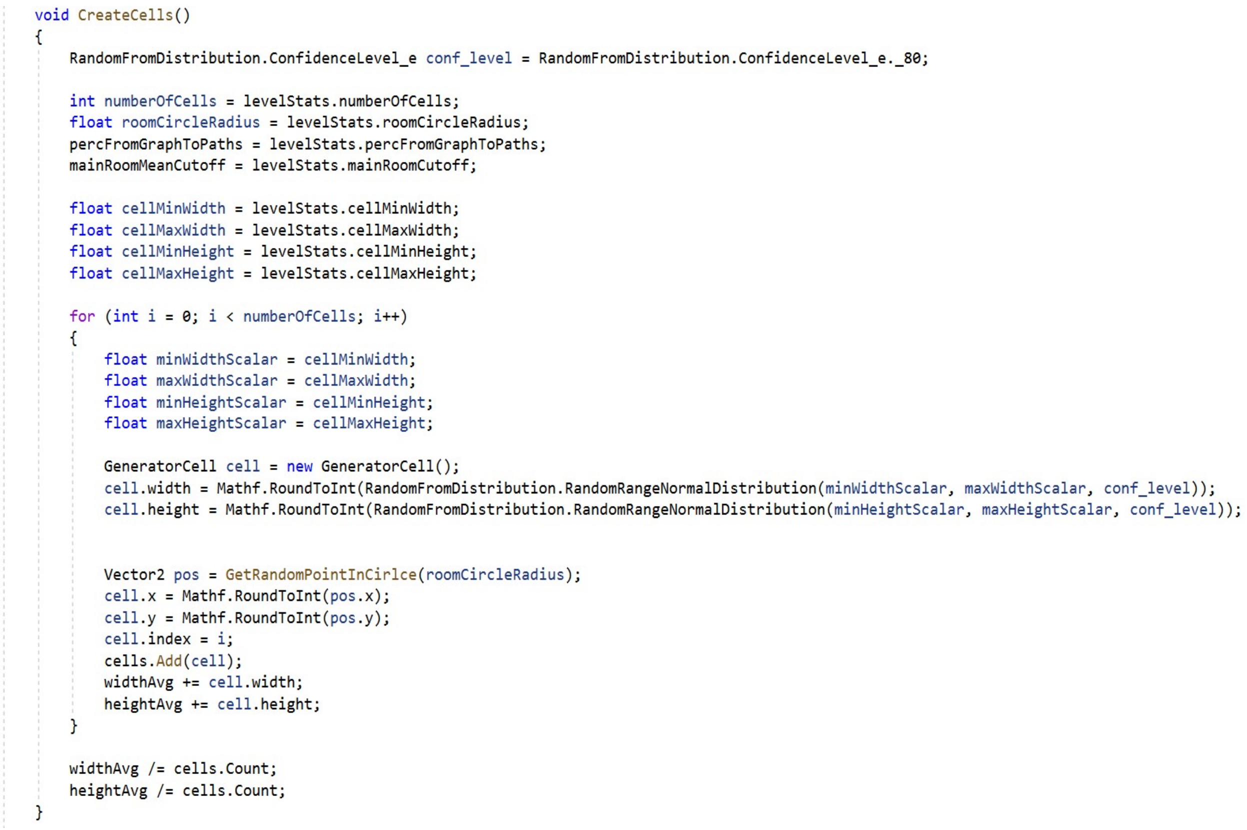
```
public int[,] genTilePos(int[,] oldMap)
{
    int[,] newMap = new int[width,height];
    int neighb;
    BoundsInt myB = new BoundsInt(-1, -1, 0, 3, 3, 1);


    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            neighb = 0;
            foreach (var b in myB.allPositionsWithin)
            {
                if (b.x == 0 && b.y == 0) continue;
                if (x+b.x >= 0 && x+b.x < width && y+b.y >= 0 && y+b.y < height)
                {
                    neighb += oldMap[x + b.x, y + b.y];
                }
                else
                {
                    neighb++;
                }
            }

            if (oldMap[x,y] == 1)
            {
                if (neighb < deathLimit) newMap[x, y] = 0;

                    else
                    {
                        newMap[x, y] = 1;

                    }
            }

            if (oldMap[x,y] == 0)
            {
                if (neighb > birthLimit) newMap[x, y] = 1;

            else
            {
                newMap[x, y] = 0;
            }
            }

        }

    }
```

E.B Perlin Noise

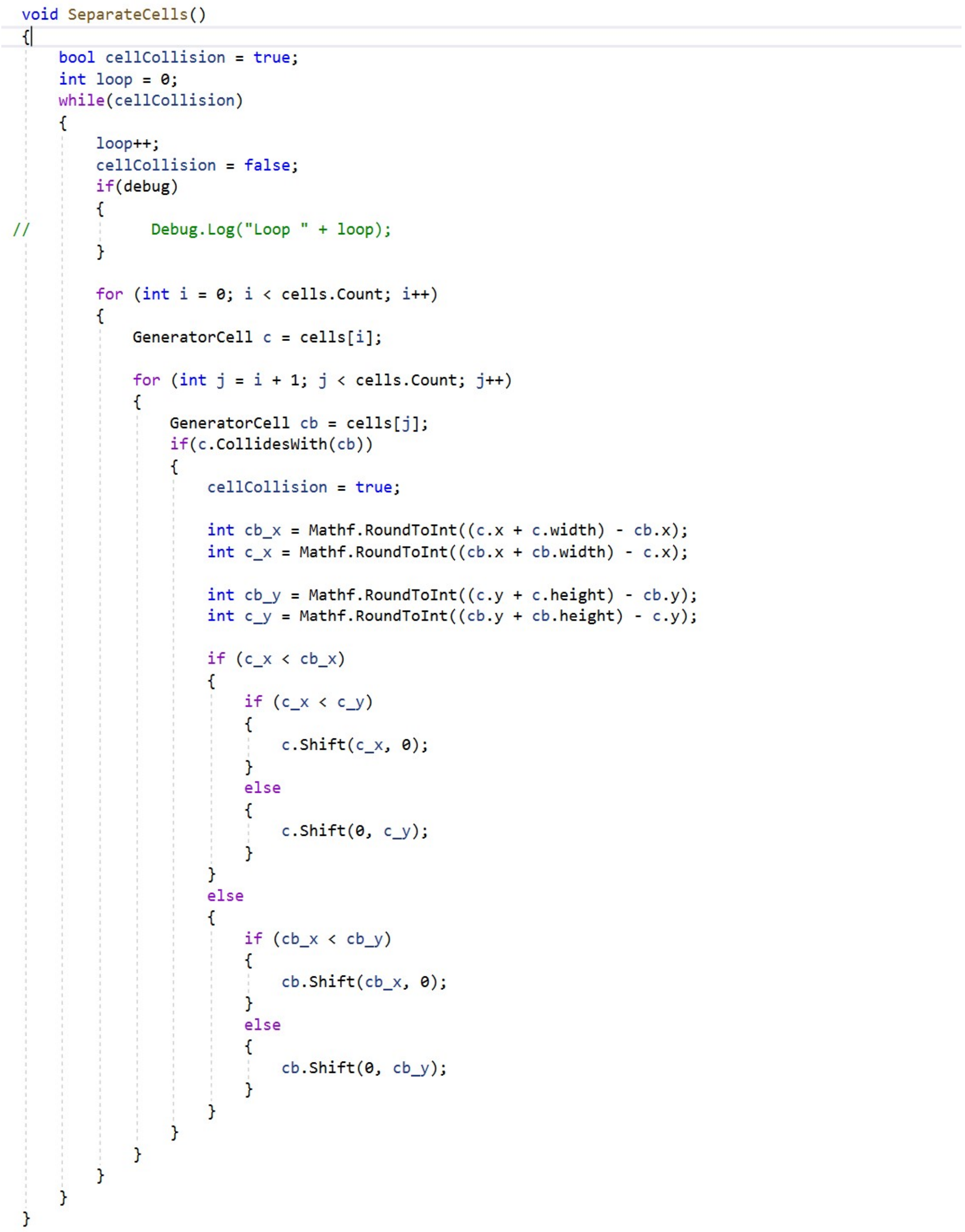


```
public class Generator : MonoBehaviour
{
    public GameObject dirtPrefab;
    public GameObject floor;

    private GameObject C;
    private float maxX = 256;
    private float maxY = 256;
    private int seed;
    void Start()
    {
        Regenerate();

    }

    private void Regenerate()
    {
        float width = dirtPrefab.transform.lossyScale.x / 5;
        float height = dirtPrefab.transform.lossyScale.y / 5;
        for (float i = 0; i < maxX; i++)
        {
            for (float k = 0; k < maxY; k++)
            {
                var perlin = Mathf.PerlinNoise(i / 10, k / 10);
                if (perlin > .5f)
                {
                    C = (GameObject)Instantiate(dirtPrefab, new Vector3(i * width, k * height, 2), Quaternion.identity);

                }
                else
                {
                    C = (GameObject)Instantiate(floor, new Vector3(i * width, k * height, 2), Quaternion.identity);
                }
            }
        }
    }

    void Update()
    {




    }

}
```

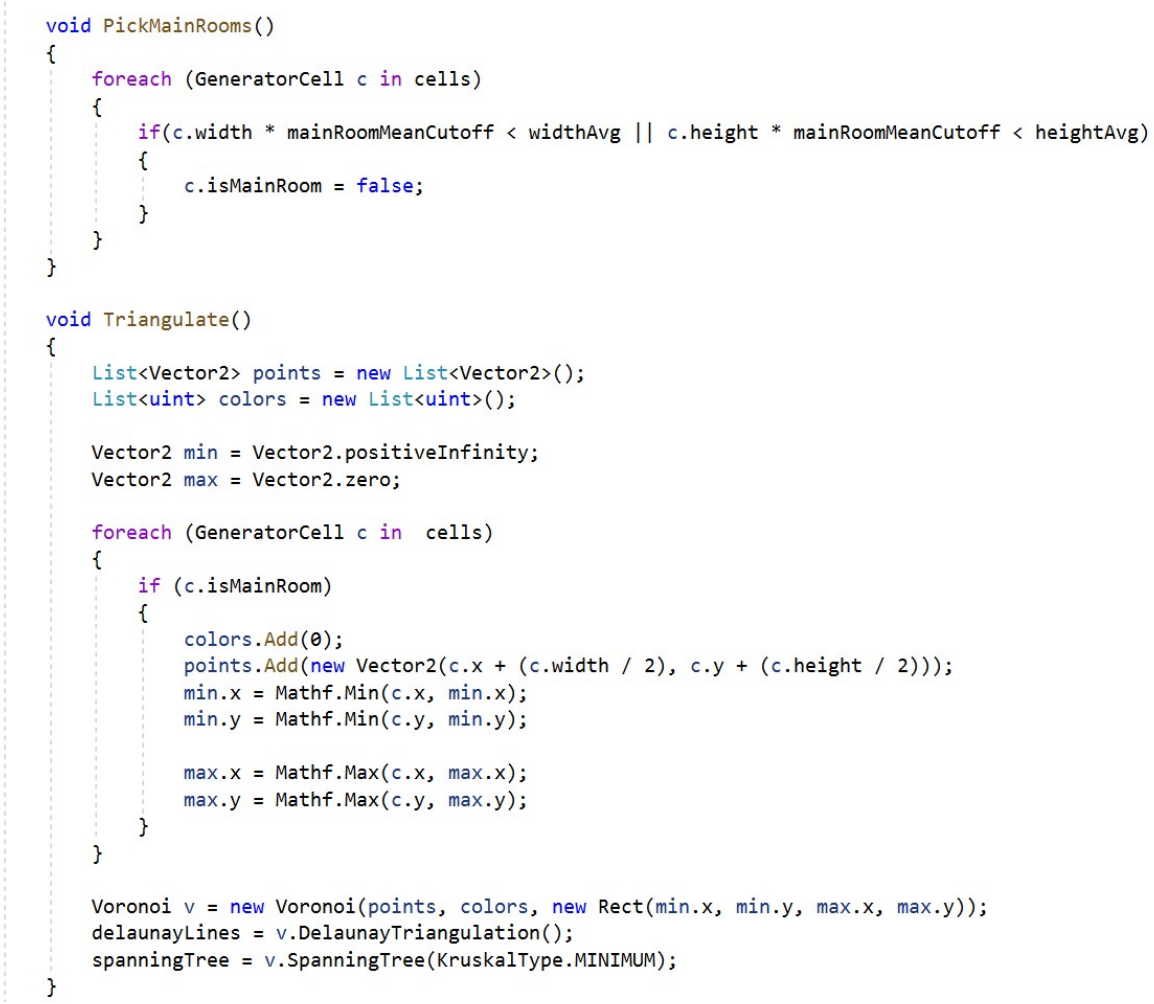
E.B Delaunay Triangulation



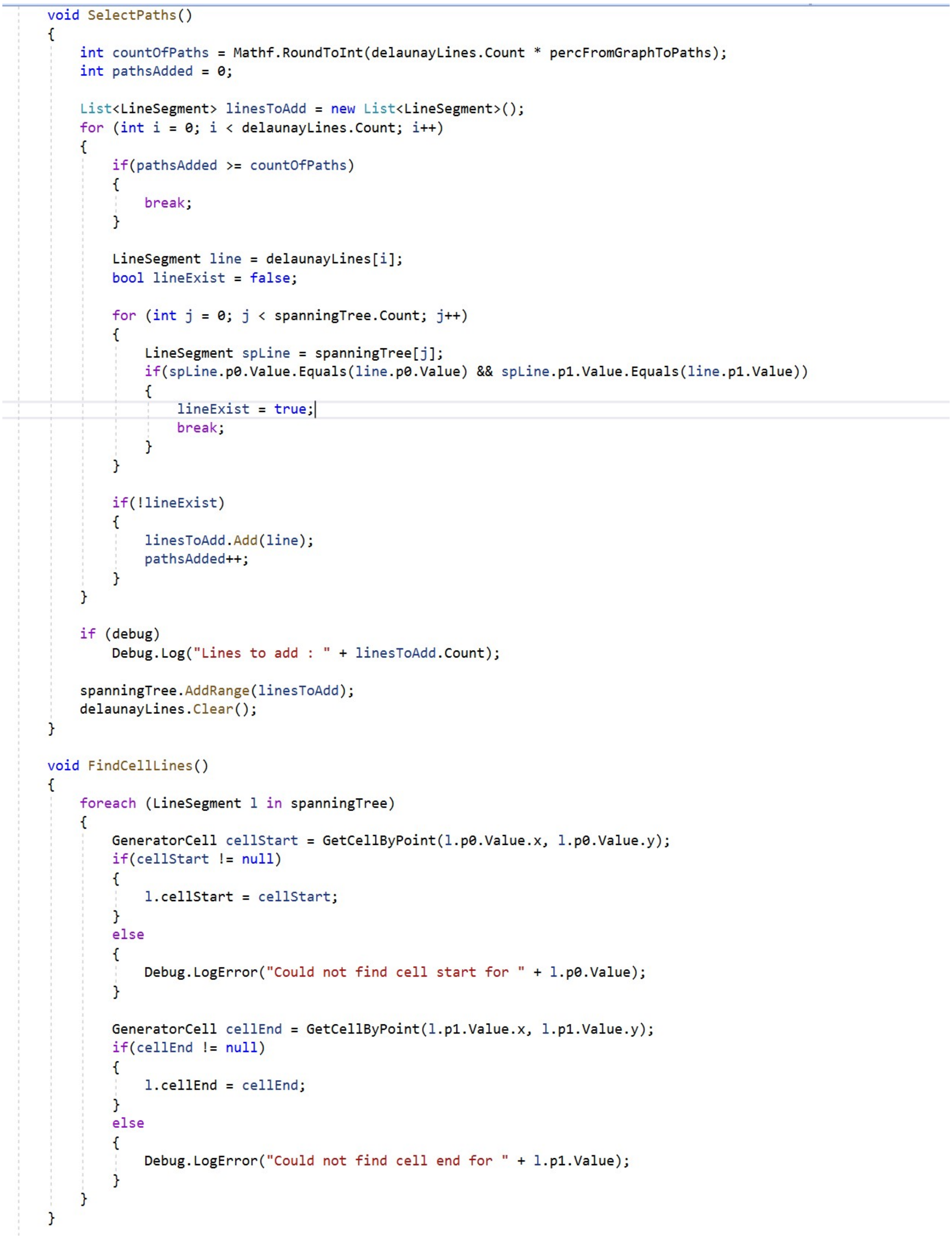
```
void CreateCells()
{
    RandomFromDistribution.ConfidenceLevel_e conf_level = RandomFromDistribution.ConfidenceLevel_e._80;

    int numberOfCells = levelStats.numberOfCells;
    float roomCircleRadius = levelStats.roomCircleRadius;
    percFromGraphToPaths = levelStats.percFromGraphToPaths;
    mainRoomMeanCutoff = levelStats.mainRoomCutoff;

    float cellMinWidth = levelStats.cellMinWidth;
    float cellMaxWidth = levelStats.cellMaxWidth;
    float cellMinHeight = levelStats.cellMinHeight;
    float cellMaxHeight = levelStats.cellMaxHeight;

    for (int i = 0; i < numberOfCells; i++)
    {
        float minWidthScalar = cellMinWidth;
        float maxWidthScalar = cellMaxWidth;
        float minHeightScalar = cellMinHeight;
        float maxHeightScalar = cellMaxHeight;

        GeneratorCell cell = new GeneratorCell();
        cell.width = Mathf.RoundToInt(RandomFromDistribution.RandomRangeNormalDistribution(minWidthScalar, maxWidthScalar, conf_level));
        cell.height = Mathf.RoundToInt(RandomFromDistribution.RandomRangeNormalDistribution(minHeightScalar, maxHeightScalar, conf_level));


        Vector2 pos = GetRandomPointInCirlce(roomCircleRadius);
        cell.x = Mathf.RoundToInt(pos.x);
        cell.y = Mathf.RoundToInt(pos.y);
        cell.index = i;
        cells.Add(cell);
        widthAvg += cell.width;
        heightAvg += cell.height;
    }

    widthAvg /= cells.Count;
    heightAvg /= cells.Count;
}
```



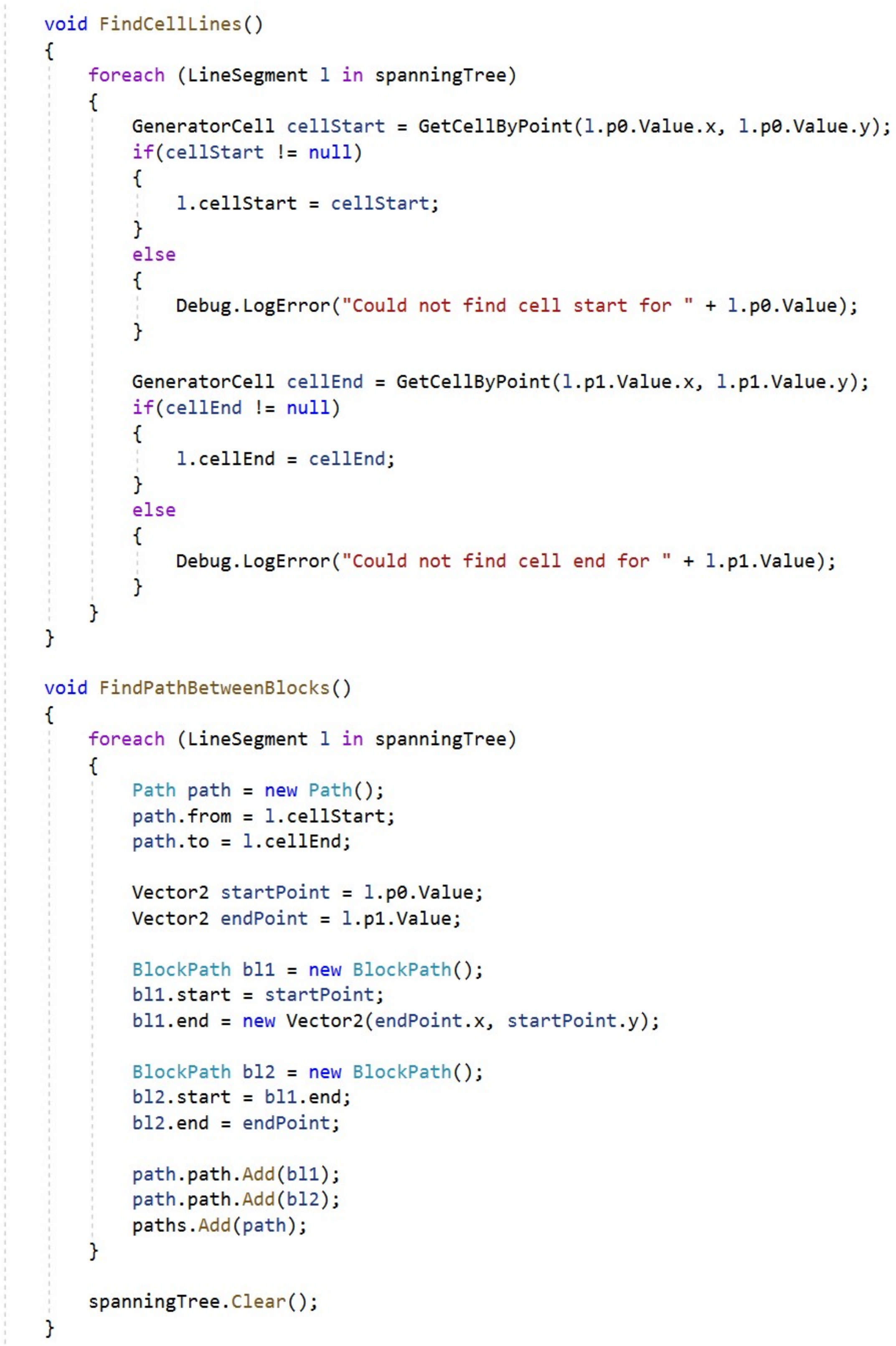
```
void SeparateCells()
{
    bool cellCollision = true;
    int loop = 0;
    while(cellCollision)
    {
        loop++;
        cellCollision = false;
        if(debug)
        {
//            Debug.Log("Loop " + loop);
        }

        for (int i = 0; i < cells.Count; i++)
        {
            GeneratorCell c = cells[i];

            for (int j = i + 1; j < cells.Count; j++)
            {
                GeneratorCell cb = cells[j];
                if(c.CollidesWith(cb))
                {
                    cellCollision = true;

                    int cb_x = Mathf.RoundToInt((c.x + c.width) - cb.x);
                    int c_x = Mathf.RoundToInt((cb.x + cb.width) - c.x);

                    int cb_y = Mathf.RoundToInt((c.y + c.height) - cb.y);
                    int c_y = Mathf.RoundToInt((cb.y + cb.height) - c.y);

                    if (c_x < cb_x)
                    {
                        if (c_x < c_y)
                        {
                            c.Shift(c_x, 0);
                        }
                        else
                        {
                            c.Shift(0, c_y);
                        }
                    }
                    else
                    {
                        if (cb_x < cb_y)
                        {
                            cb.Shift(cb_x, 0);
                        }
                        else
                        {
                            cb.Shift(0, cb_y);
                        }
                    }
                }
            }
        }
    }
}
```



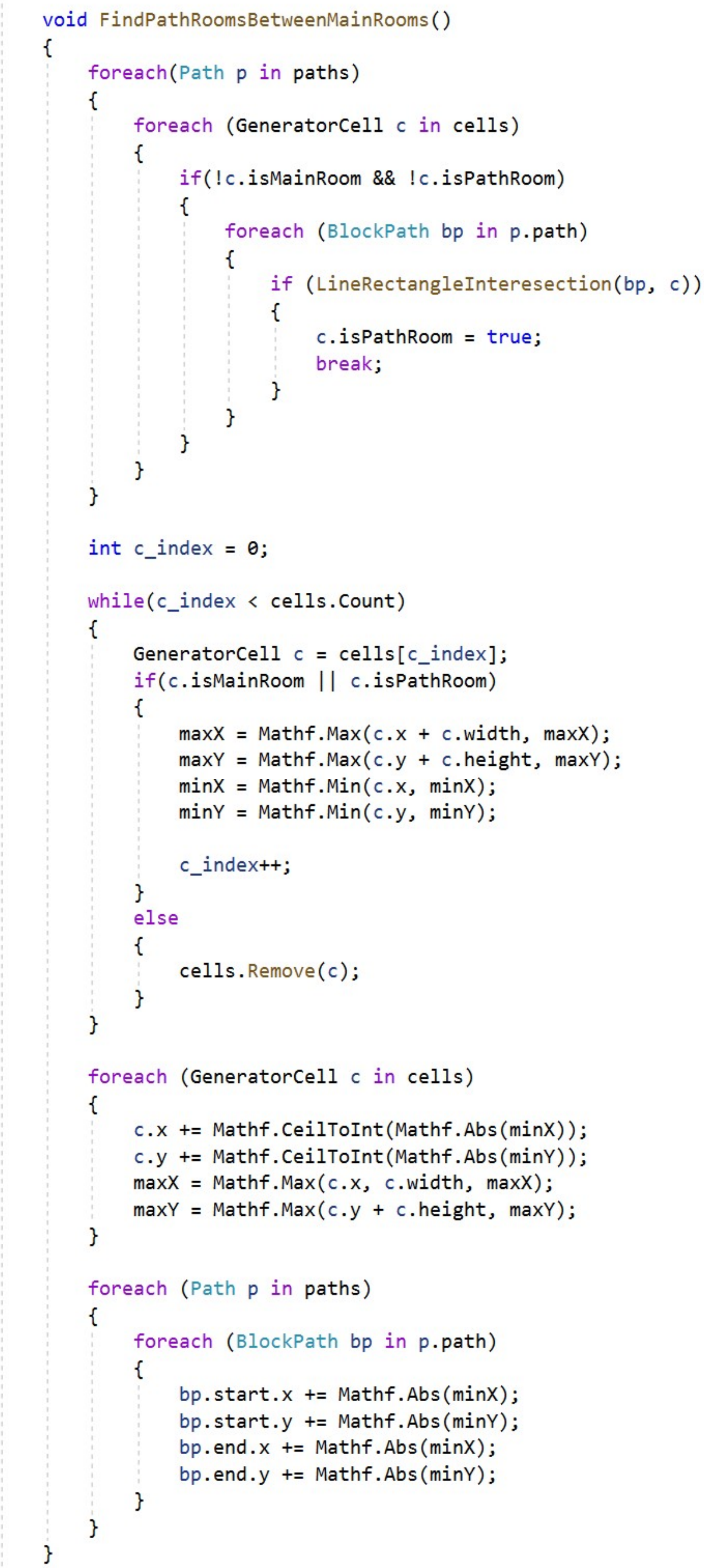
```
void PickMainRooms()
{
    foreach (GeneratorCell c in cells)
    {
        if(c.width * mainRoomMeanCutoff < widthAvg || c.height * mainRoomMeanCutoff < heightAvg)
        {
            c.isMainRoom = false;
        }
    }
}

void Triangulate()
{
    List<Vector2> points = new List<Vector2>();
    List<uint> colors = new List<uint>();

    Vector2 min = Vector2.positiveInfinity;
    Vector2 max = Vector2.zero;

    foreach (GeneratorCell c in  cells)
    {
        if (c.isMainRoom)
        {
            colors.Add(0);
            points.Add(new Vector2(c.x + (c.width / 2), c.y + (c.height / 2)));
            min.x = Mathf.Min(c.x, min.x);
            min.y = Mathf.Min(c.y, min.y);

            max.x = Mathf.Max(c.x, max.x);
            max.y = Mathf.Max(c.y, max.y);
        }
    }

    Voronoi v = new Voronoi(points, colors, new Rect(min.x, min.y, max.x, max.y));
    delaunayLines = v.DelaunayTriangulation();
    spanningTree = v.SpanningTree(KruskalType.MINIMUM);
}
```



```
void SelectPaths()
{
    int countOfPaths = Mathf.RoundToInt(delaunayLines.Count * percFromGraphToPaths);
    int pathsAdded = 0;

    List<LineSegment> linesToAdd = new List<LineSegment>();
    for (int i = 0; i < delaunayLines.Count; i++)
    {
        if(pathsAdded >= countOfPaths)
        {
            break;
        }

        LineSegment line = delaunayLines[i];
        bool lineExist = false;

        for (int j = 0; j < spanningTree.Count; j++)
        {
            LineSegment spLine = spanningTree[j];
            if(spLine.p0.Value.Equals(line.p0.Value) && spLine.p1.Value.Equals(line.p1.Value))
            {
                lineExist = true;
                break;
            }
        }

        if(!lineExist)
        {
            linesToAdd.Add(line);
            pathsAdded++;
        }
    }

    if (debug)
        Debug.Log("Lines to add : " + linesToAdd.Count);

    spanningTree.AddRange(linesToAdd);
    delaunayLines.Clear();
}

void FindCellLines()
{
    foreach (LineSegment l in spanningTree)
    {
        GeneratorCell cellStart = GetCellByPoint(l.p0.Value.x, l.p0.Value.y);
        if(cellStart != null)
        {
            l.cellStart = cellStart;
        }
        else
        {
            Debug.LogError("Could not find cell start for " + l.p0.Value);
        }

        GeneratorCell cellEnd = GetCellByPoint(l.p1.Value.x, l.p1.Value.y);
        if(cellEnd != null)
        {
            l.cellEnd = cellEnd;
        }
        else
        {
            Debug.LogError("Could not find cell end for " + l.p1.Value);
        }
    }
}
```



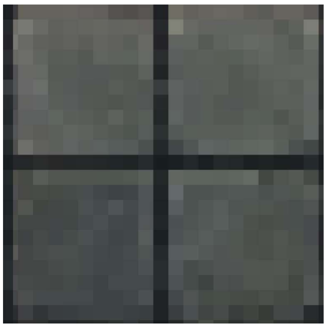
```
void FindCellLines()
{
    foreach (LineSegment l in spanningTree)
    {
        GeneratorCell cellStart = GetCellByPoint(l.p0.Value.x, l.p0.Value.y);
        if(cellStart != null)
        {
            l.cellStart = cellStart;
        }
        else
        {
            Debug.LogError("Could not find cell start for " + l.p0.Value);
        }

        GeneratorCell cellEnd = GetCellByPoint(l.p1.Value.x, l.p1.Value.y);
        if(cellEnd != null)
        {
            l.cellEnd = cellEnd;
        }
        else
        {
            Debug.LogError("Could not find cell end for " + l.p1.Value);
        }
    }
}

void FindPathBetweenBlocks()
{
    foreach (LineSegment l in spanningTree)
    {
        Path path = new Path();
        path.from = l.cellStart;
        path.to = l.cellEnd;

        Vector2 startPoint = l.p0.Value;
        Vector2 endPoint = l.p1.Value;

        BlockPath bl1 = new BlockPath();
        bl1.start = startPoint;
        bl1.end = new Vector2(endPoint.x, startPoint.y);

        BlockPath bl2 = new BlockPath();
        bl2.start = bl1.end;
        bl2.end = endPoint;

        path.path.Add(bl1);
        path.path.Add(bl2);
        paths.Add(path);
    }

    spanningTree.Clear();
}
```



```
void FindPathRoomsBetweenMainRooms()
{
    foreach(Path p in paths)
    {
        foreach (GeneratorCell c in cells)
        {
            if(!c.isMainRoom && !c.isPathRoom)
            {
                foreach (BlockPath bp in p.path)
                {
                    if (LineRectangleInteresection(bp, c))
                    {
                        c.isPathRoom = true;
                        break;
                    }
                }
            }
        }
    }

    int c_index = 0;

    while(c_index < cells.Count)
    {
        GeneratorCell c = cells[c_index];
        if(c.isMainRoom || c.isPathRoom)
        {
            maxX = Mathf.Max(c.x + c.width, maxX);
            maxY = Mathf.Max(c.y + c.height, maxY);
            minX = Mathf.Min(c.x, minX);
            minY = Mathf.Min(c.y, minY);

            c_index++;
        }
        else
        {
            cells.Remove(c);
        }
    }

    foreach (GeneratorCell c in cells)
    {
        c.x += Mathf.CeilToInt(Mathf.Abs(minX));
        c.y += Mathf.CeilToInt(Mathf.Abs(minY));
        maxX = Mathf.Max(c.x, c.width, maxX);
        maxY = Mathf.Max(c.y + c.height, maxY);
    }

    foreach (Path p in paths)
    {
        foreach (BlockPath bp in p.path)
        {
            bp.start.x += Mathf.Abs(minX);
            bp.start.y += Mathf.Abs(minY);
            bp.end.x += Mathf.Abs(minX);
            bp.end.y += Mathf.Abs(minY);
        }
    }
}
```

F Game Assets

All game assets were created by myself.



G Evaluation

G.B Evaluation Document

Edinburgh Napier University Research Consent Form

THE DEVELOPMENT OF PROCEDURAL MAP GENERATION
WITHIN GAMES

Edinburgh Napier University requires that all persons who participate in research studies give their written consent to do so. Please read the following and sign it if you agree with what it says.

1. I freely and voluntarily consent to be a participant in the research project on the topic of The Development of Procedural Map Generation Within Games to be conducted by Arran Smedley, who is an undergraduate/postgraduate student/staff member at Edinburgh Napier University.
2. The broad goal of this research study is to explore three different procedural map generation techniques and to provide input. Specifically, I have been asked to provide a qualitative approach to my evaluation that will aid me into deciding which procedural map generation is the best to use for 2D dungeon generation which should take no longer than 15 minutes to complete.
3. I have been told that my responses will be anonymised. My name will not be linked with the research materials, and I will not be identified or identifiable in any report subsequently produced by the researcher.
4. I also understand that if at any time during the session I feel unable or unwilling to continue, I am free to leave. That is, my participation in this study is completely voluntary, and I may withdraw from it without negative consequences. However, after data has been anonymised or after publication of results it will not be possible for my data to be removed as it would be untraceable at this point.
5. In addition, should I not wish to answer any particular question or questions, I am free to decline.
6. I have been given the opportunity to ask questions regarding the session and my questions have been answered to my satisfaction.
7. I have read and understand the above and consent to participate in this study. My signature is not a waiver of any legal rights. Furthermore, I understand that I will be able to keep a copy of the informed consent form for my records.

Participant's Signature　　　　　　　　　　Date

I have explained and defined in detail the research procedure in which the respondent has consented to participate. Furthermore, I will retain one copy of the informed consent form for my records.

Researcher's Signature   Date

How to Play

WASD – Move Left, Right, Up, Down.
Mouse to select a map.
Backspace to go back to menu.

Questionnaire

7. What was your first Initial thought of each procedurally generated map? (Write a brief paragraph)

Cellular Automata	
Perlin Noise	
Delaunay Triangulation	

8. Which map provided a more complete experience (Ease of exploration, Interesting dungeon patterns)? (Provide a best, 2nd best and least).

Cellular Automata	
Perlin Noise	
Delaunay Triangulation	

9. Which map had the longest load time? (Provide a longest, 2nd longest and fastest)

Cellular Automata	
Perlin Noise	
Delaunay Triangulation	

10. Which map resembles a dungeon crawler game the best? (Tick the selected map)

Cellular Automata	
Perlin Noise	
Delaunay Triangulation	

11. Which map provided the best randomisation? (Number of Corridors/Rooms)

Cellular Automata	
Perlin Noise	
Delaunay Triangulation	

12. In your own words what could you do to improve each map? (More rooms, walls around map, more accessible areas)

Cellular Automata	
Perlin Noise	
Delaunay Triangulation	

G.C Signed Consent Forms

Edinburgh Napier University Research Consent Form

THE DEVELOPMENT OF PROCEDURAL MAP GENERATION
WITHIN GAMES

Edinburgh Napier University requires that all persons who participate in research studies give their written consent to do so. Please read the following and sign it if you agree with what it says.

1. I freely and voluntarily consent to be a participant in the research project on the topic of The Development of Procedural Map Generation Within Games to be conducted by Arran Smedley, who is an undergraduate/postgraduate student/staff member at Edinburgh Napier University.
2. The broad goal of this research study is to explore three different procedural map generation techniques and to provide input. Specifically, I have been asked to provide a qualitative approach to my evaluation that will aid me into deciding which procedural map generation is the best to use for 2D dungeon generation which should take no longer than 15 minutes to complete.
3. I have been told that my responses will be anonymised. My name will not be linked with the research materials, and I will not be identified or identifiable in any report subsequently produced by the researcher.
4. I also understand that if at any time during the session I feel unable or unwilling to continue, I am free to leave. That is, my participation in this study is completely voluntary, and I may withdraw from it without negative consequences. However, after data has been anonymised or after publication of results it will not be possible for my data to be removed as it would be untraceable at this point.
5. In addition, should I not wish to answer any particular question or questions, I am free to decline.
6. I have been given the opportunity to ask questions regarding the session and my questions have been answered to my satisfaction.
7. I have read and understand the above and consent to participate in this study. My signature is not a waiver of any legal rights. Furthermore, I understand that I will be able to keep a copy of the informed consent form for my records.

23/03/2021

Participant's Signature Date

I have explained and defined in detail the research procedure in which the respondent has consented to participate. Furthermore, I will retain one copy of the informed consent form for my records.

23/03/2021

Researcher's Signature

Date

Edinburgh Napier University Research Consent Form

THE DEVELOPMENT OF PROCEDURAL MAP GENERATION
WITHIN GAMES

Edinburgh Napier University requires that all persons who participate in research studies give their written consent to do so. Please read the following and sign it if you agree with what it says.

1. I freely and voluntarily consent to be a participant in the research project on the topic of The Development of Procedural Map Generation Within Games to be conducted by Arran Smedley, who is an undergraduate/postgraduate student/staff member at Edinburgh Napier University.
2. The broad goal of this research study is to explore three different procedural map generation techniques and to provide input. Specifically, I have been asked to provide a qualitative approach to my evaluation that will aid me into deciding which procedural map generation is the best to use for 2D dungeon generation which should take no longer than 15 minutes to complete.
3. I have been told that my responses will be anonymised. My name will not be linked with the research materials, and I will not be identified or identifiable in any report subsequently produced by the researcher.
4. I also understand that if at any time during the session I feel unable or unwilling to continue, I am free to leave. That is, my participation in this study is completely voluntary, and I may withdraw from it without negative consequences. However, after data has been anonymised or after publication of results it will not be possible for my data to be removed as it would be untraceable at this point.
5. In addition, should I not wish to answer any particular question or questions, I am free to decline.
6. I have been given the opportunity to ask questions regarding the session and my questions have been answered to my satisfaction.
7. I have read and understand the above and consent to participate in this study. My signature is not a waiver of any legal rights. Furthermore, I understand that I will be able to keep a copy of the informed consent form for my records.

23/03/2021

Participant's Signature : X

Calum Macpherson

Date:

23/3/21

I have explained and defined in detail the research procedure in which the respondent has consented to participate. Furthermore, I will retain one copy of the informed consent form for my records.

23/03/2021

Researcher's Signature Date

Edinburgh Napier University Research Consent Form

THE DEVELOPMENT OF PROCEDURAL MAP GENERATION
WITHIN GAMES

Edinburgh Napier University requires that all persons who participate in research studies give their written consent to do so. Please read the following and sign it if you agree with what it says.

1. I freely and voluntarily consent to be a participant in the research project on the topic of The Development of Procedural Map Generation Within Games to be conducted by Arran Smedley, who is an undergraduate/postgraduate student/staff member at Edinburgh Napier University.
2. The broad goal of this research study is to explore three different procedural map generation techniques and to provide input. Specifically, I have been asked to provide a qualitative approach to my evaluation that will aid me into deciding which procedural map generation is the best to use for 2D dungeon generation which should take no longer than 15 minutes to complete.
3. I have been told that my responses will be anonymised. My name will not be linked with the research materials, and I will not be identified or identifiable in any report subsequently produced by the researcher.
4. I also understand that if at any time during the session I feel unable or unwilling to continue, I am free to leave. That is, my participation in this study is completely voluntary, and I may withdraw from it without negative consequences. However, after data has been anonymised or after publication of results it will not be possible for my data to be removed as it would be untraceable at this point.
5. In addition, should I not wish to answer any particular question or questions, I am free to decline.
6. I have been given the opportunity to ask questions regarding the session and my questions have been answered to my satisfaction.
7. I have read and understand the above and consent to participate in this study. My signature is not a waiver of any legal rights. Furthermore, I understand that I will be able to keep a copy of the informed consent form for my records.

Participant's Signature Stephen Daniels Date 24/03/2021

I have explained and defined in detail the research procedure in which the respondent has consented to participate. Furthermore, I will retain one copy of the informed consent form for my records.

23/03/2021

Researcher's Signature Date

Edinburgh Napier University Research Consent Form

THE DEVELOPMENT OF PROCEDURAL MAP GENERATION
WITHIN GAMES

Edinburgh Napier University requires that all persons who participate in research studies give their written consent to do so. Please read the following and sign it if you agree with what it says.

1. I freely and voluntarily consent to be a participant in the research project on the topic of The Development of Procedural Map Generation Within Games to be conducted by Arran Smedley, who is an undergraduate/postgraduate student/staff member at Edinburgh Napier University.
2. The broad goal of this research study is to explore three different procedural map generation techniques and to provide input. Specifically, I have been asked to provide a qualitative approach to my evaluation that will aid me into deciding which procedural map generation is the best to use for 2D dungeon generation which should take no longer than 15 minutes to complete.
3. I have been told that my responses will be anonymised. My name will not be linked with the research materials, and I will not be identified or identifiable in any report subsequently produced by the researcher.
4. I also understand that if at any time during the session I feel unable or unwilling to continue, I am free to leave. That is, my participation in this study is completely voluntary, and I may withdraw from it without negative consequences. However, after data has been anonymised or after publication of results it will not be possible for my data to be removed as it would be untraceable at this point.
5. In addition, should I not wish to answer any particular question or questions, I am free to decline.
6. I have been given the opportunity to ask questions regarding the session and my questions have been answered to my satisfaction.
7. I have read and understand the above and consent to participate in this study. My signature is not a waiver of any legal rights. Furthermore, I understand that I will be able to keep a copy of the informed consent form for my records.

Participant's Signature CALUM MATHISON Date 24/03/2021

I have explained and defined in detail the research procedure in which the respondent has consented to participate. Furthermore, I will retain one copy of the informed consent form for my records.

23/03/2021

Researcher's Signature Date

Edinburgh Napier University Research Consent Form

THE DEVELOPMENT OF PROCEDURAL MAP GENERATION
WITHIN GAMES

Edinburgh Napier University requires that all persons who participate in research studies give their written consent to do so. Please read the following and sign it if you agree with what it says.

1. I freely and voluntarily consent to be a participant in the research project on the topic of The Development of Procedural Map Generation Within Games to be conducted by Arran Smedley, who is an undergraduate/postgraduate student/staff member at Edinburgh Napier University.
2. The broad goal of this research study is to explore three different procedural map generation techniques and to provide input. Specifically, I have been asked to provide a qualitative approach to my evaluation that will aid me into deciding which procedural map generation is the best to use for 2D dungeon generation which should take no longer than 15 minutes to complete.
3. I have been told that my responses will be anonymised. My name will not be linked with the research materials, and I will not be identified or identifiable in any report subsequently produced by the researcher.
4. I also understand that if at any time during the session I feel unable or unwilling to continue, I am free to leave. That is, my participation in this study is completely voluntary, and I may withdraw from it without negative consequences. However, after data has been anonymised or after publication of results it will not be possible for my data to be removed as it would be untraceable at this point.
5. In addition, should I not wish to answer any particular question or questions, I am free to decline.
6. I have been given the opportunity to ask questions regarding the session and my questions have been answered to my satisfaction.
7. I have read and understand the above and consent to participate in this study. My signature is not a waiver of any legal rights. Furthermore, I understand that I will be able to keep a copy of the informed consent form for my records.

23/03/2021

Participant's Signature Date

I have explained and defined in detail the research procedure in which the respondent has consented to participate. Furthermore, I will retain one copy of the informed consent form for my records.

23/03/2021

Researcher's Signature Date

Edinburgh Napier University Research Consent Form

THE DEVELOPMENT OF PROCEDURAL MAP GENERATION
WITHIN GAMES

Edinburgh Napier University requires that all persons who participate in research studies give their written consent to do so. Please read the following and sign it if you agree with what it says.

1. I freely and voluntarily consent to be a participant in the research project on the topic of The Development of Procedural Map Generation Within Games to be conducted by Arran Smedley, who is an undergraduate/postgraduate student/staff member at Edinburgh Napier University.
2. The broad goal of this research study is to explore three different procedural map generation techniques and to provide input. Specifically, I have been asked to provide a qualitative approach to my evaluation that will aid me into deciding which procedural map generation is the best to use for 2D dungeon generation which should take no longer than 15 minutes to complete.
3. I have been told that my responses will be anonymised. My name will not be linked with the research materials, and I will not be identified or identifiable in any report subsequently produced by the researcher.
4. I also understand that if at any time during the session I feel unable or unwilling to continue, I am free to leave. That is, my participation in this study is completely voluntary, and I may withdraw from it without negative consequences. However, after data has been anonymised or after publication of results it will not be possible for my data to be removed as it would be untraceable at this point.
5. In addition, should I not wish to answer any particular question or questions, I am free to decline.
6. I have been given the opportunity to ask questions regarding the session and my questions have been answered to my satisfaction.
7. I have read and understand the above and consent to participate in this study. My signature is not a waiver of any legal rights. Furthermore, I understand that I will be able to keep a copy of the informed consent form for my records.

Hazel Chiu

Participant's Signature

23/03/2021

Date

I have explained and defined in detail the research procedure in which the respondent has consented to participate. Furthermore, I will retain one copy of the informed consent form for my records.

Researcher's Signature

23/03/2021

Date