# Procedural Content Generation

# 2D Map Generator

Author: Richard Alexander

Student ID: 19270151

Supervisor: Simon Colreavy

B.Sc. Computer Games Development

Department of Computer Science and Information Systems

Academic Year – 2022/2023

Word count: 10333

# Contents

## Abstract

The main objective of this project is to develop a tool that demonstrates the power and flexibility of procedural content generation techniques. To achieve this, I decided to develop a 2D map generator, which allows users to generate new maps and customize various map parameters to achieve different map layouts and structures quickly and easily.

The 2D map generator offers a user-friendly interface that provides multiple options for users to manipulate certain variables that affect the generated map. These variables include map size, saturation, distance and more. By changing these variables, users can create a virtually unlimited number of maps with varying degrees of complexity which can also be saved and loaded.

By using the 2D map generator, users can explore the vast possibilities of procedural content generation and learn about the different algorithms used in map generation. A user could also potentially use the 2D map generator for their own purposes, such as for developing maps in their own video games or for designing dungeons in tabletop games.

# Introduction

## What is PCG?

Procedural content generation (PCG) can be defined as "the algorithmic creation of game content with limited or indirect user input (Shaker, Togelius and Nelson, 2016). While this definition specifically addresses game content, PCG does however have much broader applications, but this definition fits well within the scope of this project. PCG can be used to generate a variety of different game content, examples of such are levels, maps, textures, stories, items, quests, music, etc.

## Why use PCG?

There are several reasons as to why one might use PCG. In contemporary game development, players' expectations are higher than ever, making PCG increasingly essential, since PCG removes the need to have a human designer to hand create content, it allows game developers to create vast, unique, and immersive game worlds that are both time and cost-effective to develop while still preserving the quality of the content. Additionally, PCG can make different aspects of games more varied and unpredictable and increase its replay ability, thus enhancing the overall gaming experience. One other interesting consequence of PCG is that it has opened the door to new types of games that could not be possible previously, if you can create game content just as fast as it is being consumed, then you can create games that do not have an ending, additionally newly generated content can be tailored to specific needs of the game or player. PCG also allows creators to be more creative, as algorithmically generating content can allow for more focus on the creative process. (Shaker, Togelius and Nelson, 2016)

## The history of PCG

In current times, PCG has been employed in numerous games, ranging from small indie titles to large AAA productions. However, the history of PCG dates back to the early days of computing and video games. The first few examples of using PCG to generate content emerged in the 1960s, one of the first such examples being "ELIZA" by Joseph Weizenbaum (Weizenbaum, 1983), which was a natural language processing program that could simulate real conversations with users.  Another very famous example was the zero-player game titled "The Game of Life" by John Conway (Gardner, 1970), which was based on cellular automaton. The only input required from a user was defining its initial state where the game would then continuously evolve, demonstrating emergent complexity through simple rules.

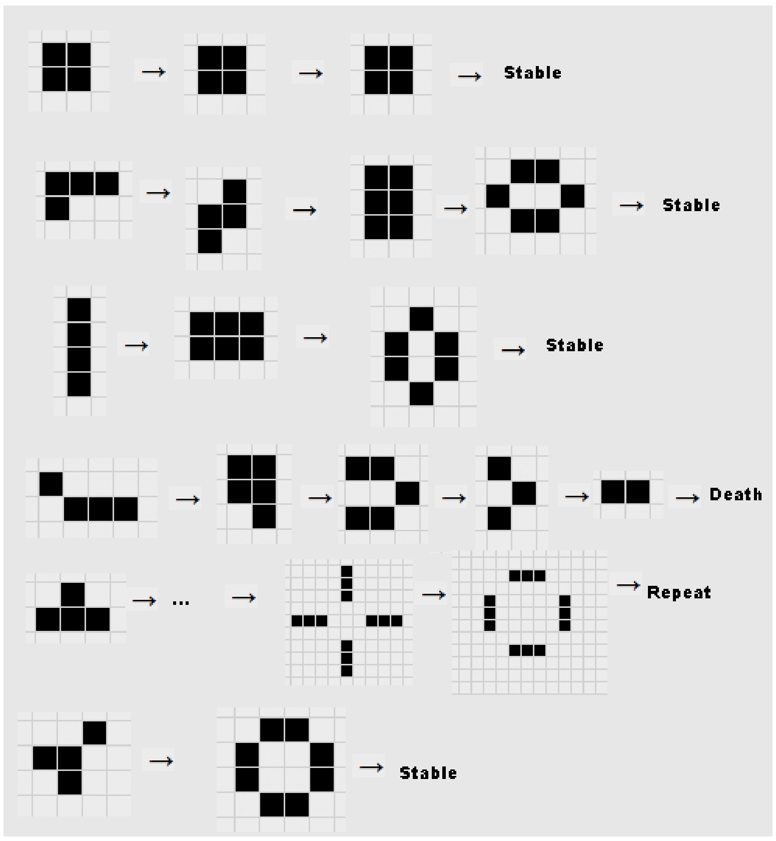


*Figure 1: Conway's Game of Life (Cornell University, no date)*

PCG began to truly flourish and make its mark in game development in the 1980s with the release of the game “Rogue” (1980) (Smith, 2021)  which introduced procedurally generated levels, items, and creatures. Rogue was originally heavily inspired by dungeons and dragons which is tabletop game that utilized the concept of using the randomness of a dice roll to generate dungeons. The game became so influential that it spawned the "Roguelike" subgenre into the video game world, which was a genre of game characterised by the main features of rogue including permadeath and procedurally generated content. However, the capabilities of the hardware at the time limited what the developers could do and so the graphics in most of these types of game lacked visual appeal.

Continuing on throughout the 1990s and 2000s, PCG techniques began to become much more sophisticated and led to the development of more advanced ‘noise’ algorithms such as Perlin Noise (1983) (Perlin, 1985) and Simplex Noise (2001) (Gustavson, 2005). These algorithms were mainly used to increase the realism in procedurally generated terrain and textures, contributing in making game worlds feel more natural.

These same advancements would later be implemented into games to enhance their level of PCG, this led to the development of much more advanced PCG based games such as in "Minecraft" (Mojang, 2009). Minecraft is a survival sandbox game where players can explore a seemingly endless world fighting to survive in a world full of interestingly formed landscapes and caves that are full of animals, and enemies that are all procedurally generated. Minecraft still is one of the most popular games today and was one of the largest games to introduce and demonstrate to players the potential and versatility of PCG for creating immersive game worlds.

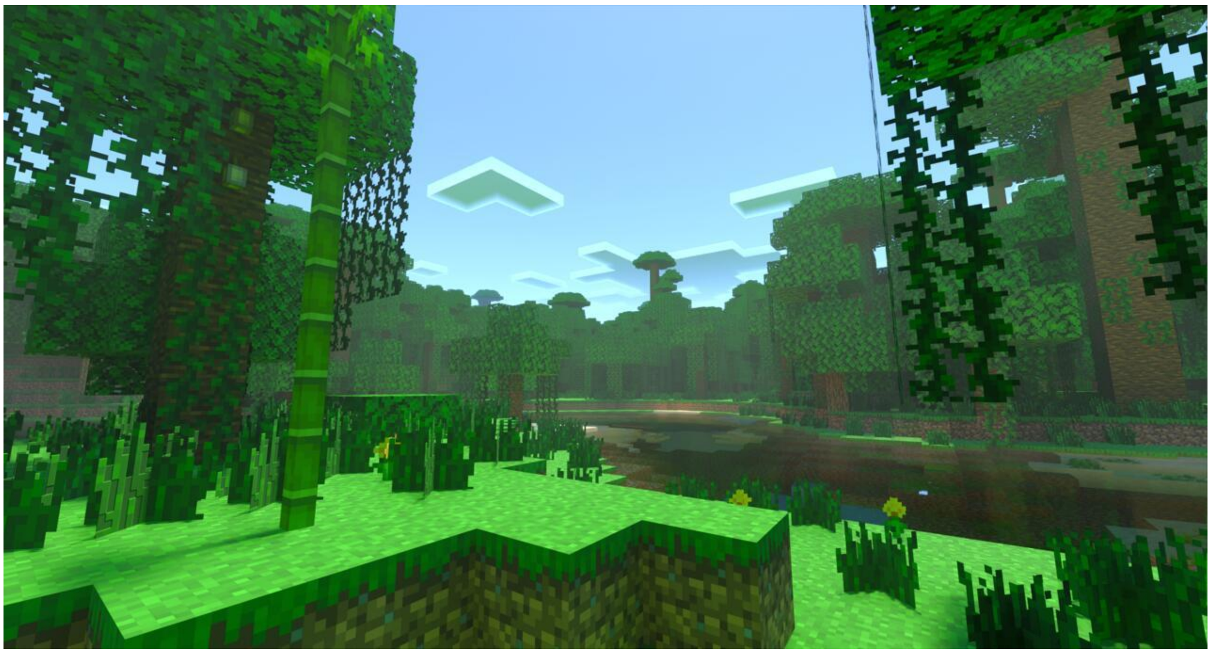


*Figure 2: Minecraft jungle (Eric Frederiksen, 2021)*

In more recent years, PCG has managed to continue to evolve thanks to the advancements made in the areas of machine learning and artificial intelligence to create more unique experiences for players (Summerville *et al.*, 2018). Alongside the integration of machine learning and artificial intelligence concepts into PCG, the advancements made in graphical hardware has allowed developers to push the envelope of what was previously possible enabling the generation of more realistic, complex and adaptive content in games.

## The future of PCG

PCG has had a surge of academic research recently with researchers from various fields all contributing to new perspectives and ideas to the current problems and challenges faced in PCG. This being said, it is useful to have an idea of where the future of PCG is going and what problems need to be overcome in the field of PCG. Below is a list of what the current research outlines as the potential future of PCG, some of these are things we cannot right now do with the current technology or that may never be things that will never be achieved (Smith, 2021):

## Multi-level Multi-content PCG

This is the idea of having a content generator that could at the push of a button generate all of the content necessary for a given game and ensuring the content would be of high quality and would seamlessly fit together. The type of world that would be generated by such a tool would be a fully complete and entirely fleshed out world including all necessary features and content the game would need, the generator would also produce variation in content between different game generations. Current approaches to creating this type of generation often solely focuses on a specific type of content for a single game, an example of this is Dwarf Fortress (Bay 12 Games, 2006) while dwarf fortress is able to generate multiple aspects of its game world, the result is still limited in variation. While a fully-fledged content generation system may not be achievable in the foreseeable future, there is potential for progress in this direction. (Julian Togelius *et al.*, 2013)

*Figure 3: Dwarf Fortress, old vs new graphics (KEVIN PURDY, 2022)*

## PCG-based Game Design

PCG based game design is the concept of having aa game that could not exist without PCG as it is an absolutely essential part of the game's existence. If you took away the PCG element from such a game, then there would be nothing recognisable left of the game to observe. Current games that simulate this idea are still only variations of well-known genres, such as games that feature endless levels, rather than examples of PCG game design since the core parts of the game could still function without PCG (Julian Togelius *et al.*, 2013).

## Generating Complete Games

This is different from Multi-level Multi-content PCG as not only are you generating all of the content needed for a game with a specific set of rules, but also you are generating the entire game itself. This means the system would be capable of generating complete and unique games each time it's ran. It would create the rules and the reward structures of the game, as well as the graphical representation and controls. There have been attempts to create game rules in the past by combining different aspects such as board games. A notable example includes Cameron Browne's Ludi system (Browne, 2008), which used evolutionary computation to create a novel board game. However, while these efforts have proved that creating game rules is possible, they only produce extremely simple games and do not remotely resemble the idea of what a complete game generator would look like (Julian Togelius *et al.*, 2013).

The future of PCG encompasses several ambitious visions, while some progress has been made in these areas, the reality of achieving these goals fully may still be distant or even unattainable. Nevertheless, continued research and development in PCG holds the potential to push the boundaries of what is currently possible and pave the way for more advanced and innovative applications in the gaming industry.

# Research

## Background

### Desirable properties of PCG

PCG methods can be seen as solutions to content generation problems, with different requirements for each application. The common desirable properties of PCG solutions include:

- Speed: The generation time may vary from milliseconds to months, depending on whether the content is generated during gameplay or game development.

- Reliability: Generators must ensure the content they produce meets specific quality criteria. The importance of reliability varies depending on the content type; for example, in an adventure game, a quest with a broken or missing objective would be a critical failure, while a non-player character (NPC) having an unusual outfit or appearance may not.

- Controllability: Content generators should be controllable to allow human users or algorithms to specify aspects of the generated content, such as the difficulty of a level or the style of a building.

- Expressivity and diversity: Generators need to produce a diverse set of content to avoid repetition and maintain player interest. Designing level generators that create diverse content without compromising quality is a complex task.

- Creativity and believability: Ideally, generated content should appear human-created, not machine-generated. There are numerous ways content can look generated, and designing PCG systems that generate believable content is a significant challenge.

Balancing these properties often involves trade-offs, such as between speed and quality or expressivity and reliability. The future of PCG research aims to develop systems that can effectively balance these properties. (Shaker, Togelius and Nelson, 2016)

## Taxonomy of PCG

Given the diverse range of PCG problems and methods currently available, it is beneficial to establish a framework that emphasizes the distinctions and commonalities among approaches. To do this it is common to sort existing approaches into a taxonomy to help understand and analyse various PCG methods.

### *Online versus offline PCG*

The first distinction in approaches, is whether the content generation is performed online during the runtime of the game, or offline during game development. Online PCG is used for generating content while the player is playing the game, this allows for real-time adaptability, creating dynamic and endless experiences for players based on their behaviour and preferences. However, online PCG requires algorithms to be extremely fast, with predictable runtime and consistent output quality to ensure a seamless gaming experience. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

On the other hand, offline PCG is used for generating content prior to the beginning of a game session or during its development. Utilizing offline content generation is especially beneficial when focussing on creating intricate elements like environments and maps. This approach provides game designers with more control and the opportunity to fine-tune the generated content, ensuring a polished final product. While offline PCG may not offer the same level of adaptability and dynamic content as online PCG, it helps create a solid foundation for the game world. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

*Generic versus adaptive PCG*

Adaptive PCG considers the players behaviour when generating content as opposed to generic which does not. It is important to note however that this is distinctly different to online versus offline, as not all online PCG is adaptive since it does not have to consider the players behaviour, however all offline PCG is generic PCG since the content has been generated before the player is involved (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018). There is argument that dimensions of control (discussed below) could be considered a form of player behaviour.

*Random Seeds Versus Parameter Vectors*

When it comes to controlling the content produced from a PCG game, there is a few ways to do this. Random seed generation is one such way. Random seeds are initial input values used to guide and control the generation process by determining the starting point of the random number used in a PCG algorithm. Different seeds will produce completely different results, however if you are reusing a seed value the same content will consistently be reproduced, allowing for reproducibility and consistency in the generated content across multiple sessions or instances of the game. This can be advantageous for sharing game experiences or levels, as players can share the seed value and recreate the same content on their separate devices.

Another aspect to consider when generating content is the extent to which the algorithm can be parameterized. Parameter vectors are used to guide and control the generation process by providing specific input values to the PCG algorithm. These input values define various aspects or properties of the generated content, ensuring that the resulting content aligns with the desired design or gameplay experience. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

*Necessary versus optional game content*

PCG can be utilized to generate both necessary and optional game content. Necessary content is required for players to progress through the game, such as traversing dungeons, defeating monsters, or adhering to essential game rules. Optional content, on the other hand, can be engaged with or avoided at the player's discretion, like choosing different weapons or entering optional locations. The critical distinction between the two is that necessary content must always be correct and balanced in terms of difficulty, while optional content can afford occasional imperfections or inconsistencies. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

Classifying whether content is either necessary or optional depends heavily on the game design and game fiction. For example, the game Borderlands (Gearbox Software, 2009) features randomly generated weapons, even if some of these weapons aren't that useful it is still important for its type of gameplay and it still stays consistent with the game's overall narrative. In contrast to this, a single unrealistic element could break immersion in a visually realistic game like Call of Duty 4: Modern Warfare (Infinity Ward, 2007). Categorizing content based on whether it is necessary or optional may vary between different game genres or even individual games, thus making it crucial to assess the content's role on a game-by-game basis. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

*Stochastic versus deterministic PCG algorithms*

Deterministic is a PCG algorithm that has options for dimensions of control (discussed above) as it is defined as an algorithm that allows for the regeneration of previously generated content when given the same starting point and method parameters, whereas with a stochastic PCG algorithm, this regeneration is not possible. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

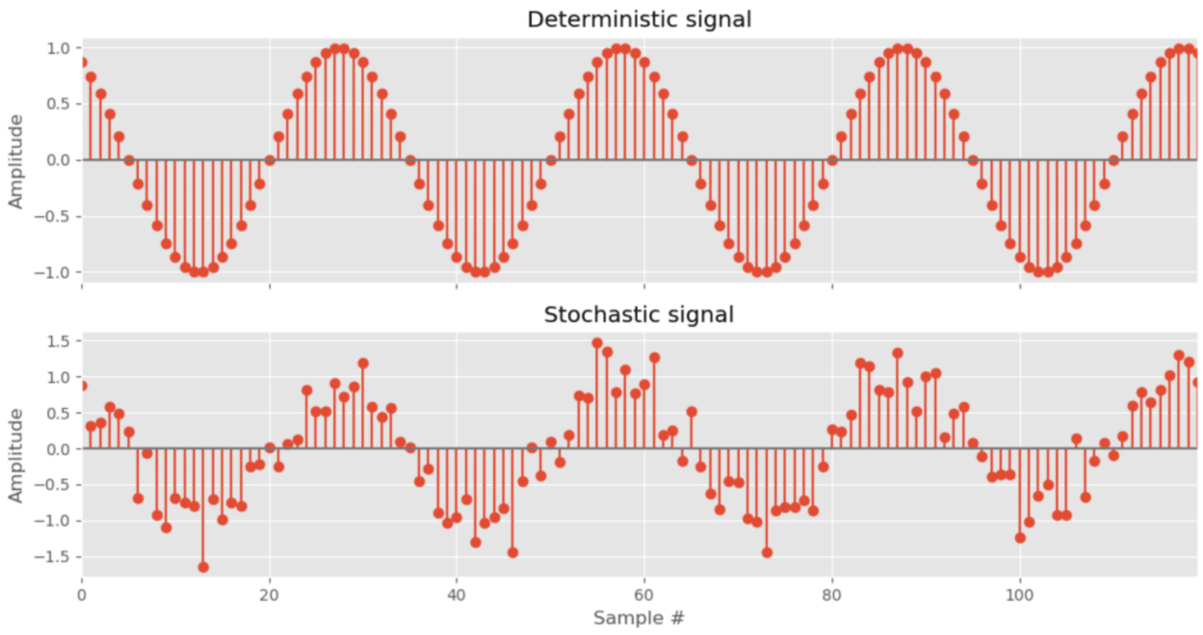


*Figure 4: deterministic and stochastic signals (Mathuranathan, 2020)*

### *Constructive versus generate-and-test*

The difference between constructive PCG algorithms and generate-and-test PCG algorithms is that constructive creates all of its content and then only has to test that the content is "good enough" whereas generate-and-test does the same process iteratively until it comes to a satisfactory solution. This means if an iteration fails the test, then the generation is discarded and regenerated, this process will repeat until the generation does not fail a test (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018). The distinction between the two is on the strictness of the testing, some use cases of PCG do not require near perfect generation, while other use cases do. This also ties in with what was discussed in the 'Necessary versus optional' section, when creating necessary content, you would want to use a generate-and-test algorithm whereas with optional content, a constructive algorithm is satisfactory.

*Mixed authorship/degrees of control*

Until recently, PCG has provided limited input opportunities for game designers, who generally directly adjust the algorithm parameters to guide the content generation in a desired path with the primary goal being the creation of endless variations of playable content. However, a new intriguing paradigm has surfaced that emphasizes the integration of designer and/or player input throughout the design process. In this mixed-initiative paradigm, a human designer or player collaborates with the algorithm to generate the desired content (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018). It is this PCG concept that heavily influenced the direction of my FYP product, resulting in the goal of my project being to create a 2D map generator, where a user can interact with an interface to change and adapt the generations directly.

## PCG algorithms

When conducting my research on the different types of PCG algorithmic approaches, it became apparent that the scope of what algorithms can be used is extremely broad and deep, with great overlap between fields. Many of the methods used have connections to other areas such as computer graphics, artificial intelligence, theoretical computer science, and even in biology.

### Search-based Methods

Search-based PCG involves the use of a search algorithm which serves as the "engine" of the search-based method and is often an evolutionary algorithm or some other stochastic search, the algorithm chosen is used to search through generated content to find sought-after attributes, it does so by iterating and tweaking potential solutions until it has arrived at a satisfactory result. In some cases, there are significant advantages to using a more sophisticated algorithm rather than just a simple one. These advanced algorithms may take constraints into consideration or be specialized for a particular content representation. This balance between simplicity and sophistication can help optimize the search process for generating content.

Search-based PCG is an example of a generate-and-test approach, however there is some slight differences. In a generic generate-and-test approach, when it comes to testing, the test function would simply accept or reject the content that has been generated, but in a search-based approach the test function grades the content with a fitness value. Producing further generations then depends upon the previously assigned fitness value. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

*Content Representation*

Another important factor when using a search-based method is content representation. In game development content representation is a critical aspect that influences the efficiency of generation algorithms and the variety of content that can be produced. As part of evolutionary algorithms, solutions are encoded as genotypes which are then converted into phenotypes, the actual entities being evolved. In gaming scenarios, the genotype can be thought of as the blueprint for creating a game level, while the phenotype is the actual game level.

Various examples of content representation in the gaming domain include indirect representation for evolving maps in real-time strategy games, evolving tracks for car racing games, and different representation methods for platform games. These representations result in different search spaces, and the choice of representation depends on the problem being solved.

An important consideration in content representation is the distinction between direct and indirect encoding. Direct encoding entails a simple computational relationship between genotype and phenotype, while indirect encodings involve a nonlinear mapping, often requiring more complex computation. The choice of representation should consider factors such as dimensionality, locality, and expressive range, as well as the ability to actually represent the solutions achieved. Being able to find a balance between direct and indirect representation is important to achieve a quality generation. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

*Evaluation Functions*

Evaluation functions play a crucial role in PCG by assessing the suitability of generated content for use in a game. A Fitness value is a scalar or vector of real numbers that represents the content's value. The challenge lies in determining what should be optimized and how to formalize it, especially when considering subjective factors of a player's experience such as fun and overall enjoyment.
There are three primary classes of evaluation functions which are direct, simulation-based, and interactive evaluation functions.

Direct evaluation functions extract features from generated content and map them to a fitness value, allowing for personalization based on player models. Direct evaluation functions are typically quick to compute and can be relatively simple to implement. However, it can be challenging to develop a direct evaluation function that effectively evaluates certain aspects of game content.

Simulation-based evaluation functions employ artificial agents to play through the content and estimate its quality. They can be static, where the agent's behaviour remains constant, or dynamic, where the agent adapts during gameplay. Dynamic functions can measure learnability, or how well and fast the agent learns to play the content being evaluated.

Interactive evaluation functions involve evaluating content based on interaction with a human player. Data can be collected either explicitly, through direct means like questionnaires, or implicitly by measuring player interactions with the content. Although explicit data collection can be accurate and reliable, it may interrupt gameplay. On the other hand, implicit data collection can be noisy and based on assumptions.

Ultimately, designing effective evaluation functions in PCG relies on understanding the specific aspects of game content that need to be optimized and the best way to formalize them, while also considering factors like player experience and personalization. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

*Evolutionary Algorithms*

Evolutionary algorithms are commonly applied to PCG problems to help create a content for a game. An evolutionary algorithm is a probabilistic optimization technique that takes inspiration from the process of natural selection in Darwinian evolution. The main concept revolves around maintaining a population of individuals, each new generation of individuals gets evaluated, and the fittest among them are allowed to reproduce, while the least fit are eliminated. This over time will produce increasingly better generations. The process works even when the initial generation is randomly generated and relatively unfit for the intended purpose, but a well-designed evaluation function will discern differences in the fitness of individuals and select the best ones out of the bunch. One example of a simple yet effective evolutionary algorithm is the μ + λ evolution strategy. Here, μ represents the population size maintained between generations, while λ represents the size generated through reproduction in each generation.

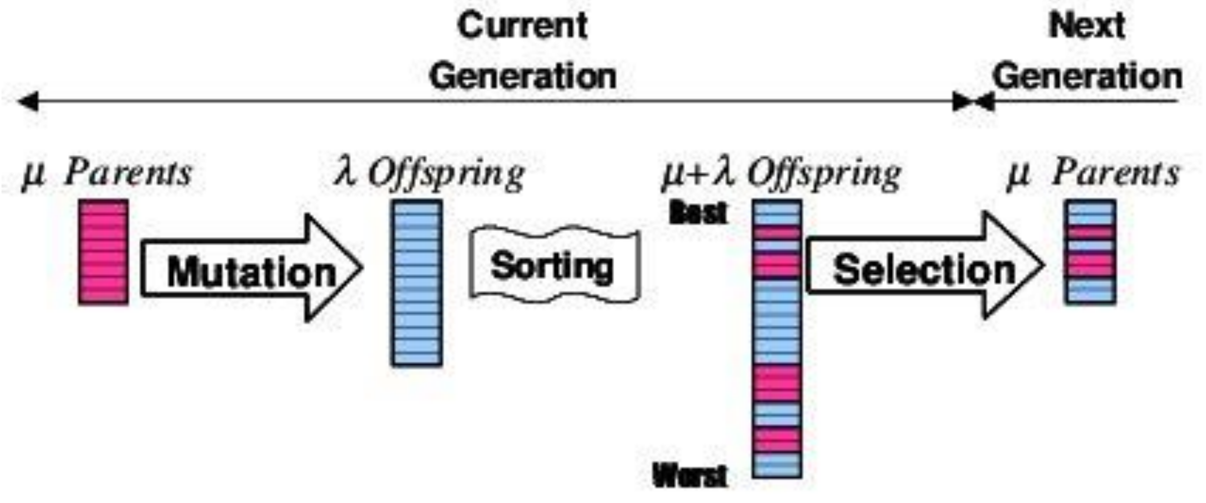


*Figure 5: The μ + λ evolution strategy (Costa and Oliveira, 2002)*

The μ + λ evolution strategy is just one of several types of evolutionary algorithms, such as genetic algorithms that rely more on recombination and use different selection mechanisms. Other stochastic search and optimization algorithms, like particle swarm optimization and ant colony optimization, can also be used for similar purposes.

When the quality of generated content cannot be assessed using a single evaluation function, multi-objective evolutionary algorithms come into play. These algorithms optimize for multiple objectives simultaneously and can identify individuals with unique combinations of strengths that are not dominated by others.

Aside from evolutionary algorithms, other search techniques, such as exhaustive search or random search, can be applied depending on the size of the search space, time constraints, or the need for diversity in generated content. However evolutionary algorithms aren't always the best solution for all cases of PCG problems as sometimes it could be considered over-engineering and a simple solution would be just as suitable. Regardless of the search method though, content representation and evaluation functions are crucial for distinguishing good content from bad. (Goldberg and Holland, 1988; Shaker, Togelius and Nelson, 2016; Yannakakis and Togelius, 2018)

### Constructive generation methods

Constructive-based methods in PCG are techniques used to create game content by following a set of predefined rules, algorithms, or procedures. These methods build the content step by step, ensuring that the output is valid and playable without the need for further evaluation or iterations.

Unlike search-based methods, which involve evaluating and refining content based on a fitness function, constructive methods generate content directly and typically run in a fixed, usually short, amount of time. This makes them particularly useful for generating content on-the-fly during gameplay or when rapid content creation is needed, however they provide limited control over the output and its properties.

Examples of constructive-based methods include cellular automata, grammars, and noise-based techniques, which are commonly used to create game content such as platform game levels, dungeons, and terrain. (Togelius *et al.*, 2011; Shaker, Togelius and Nelson, 2016)

*Perlin Noise*

Ken Perlin introduced Perlin noise in 1983 to improve the realism of computer-generated imagery (CGI), which appeared too artificial at the time. However, Perlin noise found many use cases outside of just CGI, and was especially useful for computer graphics artists to better represent the complexity of natural phenomena surfaces, such as rocks, smoke, trees etc. It does this by imitating the random appearance of textures found in nature.

Perlin noise found itself commonly being used in PCG applications because of a few properties it possess, one is its abilities is to make graphical art look more realistic which is perfect for algorithmic generation as it is easy to apply and instantly makes generated content look better. Another reason is because it does not demand much memory during creation, because of this it can be used in real time graphics.

*Figure 6: Perlin noise effect (digitalfreepen, 2017)*

However, Perlin noise has been succeeded by algorithms like fractal noise and simplex noise, which are also commonly used in real-time graphics and procedural textures in computer graphics/video games to make them more realistic.

The Perlin noise algorithm is typically implemented as a two, three, or four-dimensional function, with the process involving three steps: defining a grid of random gradient vectors, computing the dot product between the gradient vectors and their offsets, and interpolating between these values. The algorithm's complexity is O(2^n), where n represents the number of dimensions. (Perlin, 1985; Shaker, Togelius and Nelson, 2016)

*Rule-Based*

Cellular automata

A cellular automaton is composed of an n-dimensional grid, a collection of states, and a set of transition rules. Typically, cellular automata are either one-dimensional (vectors) or two-dimensional (matrices). Each cell can exist in one of multiple states; in the simplest scenario, cells can be either on or off. The initial state of a cellular automaton is determined by the distribution of cell states at the beginning of the experiment. From this point, the automaton evolves in discrete steps according to its specific rules. At each time t, each cell determines its new state based on its current state and the states of its neighbouring cells at time t-1.

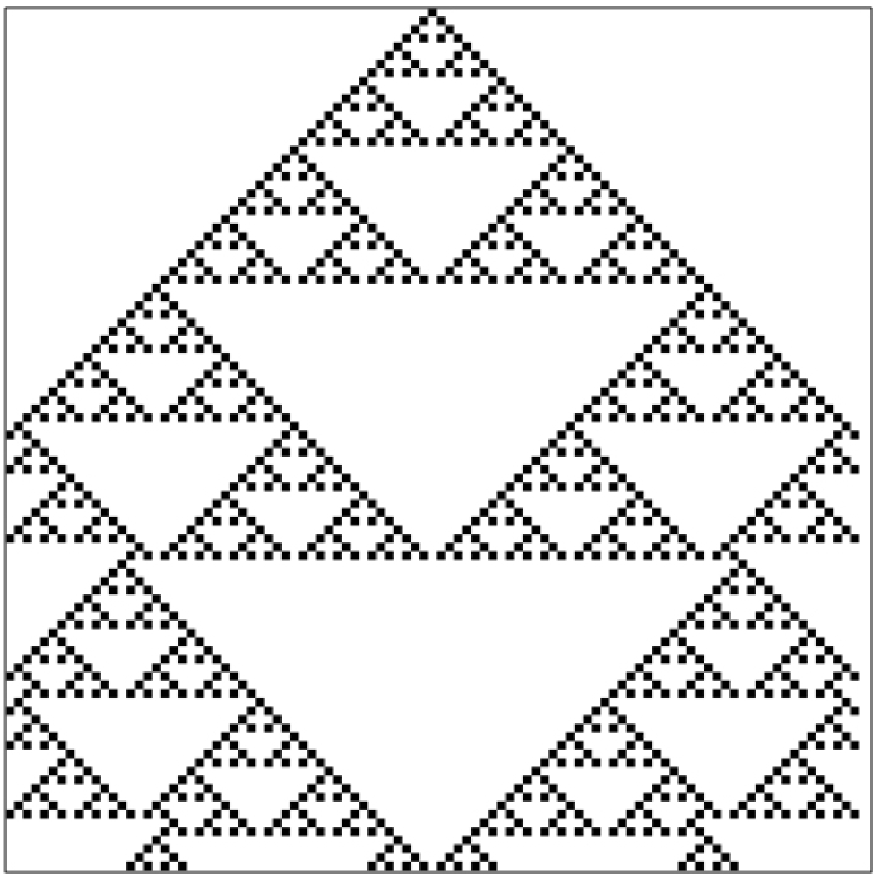


*Figure 7: Cellular Automata. Consecutive rows represent the evolution in time through subsequent classes. (Francesco Berto and Jacopo Tagliabue, 2017)*

The neighbourhood determines which surrounding cells impact a specific cell's future state. In one-dimensional cellular automata, the neighbourhood is defined by its size, meaning the number of cells it extends to the left or right. In two-dimensional automata, the two most common neighbourhood types are Moore and von Neumann neighbourhoods. Both types can have a size of any whole number that is greater than or equal to one. A Moore neighbourhood is square-shaped, for example a Moore neighbourhood of size 1 consists of the eight cells immediately surrounding a specific cell, including the cells on the diagonal. A von Neumann neighbourhood is shaped like a cross centred on  a cell, and a size 1 von Neumann neighbourhood includes the four cells directly above, below, to the left, and to the right the cell. The total number of possible neighbourhood configurations is equal to the number of states a cell can have, raised to the power of the number of cells in the neighbourhood, the number of possibilities can become huge very fast.

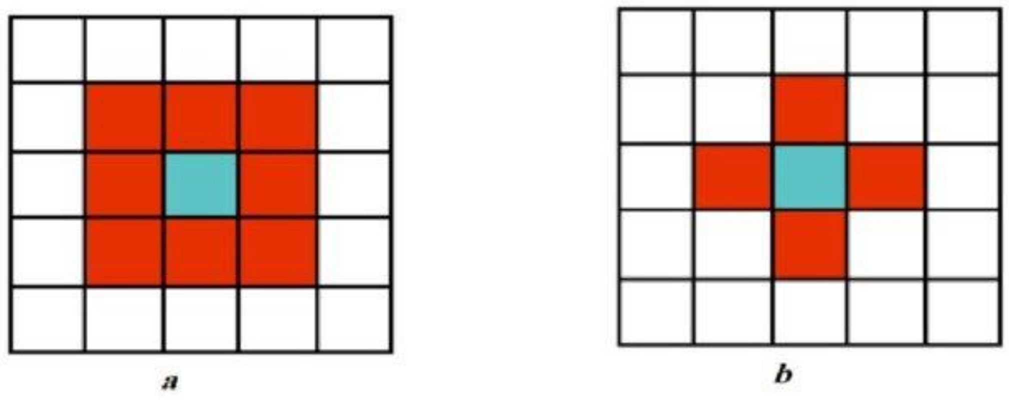


*Figure 8:a) Moore neighborhood (b) Von Neumann neighborhood (Gazmeh Khaje Nasir et al., 2014)*

Due to the versatility and flexibility of cellular automata, they have been effectively utilized in PCG for various applications, such as creating realistic terrain, natural patterns, and dungeon-like structures. They can generate a wide variety of content with unique characteristics and offer a powerful and adaptable approach as they can model complex and diverse phenomena while maintaining simplicity in their underlying structure. (Goldberg and Holland, 1988; Shaker, Togelius and Nelson, 2016)

Grammar based and L-systems

In computer science, grammars serve as fundamental structures with numerous applications in procedural content generation, particularly in generating vegetation for games.

One effective and straightforward way to create trees or bushes is to utilize a specific type of formal grammar called an L-system and interpret the outcomes as drawing instructions. L-systems are well-suited for reproducing the self-similarity observed in nature for example with fern branches, where the branch shape is repeated in each sub-branch.

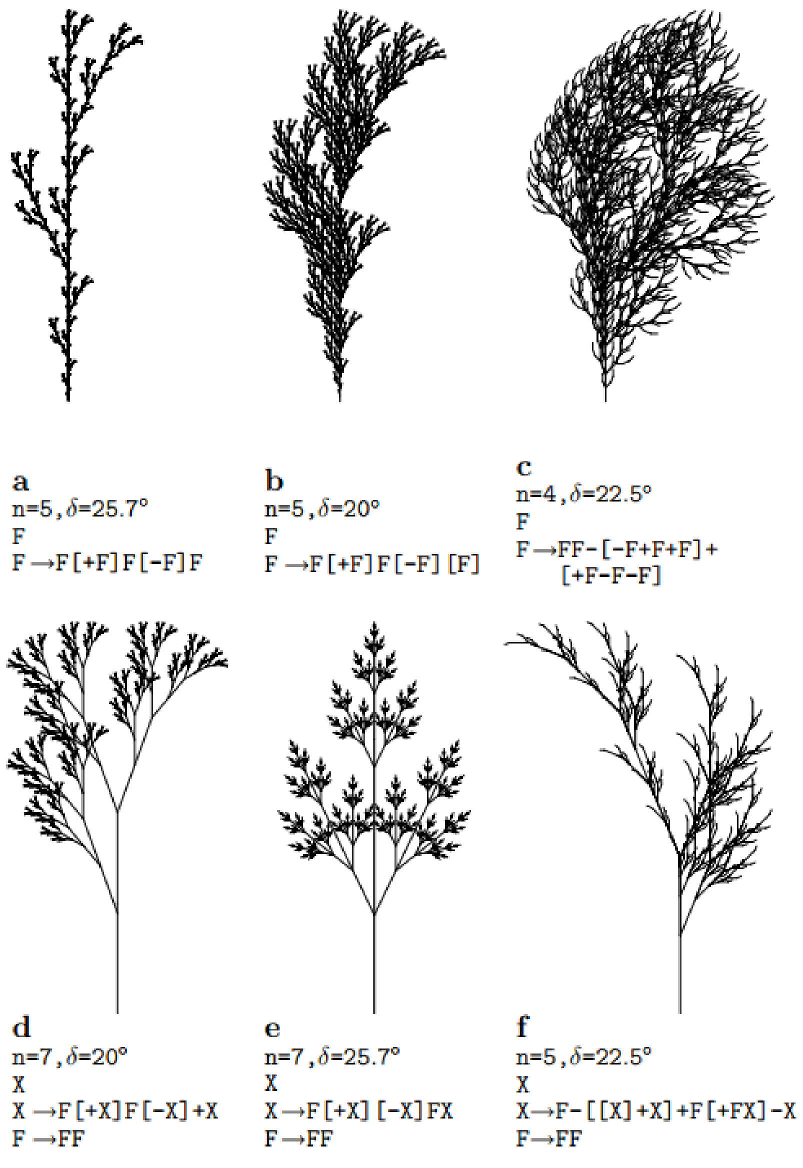


*Figure 9: L-system 2D trees (Aidan McInerny, 2015)*

Formal grammars consist of production rules for rewriting strings, with each rule transforming one symbol or sequence of symbols into another. Grammars were first introduced by linguist Noam Chomsky in the 1950s to model natural language and have since been widely applied in computer science. They can be deterministic, with one rule for each symbol or sequence, or nondeterministic, with several possible rules and random selection.

L-systems, a class of grammars defined by parallel rewriting, were introduced by biologist Aristid Lindenmayer in 1968 to model the growth of organic systems such as plants and algae. By interpreting the generated strings as instructions for turtle graphics, L-systems can create 2D and 3D artifacts. Turtle graphics is a simple drawing method often used in computer graphics; it involves an imaginary "turtle" that moves around on a 2D plane while carrying a pen. The turtle can follow commands to move forward or backward, and to turn left or right. As the turtle moves, it draws lines, creating shapes and patterns. However, many interesting shapes cannot be drawn this way, to generate complex branching structures, bracketed L-systems were developed which feature "push" and "pop" commands, which essentially pick the turtle up and move it somewhere else, breaking the continuous line being drawn.

Procedural content generation using L-systems can be applied to various game content types, plants are not the only thing for which formal grammars are useful, they can also be applied in areas like level and dungeon design, terrain generation, and quest/mission generation. Grammars can also be created by using search-based methods such as evolution algorithms, this can then enable an automatic grammar design. Similarly, L-systems can also be combined evolution algorithms, and by varying the fitness function it is possible to control the variation of content generated. (Prusinkiewicz, no date; Harold Abelson and Andrea Disessa, 1981; Prusinkiewicz and Lindenmayer, 1990; Shaker, Togelius and Nelson, 2016)

*Flood fill algorithm*

The flood fill algorithm, also known as seed fill, is a technique used to determine and modify connected areas in a multi-dimensional array (such as a grid or image) that share a common attribute, like colour. It is commonly used in paint programs for the "bucket" fill tool to change the colour of connected, similarly coloured areas, and in games like Go and Minesweeper to identify which pieces are cleared.

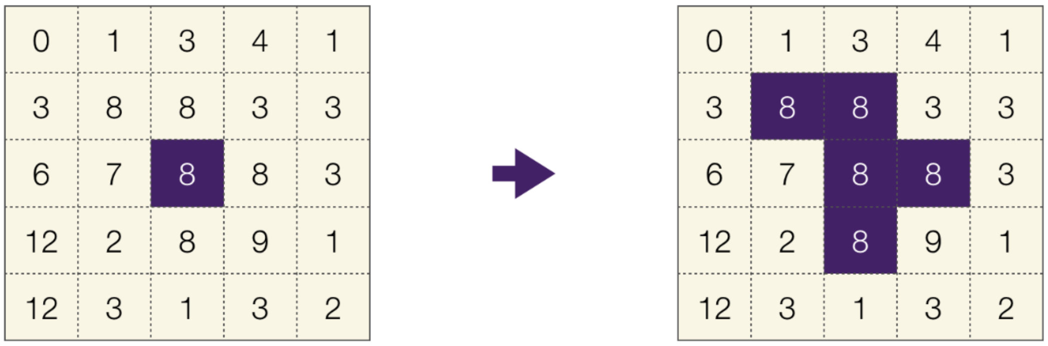


*Figure 10: Flood Fill algorithm connecting pixels (AlgoMonster, no date)*

The algorithm typically takes three input parameters: a starting node (or pixel), a target colour, and a replacement colour. The goal is to replace the target colour with the replacement colour in all connected nodes. The algorithm then starts at the beginning node and checks if its colour matches the target colour, if it doesn't match the algorithm stops since there's no need for modification. If the starting node's colour matches the target colour, it changes it to the replacement colour. It then recursively repeats the process for all neighbouring nodes in either four-way or eight-way connectivity. If it's in Four-way connectivity then the top, bottom, left, and right neighbours of the current node are checked. Alternatively, if its Eight-way connectivity it checks all eight neighbours, including the diagonal ones, of the current node. The algorithm continues to process connected nodes until no more nodes with the target colour are found. The flood fill algorithm can be implemented using various techniques, such as depth-first search, breadth-first search, or even scanline fill methods. The choice of implementation depends on factors like performance, stack/queue size, and the specific use case. (Thomas H. Cormen *et al.*, no date; Chudasama *et al.*, 2015)

## Research Conclusion

In conclusion, procedural content generation encompasses a variety of methods and techniques aimed at creating game content algorithmically, offering flexibility, adaptability, and often time and resource efficiency. Search-based methods, utilizing evolutionary algorithms or other stochastic search techniques, rely on iterating and refining potential solutions based on fitness values. These methods benefit from carefully chosen content representation and evaluation functions, which determine the quality and variety of generated content. Constructive generation methods, on the other hand, build content directly using predefined rules, algorithms, or procedures, making them particularly useful for rapid content creation and on-the-fly generation during gameplay. Examples of constructive-based methods include cellular automata, grammars (L-systems), and flood fill algorithms, each offering unique benefits for generating specific types of content. Balancing the choice of methods and techniques, as well as optimizing content representation and evaluation functions, is crucial for generating high-quality game content that meets desired objectives and enhances player experience.

## Implementation

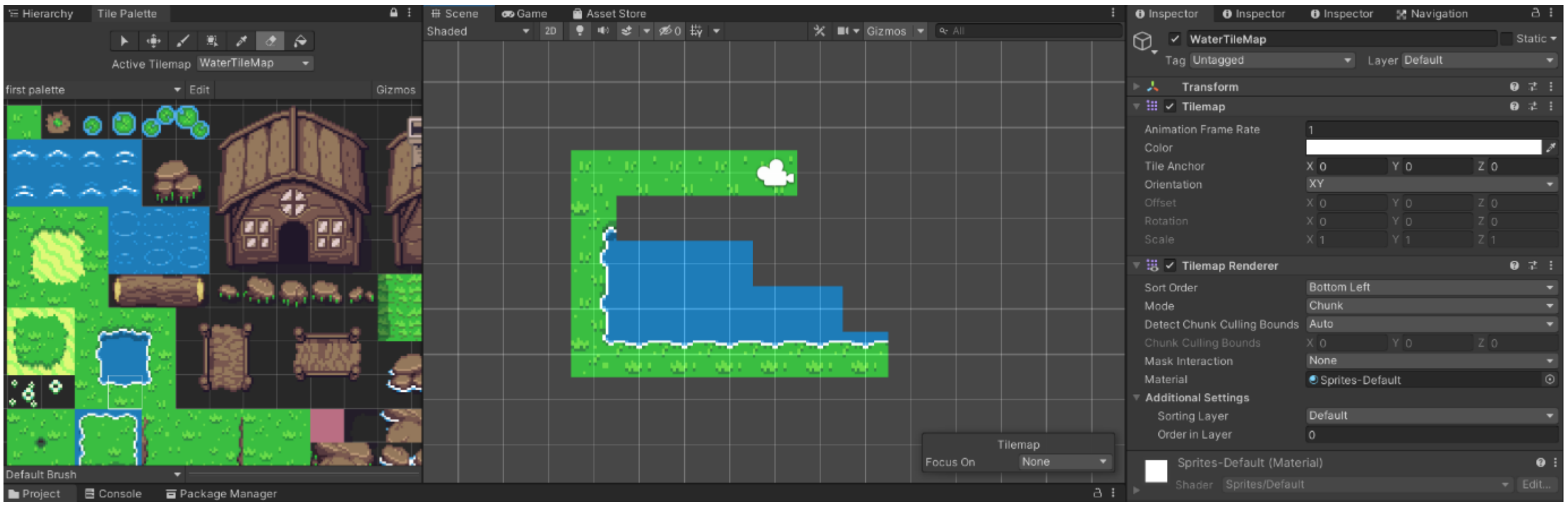
I originally chose the to do my project on PCG as I thought it would be an interesting topic and I had to make a game or tool that involved PCG. Early on, I had intentions of making some sort of PCG game like an endless runner. However, I decided instead to develop a 2D map generator as I felt the nature of the project would allow me to easily showcase the main areas involved in PCG creation. When it came to creating the 2D map generator I had to keep several things in mind:

- I wanted to demonstrate PCG in action, which involves the user visibly being able to see new generations.
- I also wanted the user to have the ability to influence the generated map and then see the effect of their changes.
- I wanted the maps to be saveable and loadable.
- I wanted it to be easy for a user to add their own content.

Firstly, I needed to decide on what game engine I would make my 2D map generator in. I decided on Unity as I had a small amount of previous experience working with Unity where I had created a 2D game, so I knew certain tools that were available in unity for 2D creation but there was still a large learning outcome to be achieved.

When beginning, I figured that using the 2D tile map was my best approach as it allowed me to stick to a simple grid, starting with the tile map, it also allowed me to pay around with various types of tiles and palettes in unity, this helped me to get an idea of how the map would be constructed.

I knew I needed to start with some simple terrain elements and decided to focus initially on establishing water and land as the two primary components of the map generation. I created a tile map and found an appropriate sprite palette (ArMM1998, 2017) for what I had in mind.



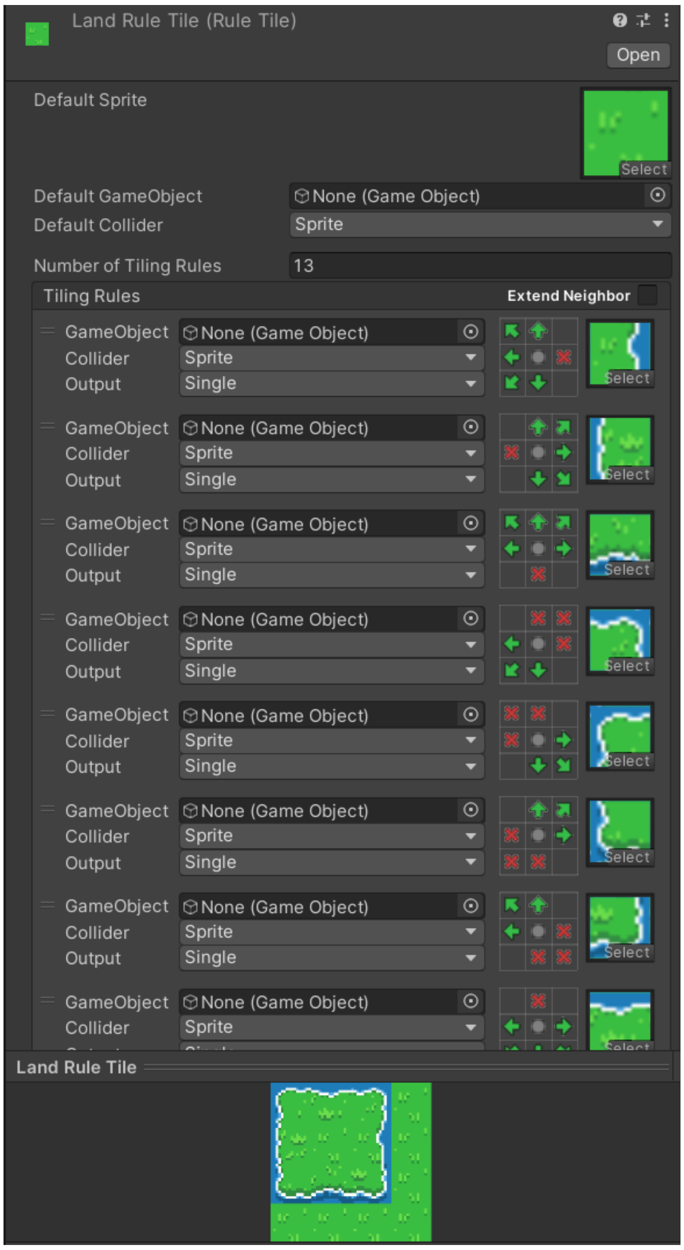
WaterTileMap
Static
Tag Untagged | Layer Default
Transform
Tilemap
Animation Frame Rate 1
Color
Tile Anchor X 0 Y 0 Z 0
Orientation XY
Offset X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Tilemap Renderer
Sort Order Bottom Left
Mode Chunk
Detect Chunk Culling Bounds Auto
Chunk Culling Bounds X 0 Y 0 Z 0
Mask Interaction None
Material Sprites-Default
Additional Settings
Sorting Layer Default
Order in Layer 0
Sprites-Default (Material)
Shader Sprites/Default
Edit...

*Figure 11: Tile map and Tile pallette (from my project)*

After placing tiles manually down on the tile map, I soon realised that I would need some way of specifying the orientation of my sprites automatically. When looking around for resources on different types of 2D tiles that might fix the issue. I encountered the unity 2D extra pack (Unity technologies, 2017). I had to go into unity packages files on my computer and edit the JSON file to add it. One problem I encountered was that the GitHub I had found was an older version. So after so figuring out I managed to correct the version type in the JSON file itself.

After setting up the 2D extras pack, I found two different types of tiles included that were perfect for what I wanted.

For solving the issue of the orientation of my tiles, I used the rule tile. The rule tile allows me to specify what tiles will appear, depending on if tiles of the same tile are nearby in a specific arrangement or if there is a different tile nearby. The more rules I specified the nicer the land tiles look when drawing.



Land Rule Tile (Rule Tile)

| | |
|---|---|
| Default Sprite | |
| Default GameObject | None (Game Object) |
| Default Collider | Sprite |
| Number of Tiling Rules | 13 |

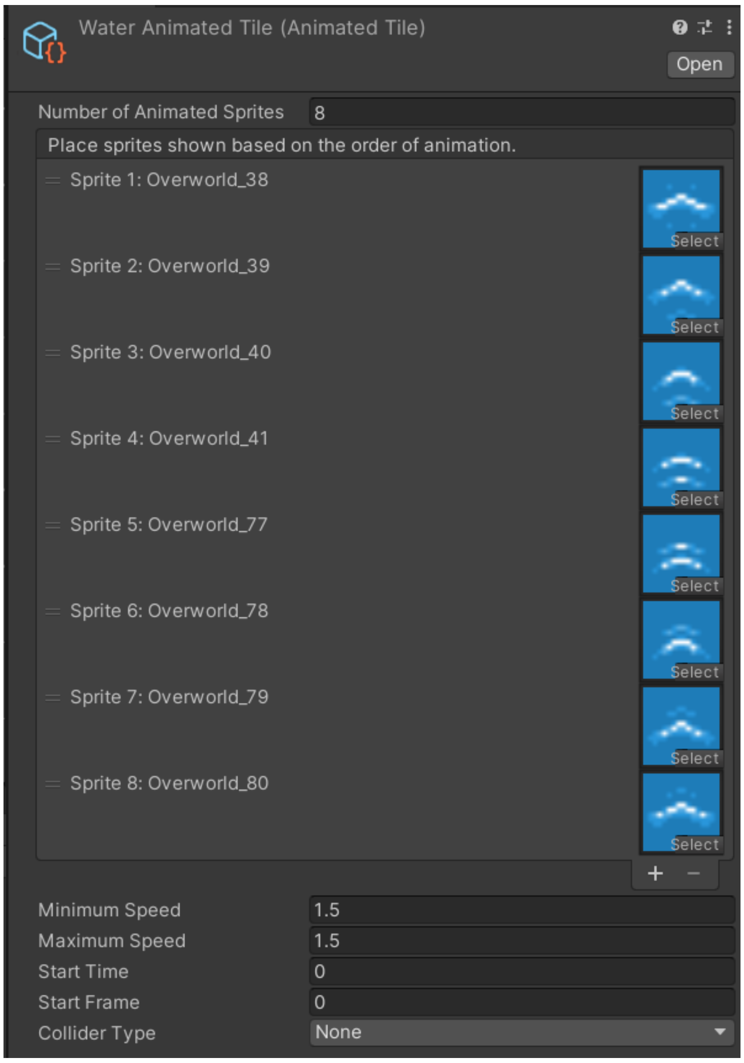
Tiling Rules — Extend Neighbor

| GameObject | Collider | Output |
|---|---|---|
| None (Game Object) | Sprite | Single |
| None (Game Object) | Sprite | Single |
| None (Game Object) | Sprite | Single |
| None (Game Object) | Sprite | Single |
| None (Game Object) | Sprite | Single |
| None (Game Object) | Sprite | Single |
| None (Game Object) | Sprite | Single |
| None (Game Object) | Sprite | |

Land Rule Tile

Figure 12: Rule tiles for land tiles (from my project)

The other type of tile that was useful from the 2D extras pack, was the animated tile. The animated tile allowed me to arrange the water tiles from my sprite pack (ArMM1998, 2017) to create a wave like motion when the game was in play.



Water Animated Tile (Animated Tile)

Number of Animated Sprites: 8

Place sprites shown based on the order of animation.

- Sprite 1: Overworld_38
- Sprite 2: Overworld_39
- Sprite 3: Overworld_40
- Sprite 4: Overworld_41
- Sprite 5: Overworld_77
- Sprite 6: Overworld_78
- Sprite 7: Overworld_79
- Sprite 8: Overworld_80

| | |
|---|---|
| Minimum Speed | 1.5 |
| Maximum Speed | 1.5 |
| Start Time | 0 |
| Start Frame | 0 |
| Collider Type | None |

Figure 13: Animation tiles for water tiles (from my project)

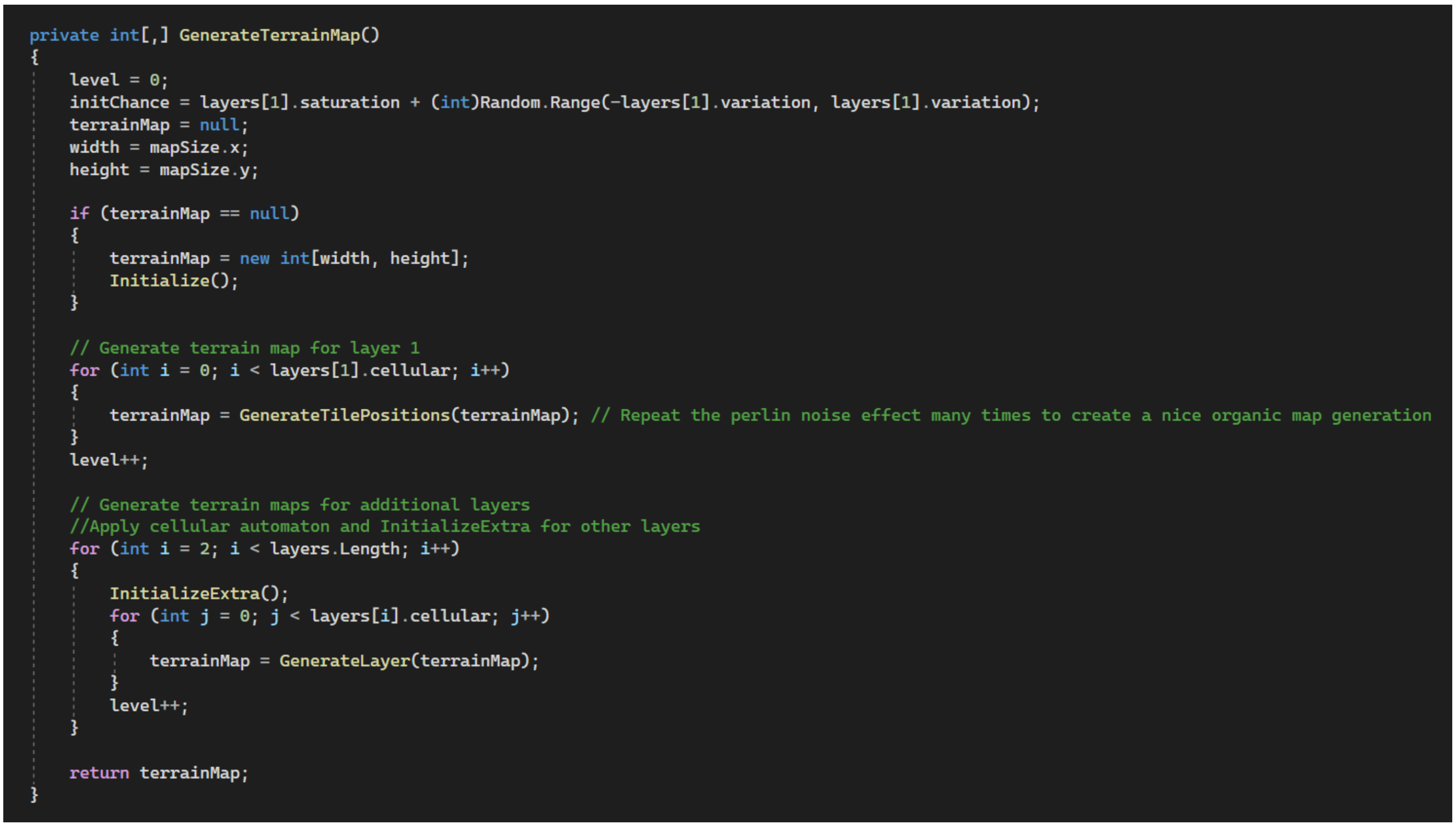
Now that the issue of tiles had been taken care of I had to figure out how to effectively generate the tiles in some sort of arrangement. Firstly, I started by creating separate tile maps for the Land and Water layers. I thought from reading about 2D PCG that my best approach would be to stack layers on top of each other and from there define the rules for spawning.

I then needed to figure out what PCG techniques would be applicable for the type of generation I wanted to pursue for a 2D map generator; from my research I identified that the following concepts would be useful to note:

Constructive Generation methods seemed like the right choice as they provided fast generation based on predefined rules, this would allow me to create a system where users could generate new maps quickly, while also being able to ensure quality content through the use of specific algorithms.

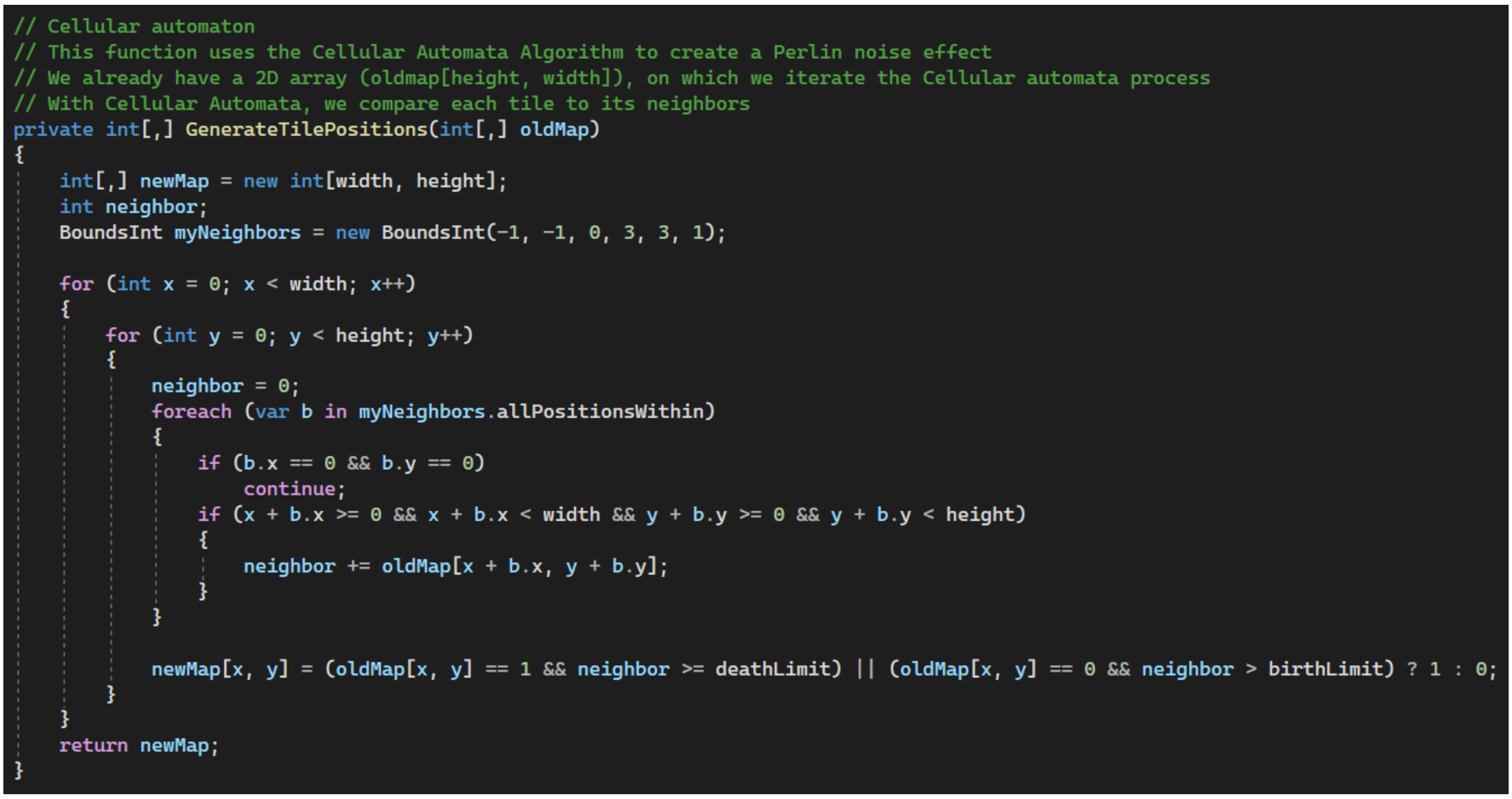
Perlin noise and Cellular automata seemed an appropriate approach for developing the organic nature of the land in the map. And Flood Fill seemed and appropriate algorithm choice for achieving more control of the land generation

Mixed authorship was a concept discussed earlier and was precisely what I had in mind for how the user can interact with the generation, showcasing the changes they make with each map.
The main functionality of the map generation was created in the following two functions:



```
private int[,] GenerateTerrainMap()
{
    level = 0;
    initChance = layers[1].saturation + (int)Random.Range(-layers[1].variation, layers[1].variation);
    terrainMap = null;
    width = mapSize.x;
    height = mapSize.y;

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

    // Generate terrain map for layer 1
    for (int i = 0; i < layers[1].cellular; i++)
    {
        terrainMap = GenerateTilePositions(terrainMap); // Repeat the perlin noise effect many times to create a nice organic map generation
    }
    level++;

    // Generate terrain maps for additional layers
    //Apply cellular automaton and InitializeExtra for other layers
    for (int i = 2; i < layers.Length; i++)
    {
        InitializeExtra();
        for (int j = 0; j < layers[i].cellular; j++)
        {
            terrainMap = GenerateLayer(terrainMap);
        }
        level++;
    }

    return terrainMap;
}
```

*Figure 14: GenerateTerrainMap() function (from my project)*



```
// Cellular automaton
// This function uses the Cellular Automata Algorithm to create a Perlin noise effect
// We already have a 2D array (oldmap[height, width]), on which we iterate the Cellular automata process
// With Cellular Automata, we compare each tile to its neighbors
private int[,] GenerateTilePositions(int[,] oldMap)
{
    int[,] newMap = new int[width, height];
    int neighbor;
    BoundsInt myNeighbors = new BoundsInt(-1, -1, 0, 3, 3, 1);

    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            neighbor = 0;
            foreach (var b in myNeighbors.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)
                {
                    neighbor += oldMap[x + b.x, y + b.y];
                }
            }

            newMap[x, y] = (oldMap[x, y] == 1 && neighbor >= deathLimit) || (oldMap[x, y] == 0 && neighbor > birthLimit) ? 1 : 0;
        }
    }
    return newMap;
}
```

*Figure 15: GenerateTilePositions() function (from my project)*

These two functions work together to generate a terrain map using cellular automata and Perlin noise. The first function, **GenerateTerrainMap**, is responsible for generating the terrain map for each layer, while the second function, **GenerateTilePositions**, uses cellular automata to modify the terrain map based on neighbouring tiles.

Here is a run down of the specifics of the code:

**GenerateTerrainMap**:

1. Initializes variables for level, initial chance, terrain map, width, and height.
2. If the terrain map is null, it creates a new 2D integer array for the terrain map and calls the **Initialize** function.
3. Generates terrain map for layer 1 by calling the **GenerateTilePositions** function multiple times, depending on the value of **layers[1].cellular**. This is done to create a nice organic map generation.
4. Increment the **level** variable.
5. Generates terrain maps for additional layers by applying cellular automata and the **InitializeExtra** function for other layers.
6. After all layers have been generated, the **terrainMap** is returned by the function.

**GenerateTilePositions**:

1. This function takes a 2D integer array **oldMap** as input, which is what will be modified using cellular automata.
2. Creates a new 2D integer array **newMap** with the same dimensions as **oldMap** to store the modified terrain map.
3. Defines **myNeighbors** as a **BoundsInt** object representing the 8 neighboring tiles around a given tile (excluding the tile itself).
4. Iterates through all the tiles in **oldMap** using nested loops for the **x** and **y** coordinates.

5. For each tile in **oldMap**, it counts the number of neighboring tiles that have a value of 1, storing the count in the **neighbor** variable.
6. Updates the corresponding tile in **newMap** based on the cellular automata rules:
    - If the current tile in **oldMap** has a value of 1 and the **neighbor** count is greater than or equal to **deathLimit**, the new tile in **newMap** is set to 1.
    - If the current tile in **oldMap** has a value of 0 and the **neighbor** count is greater than **birthLimit**, the new tile in **newMap** is set to 1.
    - In all other cases, the new tile in **newMap** is set to 0.
7. Returns the modified terrain map **newMap**.

The **GenerateTerrainMap** function generates the terrain map for each layer, while the **GenerateTilePositions** function modifies the terrain map using cellular automata based on neighboring tiles. This combination of Perlin noise and cellular automata results in a more organic and visually pleasing terrain generation.

Here is an image of what the map generation looked like in the early stages of development, as you can see there is no fancy sprites, but I had managed to achieve some basic map generation through the use of the two functions **GenerateTerrainMap** and **GenerateTilePositions**.

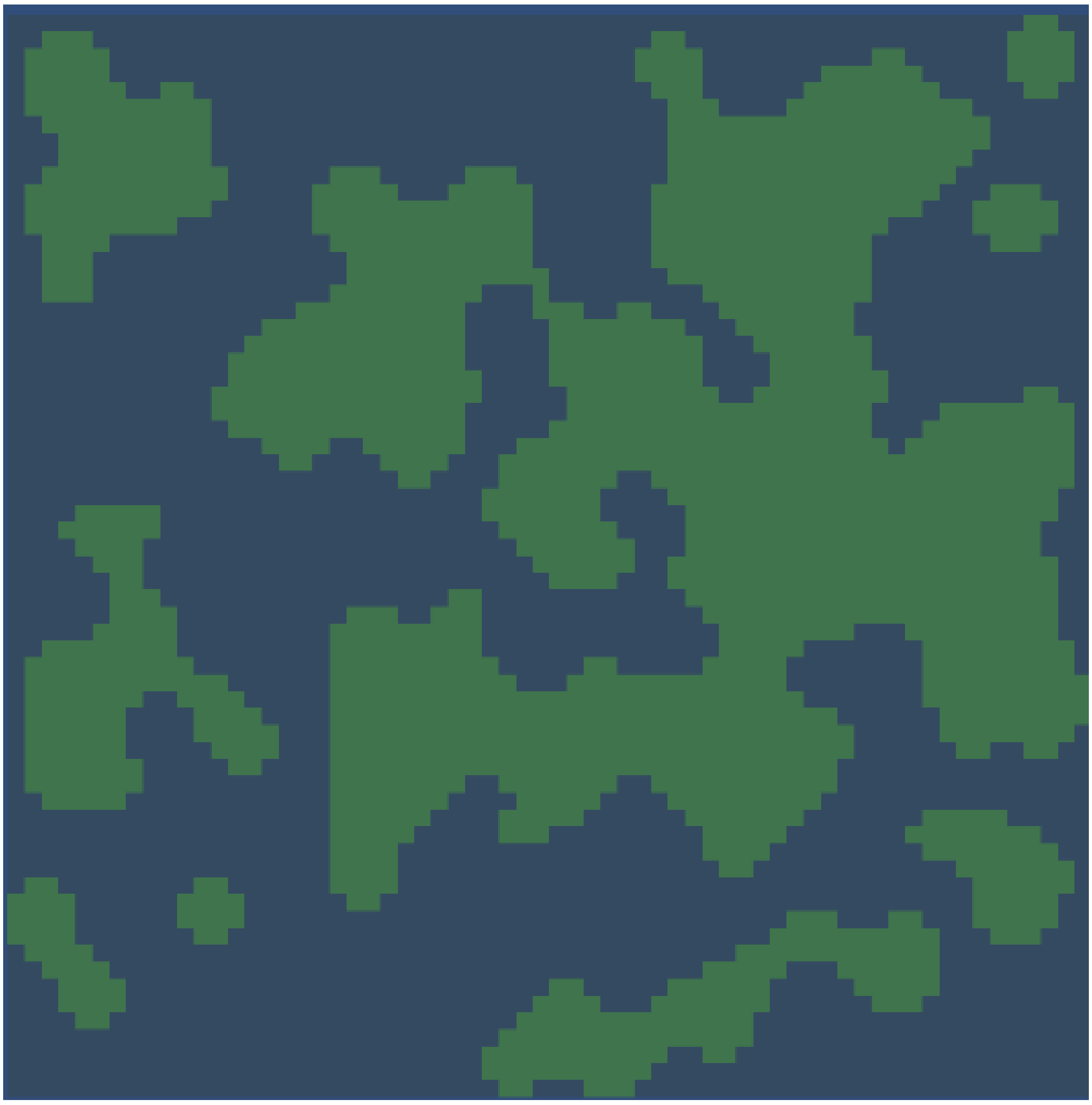
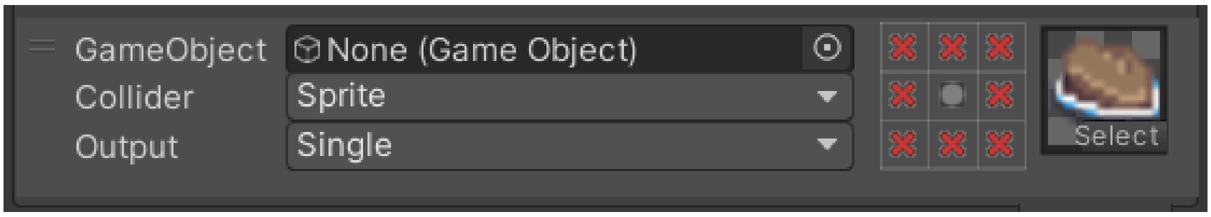


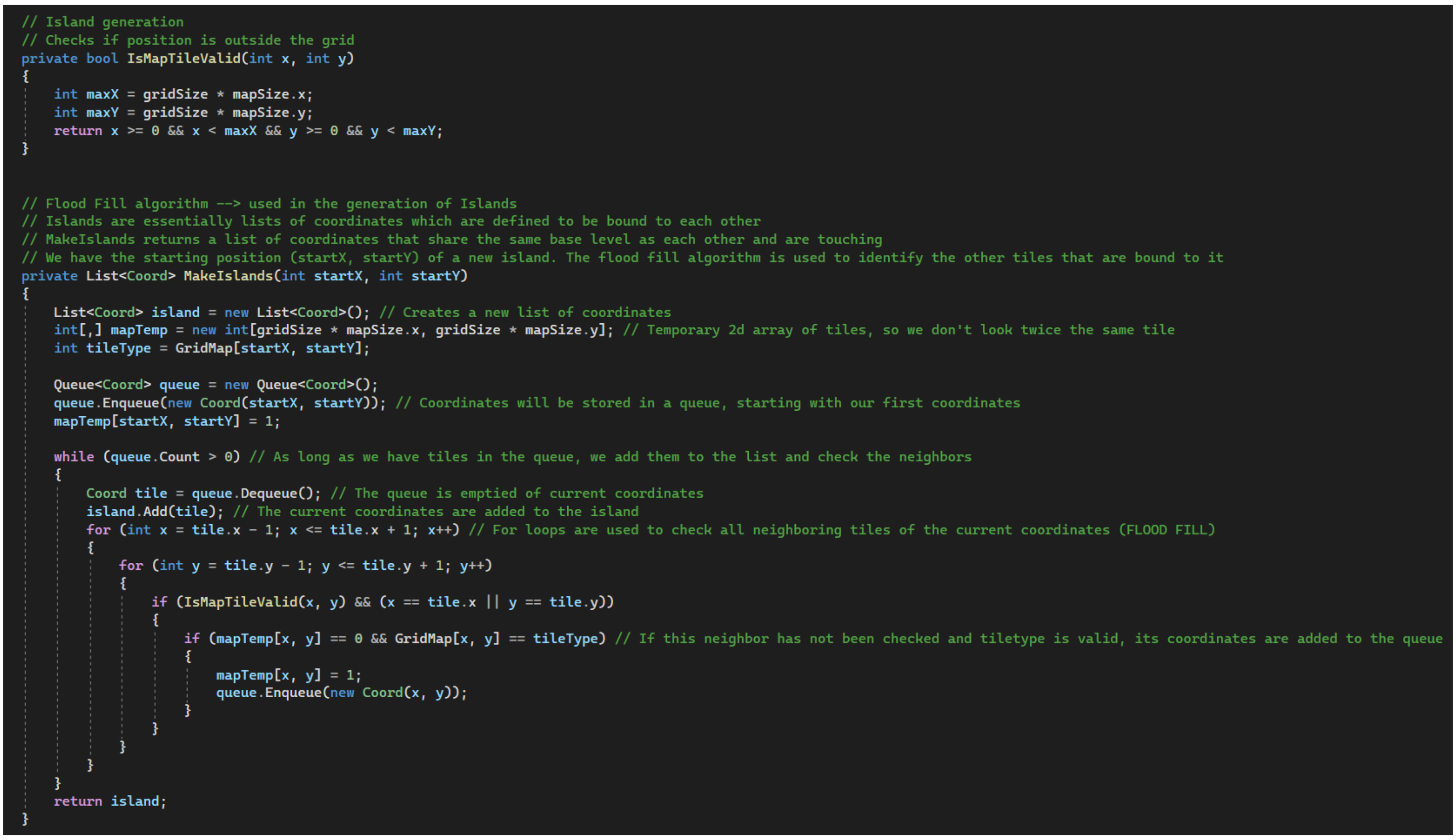
*Figure 16: Simple genereated map (from my project)*

However, I found there was a lack of control in the generation, specifically with blocks on one spawning for land, which didn't look so nice. At first, my solution was to use rule tiles. I went and specified many more rules for the tiles and tried to specify if land tiles were on their own, i.e. no land tiles were adjacent to the tile. Then it would be turned into a water rock sprite.



GameObject: None (Game Object)
Collider: Sprite
Output: Single

However, this fix was just a band aid and didn't fix the underlying issue. I also then began specifying rules for single land tiles only next to an island as the sticking out tile looked messy. This quickly became a messy solution with many rules for changing the tiles into rocks. I then decided to scrap this method and began approaching the problem with a new mindset, I looked over the different PCG techniques and realised I could use the flood fill algorithm to specify the minimum sizes for an island. Below is the function that I used to create the islands.



```
// Island generation
// Checks if position is outside the grid
private bool IsMapTileValid(int x, int y)
{
    int maxX = gridSize * mapSize.x;
    int maxY = gridSize * mapSize.y;
    return x >= 0 && x < maxX && y >= 0 && y < maxY;
}


// Flood Fill algorithm --> used in the generation of Islands
// Islands are essentially lists of coordinates which are defined to be bound to each other
// MakeIslands returns a list of coordinates that share the same base level as each other and are touching
// We have the starting position (startX, startY) of a new island. The flood fill algorithm is used to identify the other tiles that are bound to it
private List<Coord> MakeIslands(int startX, int startY)
{
    List<Coord> island = new List<Coord>(); // Creates a new list of coordinates
    int[,] mapTemp = new int[gridSize * mapSize.x, gridSize * mapSize.y]; // Temporary 2d array of tiles, so we don't look twice the same tile
    int tileType = GridMap[startX, startY];

    Queue<Coord> queue = new Queue<Coord>();
    queue.Enqueue(new Coord(startX, startY)); // Coordinates will be stored in a queue, starting with our first coordinates
    mapTemp[startX, startY] = 1;

    while (queue.Count > 0) // As long as we have tiles in the queue, we add them to the list and check the neighbors
    {
        Coord tile = queue.Dequeue(); // The queue is emptied of current coordinates
        island.Add(tile); // The current coordinates are added to the island
        for (int x = tile.x - 1; x <= tile.x + 1; x++) // For loops are used to check all neighboring tiles of the current coordinates (FLOOD FILL)
        {
            for (int y = tile.y - 1; y <= tile.y + 1; y++)
            {
                if (IsMapTileValid(x, y) && (x == tile.x || y == tile.y))
                {
                    if (mapTemp[x, y] == 0 && GridMap[x, y] == tileType) // If this neighbor has not been checked and tiletype is valid, its coordinates are added to the queue
                    {
                        mapTemp[x, y] = 1;
                        queue.Enqueue(new Coord(x, y));
                    }
                }
            }
        }
    }
    return island;
}
```

*Figure 17: IsMapTileValid() and MakeIslands() functions (from my project)*

These two functions work together to generate islands in the map. The first function, **IsMapTileValid**, checks whether a given tile coordinate is within the bounds of the map. The second function, **MakeIslands**, uses the flood fill algorithm to identify and group together connected tiles with the same base level as an island.

**IsMapTileValid**:

1. Takes two integer parameters **x** and **y**, representing the x and y coordinates of a tile on the map.
2. Calculates the maximum x-coordinate **maxX** and y-coordinate **maxY** based on the product of **gridSize** and **mapSize**.
3. Returns **true** if the provided **x** and **y** are within the valid range of the map, i.e., between 0 (inclusive) and **maxX** or **maxY** (exclusive), respectively. Otherwise, it returns **false**.

**MakeIslands**:

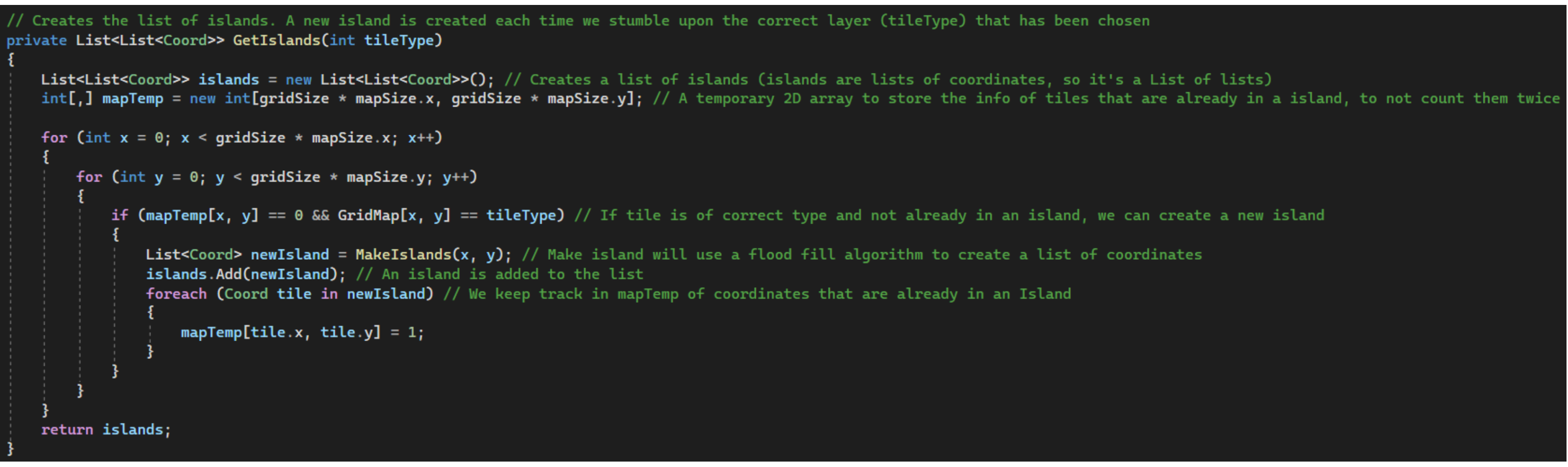
1. Takes two integer parameters **startX** and **startY**, they represent the coordinates of a new island.
2. Initializes an empty list of coordinates **island** to store the tiles that make up the island.
3. Creates a temporary 2D integer array **mapTemp** to keep track of visited tiles, ensuring the same tile isn't processed more than once.
4. Retrieves the **tileType** of the starting tile from the **GridMap** 2D array.
5. Initializes a **Queue** of **Coord** objects and enqueues the starting coordinates. The queue will be used to process neighbouring tiles.
6. Sets the corresponding starting position in **mapTemp** to 1, indicating that it has been visited.
7. Iterates through the tiles in the queue until the queue is empty. For each tile in the queue:
    - Dequeue the current tile and add it to the **island** list.
    - Check the neighbouring tiles of the current tile using nested for loops.

  - If a neighbouring tile is valid (using **IsMapTileValid** function) and has the same **tileType** as the current tile, and hasn't been visited (indicated by a 0 in **mapTemp**), mark it as visited in **mapTemp** and enqueue its coordinates in the queue.

8. Returns the list of coordinates **island**, representing the connected tiles that make up the island.

The **MakeIslands** function uses the flood fill algorithm to group connected tiles with the same base level, while the **IsMapTileValid** function is used to ensure only valid tiles are processed. Together, these functions generate islands and their minimum tiles in the map but can also be used to specify minimum tiles for bodies of water.

Additionally, to these two functions that are responsible for island creating, I wanted to be able to account for each individual island, to do this I created the **GetIslands** function which also makes use of the flood fill algorithm for creating a list of coordinates.



```
// Creates the list of islands. A new island is created each time we stumble upon the correct layer (tileType) that has been chosen
private List<List<Coord>> GetIslands(int tileType)
{
    List<List<Coord>> islands = new List<List<Coord>>(); // Creates a list of islands (islands are lists of coordinates, so it's a List of lists)
    int[,] mapTemp = new int[gridSize * mapSize.x, gridSize * mapSize.y]; // A temporary 2D array to store the info of tiles that are already in a island, to not count them twice

    for (int x = 0; x < gridSize * mapSize.x; x++)
    {
        for (int y = 0; y < gridSize * mapSize.y; y++)
        {
            if (mapTemp[x, y] == 0 && GridMap[x, y] == tileType) // If tile is of correct type and not already in an island, we can create a new island
            {
                List<Coord> newIsland = MakeIslands(x, y); // Make island will use a flood fill algorithm to create a list of coordinates
                islands.Add(newIsland); // An island is added to the list
                foreach (Coord tile in newIsland) // We keep track in mapTemp of coordinates that are already in an Island
                {
                    mapTemp[tile.x, tile.y] = 1;
                }
            }
        }
    }
    return islands;
}
```

*Figure 18: GetIslands() function (from my project)*

The **GetIslands** function takes an integer **tileType** as input and returns a list of islands, where each island is a list of coordinates (**Coord**). Here's an in-depth explanation of the function:

1. Initializes a new list of lists of **Coord** objects named **islands**. Each inner list represents an individual island.
2. Creates a temporary 2D integer array **mapTemp** with dimensions **gridSize * mapSize.x** and **gridSize * mapSize.y** to store information about which tiles have already been assigned to an island, to avoid counting them twice.

3. Iterates through all the tiles in the **GridMap** using nested loops for the **x** and **y** coordinates.
4. If the current tile in **mapTemp** is not yet assigned to an island (value 0) and its corresponding tile in **GridMap** has the specified **tileType**, a new island is created.
5. Calls the **MakeIslands** function with the current tile's **x** and **y** coordinates. **MakeIslands** uses a flood-fill algorithm to create a list of coordinates that belong to the same island, starting from the current tile.
6. Adds the **newIsland** list to the **islands** list.
7. Iterates through all the **Coord** objects in the **newIsland** list and marks them in the **mapTemp** array by setting their corresponding value to 1, indicating that they have been assigned to an island.
8. After processing all the tiles, returns the **islands** list, containing all the islands found in the **GridMap**.

The **GetIslands** function identifies groups of contiguous tiles with the same **tileType** and organizes them into islands. This allows the generation of more complex and interconnected map features.

With the issue of the islands fixed, I then added in the rule tiles for nicer looking land and animation tiles for the water animation into the map generator script. However, the map was looking very empty at this point.

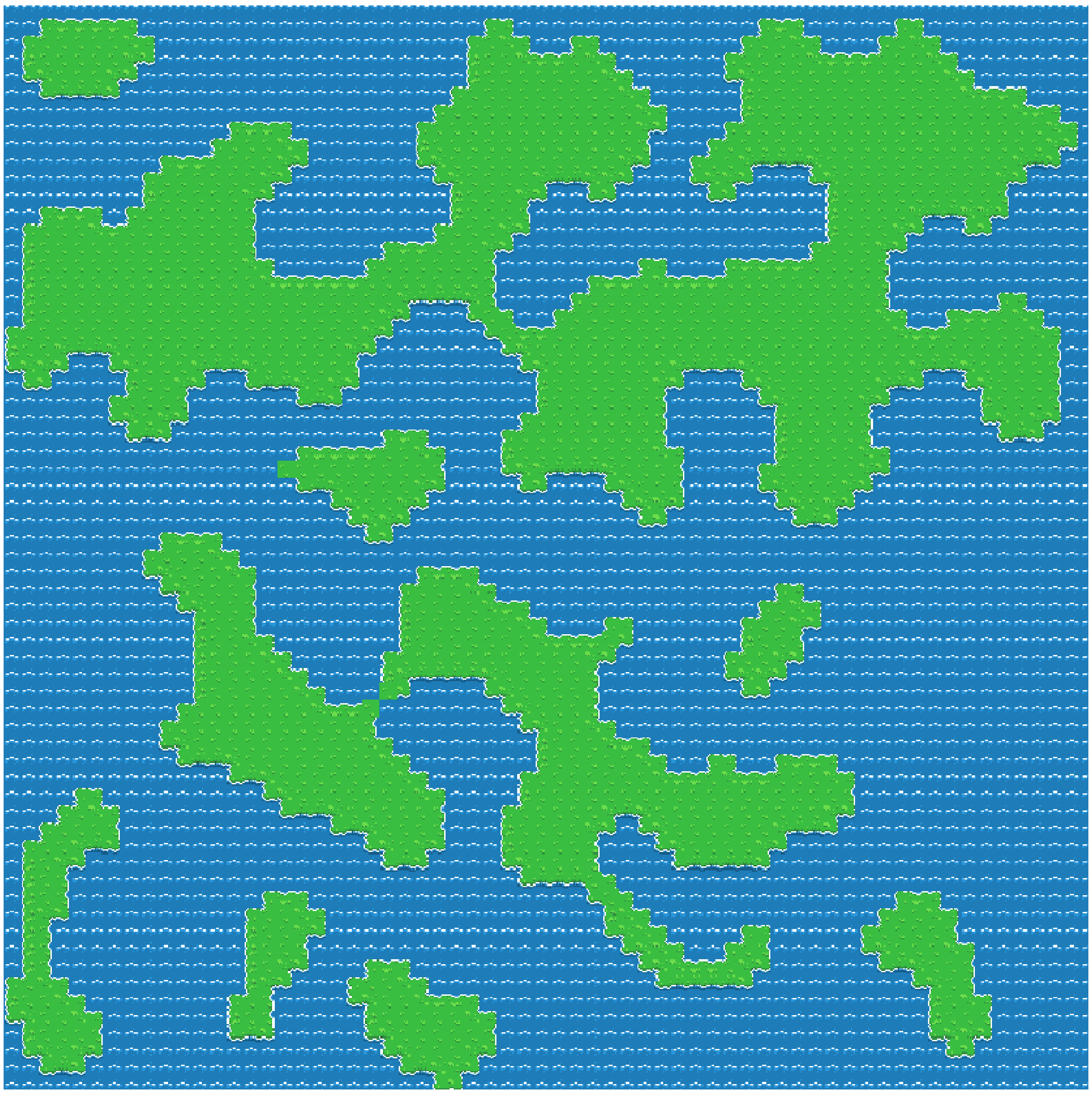


*Figure 19: Updated generated map (from my project)*

I then tried the approach of adding in another layer, just like I had done for the land and water. This worked and produced some interesting shapes and patterns between the rocks. I made it so that the rocks could only spawn on the sublayer (land) and was also able to specify the distance the rocks would be from the sub layers sub layer (water).

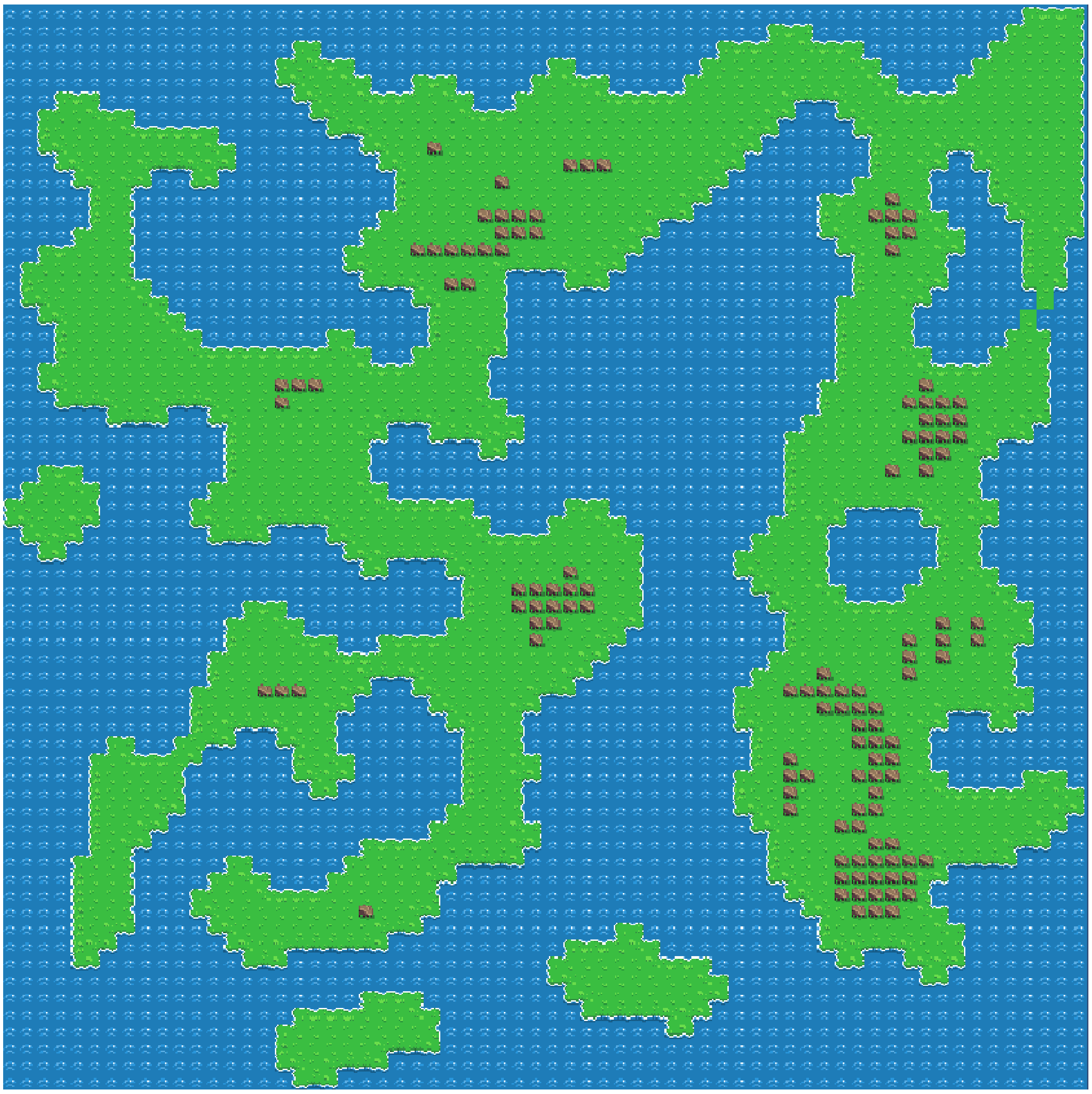
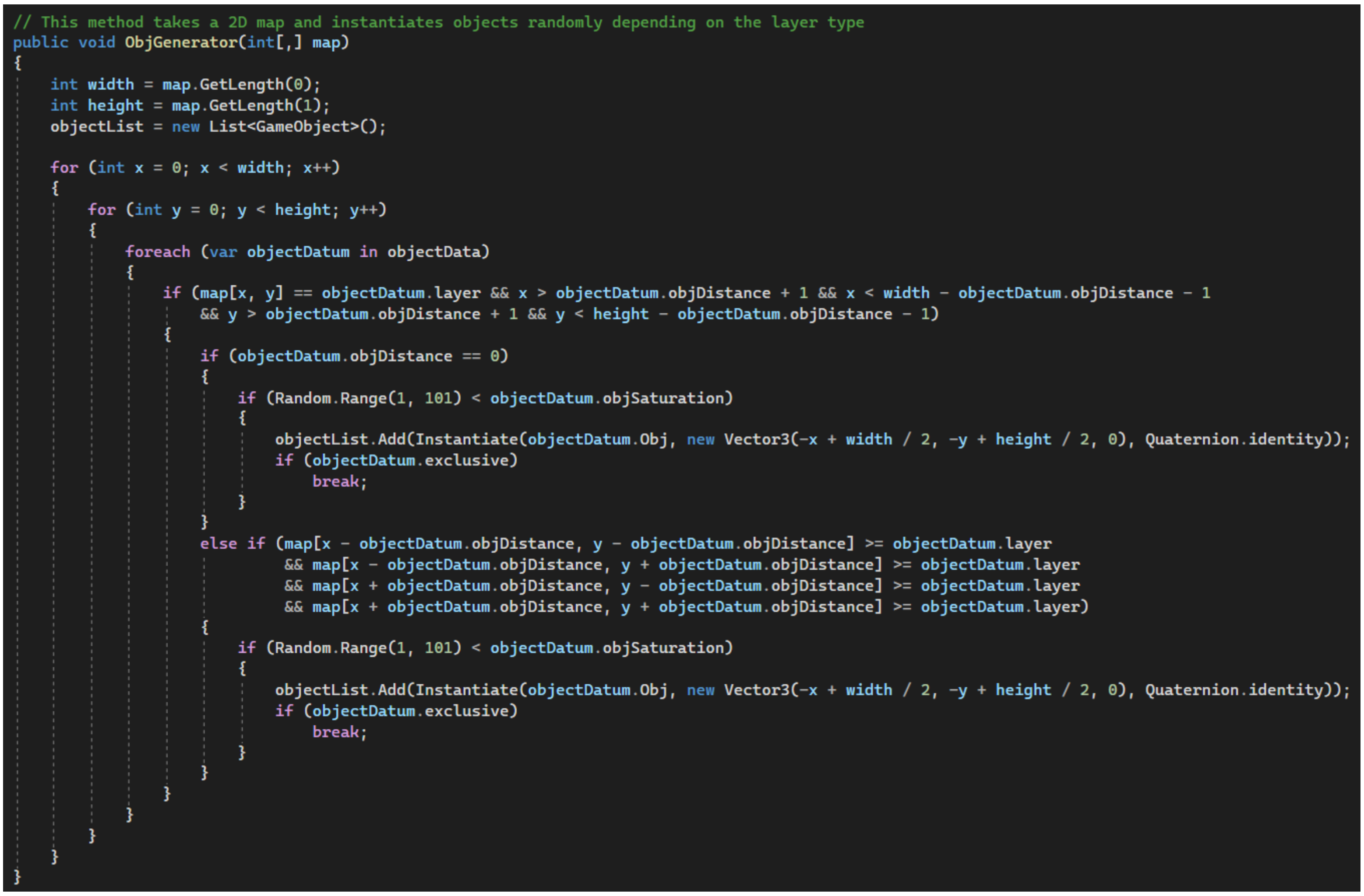


*Figure 20: Updated generated map with rock layer (from my project)*

This approached worked perfectly fine and is very suitable for types of terrain features such as mountains (which the rocks above can be imagined as). However, I also felt it would be appropriate to create a way for a user to add objects and scatter them randomly around the map, while also having some control such as saturation and distance from the water.

To do this I decided to create an additional script to handle these new game objects that would be implemented. The main function of the script **ObjGenerator** works by taking a 2D map and instantiating objects randomly throughout the map based on the layer type (e.g. water layer or land layer), object distance, and object saturation.



```
// This method takes a 2D map and instantiates objects randomly depending on the layer type
public void ObjGenerator(int[,] map)
{
    int width = map.GetLength(0);
    int height = map.GetLength(1);
    objectList = new List<GameObject>();

    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            foreach (var objectDatum in objectData)
            {
                if (map[x, y] == objectDatum.layer && x > objectDatum.objDistance + 1 && x < width - objectDatum.objDistance - 1
                    && y > objectDatum.objDistance + 1 && y < height - objectDatum.objDistance - 1)
                {
                    if (objectDatum.objDistance == 0)
                    {
                        if (Random.Range(1, 101) < objectDatum.objSaturation)
                        {
                            objectList.Add(Instantiate(objectDatum.Obj, new Vector3(-x + width / 2, -y + height / 2, 0), Quaternion.identity));
                            if (objectDatum.exclusive)
                                break;
                        }
                    }
                    else if (map[x - objectDatum.objDistance, y - objectDatum.objDistance] >= objectDatum.layer
                             && map[x - objectDatum.objDistance, y + objectDatum.objDistance] >= objectDatum.layer
                             && map[x + objectDatum.objDistance, y - objectDatum.objDistance] >= objectDatum.layer
                             && map[x + objectDatum.objDistance, y + objectDatum.objDistance] >= objectDatum.layer)
                    {
                        if (Random.Range(1, 101) < objectDatum.objSaturation)
                        {
                            objectList.Add(Instantiate(objectDatum.Obj, new Vector3(-x + width / 2, -y + height / 2, 0), Quaternion.identity));
                            if (objectDatum.exclusive)
                                break;
                        }
                    }
                }
            }
        }
    }
}
```

*Figure 21: ObjGenerator() function (from my project)*

The **ObjGenerator** function takes a 2D integer array **map** as input and instantiates game objects on the map based on layer types and various properties defined in **objectData**. Here's an in-depth explanation of the function:

1. Retrieves the width and height of the input map by calling **GetLength** with the appropriate dimension index.
2. Initializes an empty list of **GameObject** objects named **objectList** to store the instantiated game objects.
3. Iterates through all the tiles in the map using nested loops for the **x** and **y** coordinates.
4. For each tile, iterates through the **objectData** collection, which contains information about game objects that can be instantiated.

5. Checks if the current tile's layer matches the **objectDatum.layer**, and if the tile's position is within the valid range (not too close to the map edges) based on the **objectDatum.objDistance**.
6. If the object distance is 0, checks whether a random number between 1 and 100 (inclusive) is less than **objectDatum.objSaturation**. If true, instantiates the game object at the current tile position (adjusted to center the object) and adds it to **objectList**. If the object is exclusive, breaks out of the inner loop.
7. If the object distance is not 0, checks if all neighboring tiles at the specified object distance have a layer equal to or greater than **objectDatum.layer**. If true, and if a random number between 1 and 100 (inclusive) is less than **objectDatum.objSaturation**, instantiates the game object at the current tile position (adjusted to center the object) and adds it to **objectList**. If the object is exclusive, breaks out of the inner loop.

The **ObjGenerator** function is used to populate the map with various game objects based on the layer type, object distance, object saturation, and exclusivity. This allows for the creation of diverse and interesting environments in the game world.

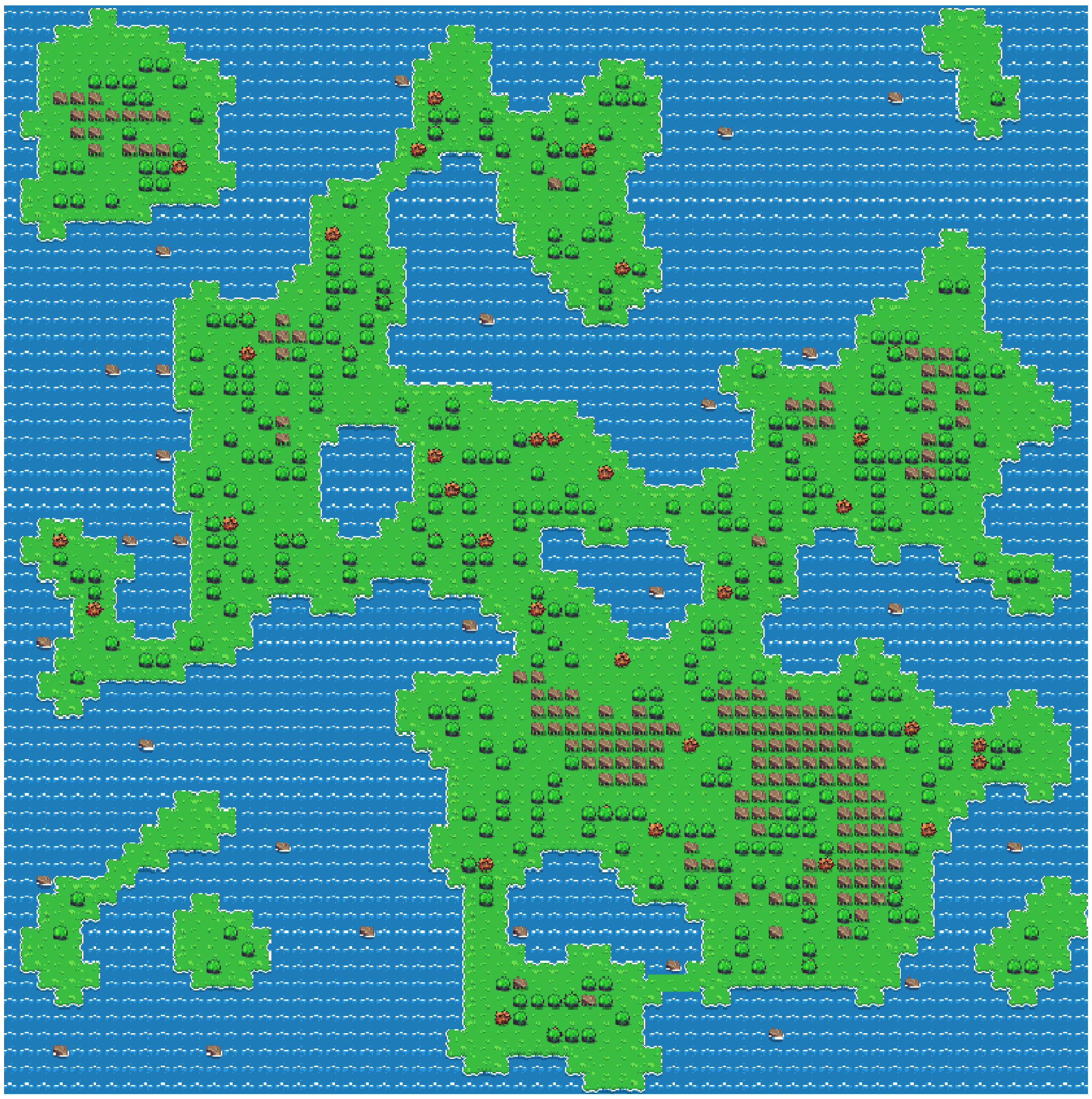


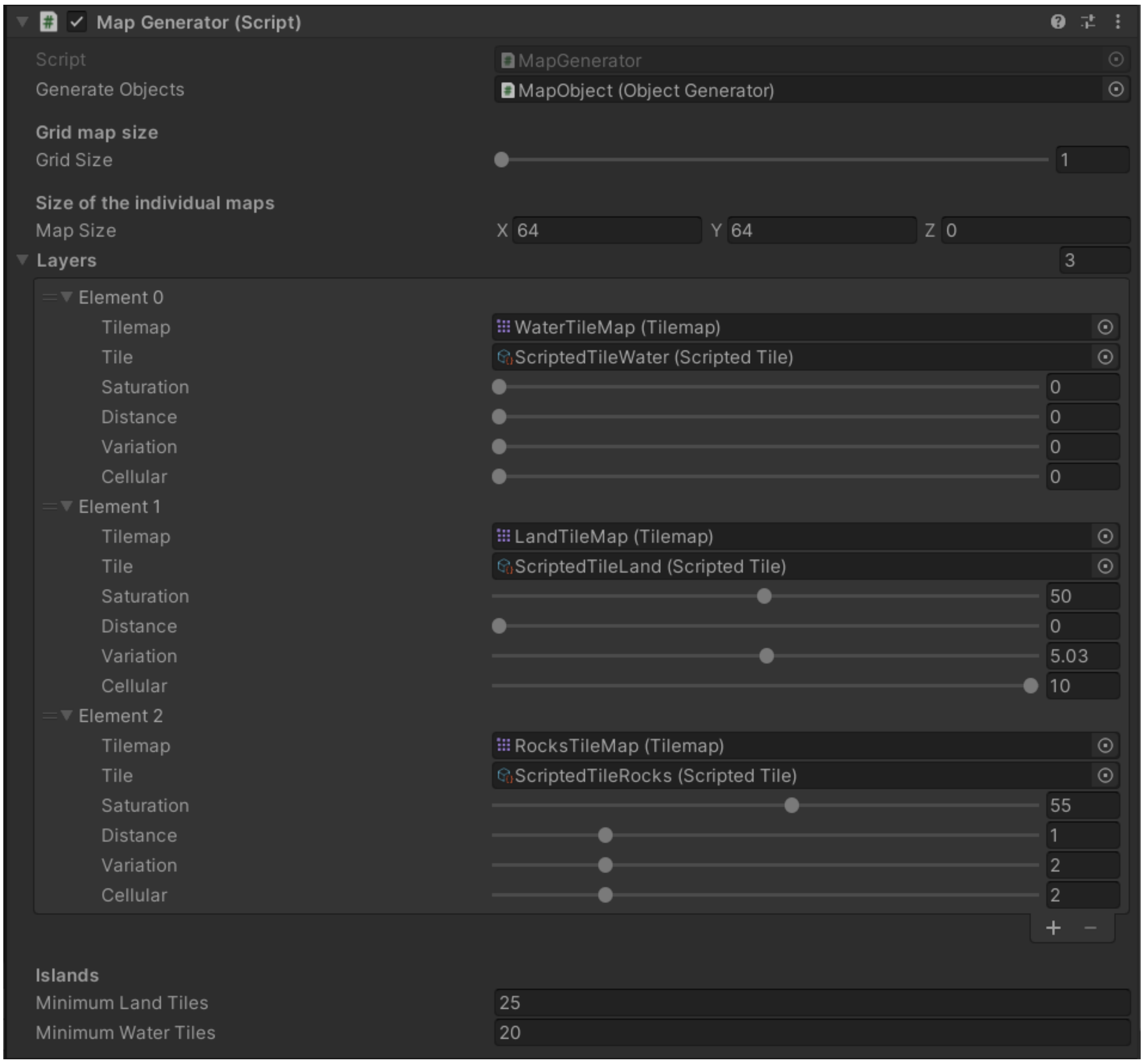
*Figure 22: Updated generated map with rock layer and game objects (from my project)*

Once implemented, users are able to easily add different game objects to populate their maps as shown above with the added trees, flowers and water rocks. All of these objects are controllable through the use of serialized variables which are shown in the user interface in the results section.

In addition to all the previously discussed functions, there was also other functions created for things such as deleting instantiated objects and layers in order to clear the map for a new one, and saving and loading functions, which I'll show some of in the results section.

## Results

Firstly, when discussing the results of the project I think it is important to show the user interface and explain what each setting does. Some of the settings in the interface aren't exclusively just for adapting the PCG as some are required from the map generator to function. Below is an image of the interface for the **MapGenerator** script.



Map Generator (Script)

| | |
|---|---|
| Script | MapGenerator |
| Generate Objects | MapObject (Object Generator) |
| **Grid map size** | |
| Grid Size | 1 |
| **Size of the individual maps** | |
| Map Size | X 64 Y 64 Z 0 |
| **Layers** | 3 |
| Element 0 | |
| Tilemap | WaterTileMap (Tilemap) |
| Tile | ScriptedTileWater (Scripted Tile) |
| Saturation | 0 |
| Distance | 0 |
| Variation | 0 |
| Cellular | 0 |
| Element 1 | |
| Tilemap | LandTileMap (Tilemap) |
| Tile | ScriptedTileLand (Scripted Tile) |
| Saturation | 50 |
| Distance | 0 |
| Variation | 5.03 |
| Cellular | 10 |
| Element 2 | |
| Tilemap | RocksTileMap (Tilemap) |
| Tile | ScriptedTileRocks (Scripted Tile) |
| Saturation | 55 |
| Distance | 1 |
| Variation | 2 |
| Cellular | 2 |
| **Islands** | |
| Minimum Land Tiles | 25 |
| Minimum Water Tiles | 20 |

*Figure 23: Map Generator script interface (from my project)*

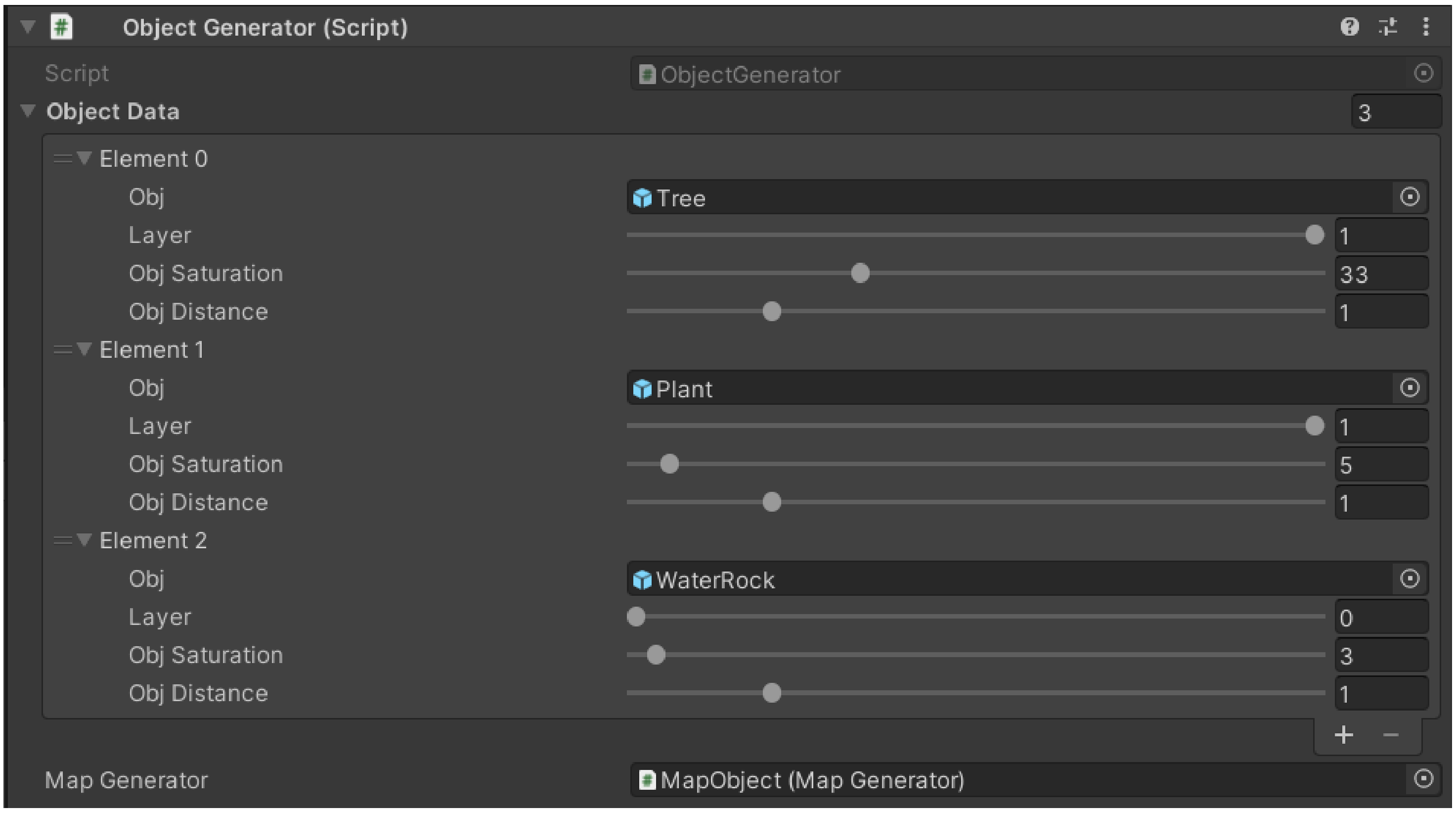
Going in order from top to bottom, firstly we have <u>Generate Objects</u>, here we just attach a simple unity game object which serves as the object for the map generator. Next is the <u>Grid Size</u>, when changing this from 1 to 2 for example, the map will now spawn a 2x2 grid of singular maps stitched together, which can be particularly useful (when accounting for the variation slider which we will get to) as it allows for some interesting differences between each map, making a more unique looking map overall. After that we also have Map Size, which allows the user to specify the size of map by the number of tiles, the image above will result in a map of 64x64 tiles.

Now we get into the terrain layers. Each layer is labelled as an element and each element features the same functionality. You can also add as many elements as you wish by clicking the plus. The Tilemap is just used for adding the specific tile map for which the layer will be draw on and Tile is the type of tile the map will print, so in my case I added the animation tile for my water layer, the rule tile for my land tiles and another scripted tile for my rocks (which have no specific rules allocated to them). Next, we have the features that will make a difference in how the generation shapes itself. This is where there is an exception, since the water layer is our base layer, we do not edit the settings and leave everything as default (zero). Saturation is used for increasing or decreasing the possibility of a particular layer's tile spawning in the map. Setting this too high will result in the layer overcrowding the map, and too low will result in the layer being extremely sparse. Distance is the minimum distance a layer's tile will be from the sublayer's sublayer, this only comes into play with the rocks layer as it is the only layer with this property, in simple terms it defines how far the rocks will be from the water. This can be utilized more if additional layers are added. Variation is the a number that defines the plus/minus of Saturation between different maps, this means that which each map generated you will have different results, for example increasing the variation of the land layer will produce maps that vary more in the amount of land that is generated. This is particularly useful for when constructing maps that are of a grid size larger that 1 as you can visibly see the difference between the stitched together maps. Cellular changes the look of the organic shape produced in a layer. Here you particularly want to have it set as high as possible the land layer and can be set lower for all further layers.

Then we have the island section, here a user can define the minimum amount of tiles an island must be to appear using Minimum Land Tiles, and the same but for the minimum tiles a body of water must be using Minimum Water Tiles.

Also, to note I did add tooltips in the interface, so that when a user hovers over it will explain each setting.

Now we'll move onto the ObjetGenerator scripts settings.



Object Generator (Script)

| | |
|---|---|
| Script | ObjectGenerator |
| Object Data | 3 |
| Element 0 | |
| Obj | Tree |
| Layer | 1 |
| Obj Saturation | 33 |
| Obj Distance | 1 |
| Element 1 | |
| Obj | Plant |
| Layer | 1 |
| Obj Saturation | 5 |
| Obj Distance | 1 |
| Element 2 | |
| Obj | WaterRock |
| Layer | 0 |
| Obj Saturation | 3 |
| Obj Distance | 1 |
| Map Generator | MapObject (Map Generator) |

*Figure 24: Object Generator Script (from my project)*

Again, from top to bottom, we have our element which is just the label of the object, you can also add as many elements as you wish by clicking the plus. The <u>Obj</u> is the game object containing the sprite of the object is attached.

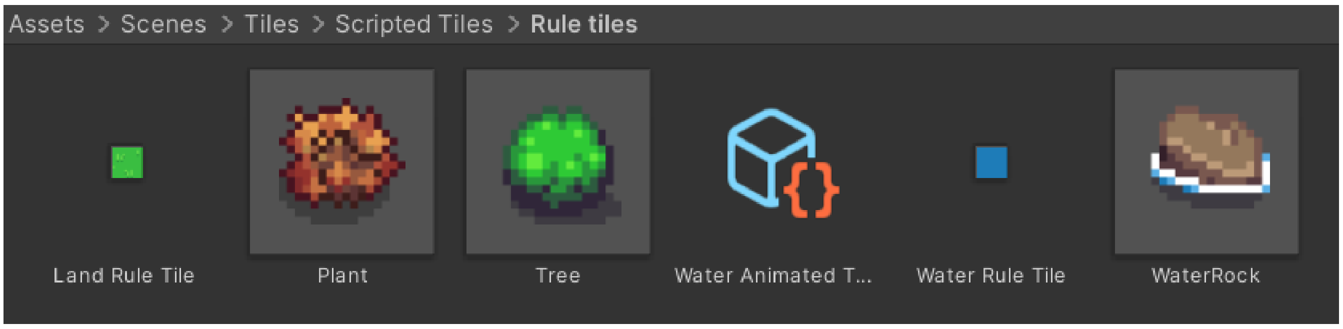


*Figure 25: Game objects (from my project)*

Layer specifies which terrain layer the object will spawn on. So, for the water rock the layer is 0, while for the Plant and tree it is layer 1. Obj Saturation works the same as in the saturation in the **MapGenerator** script by increasing or decreasing the possibility of a particular object spawning in the map. Obj Distance specifies the minimum distance between the object and sublayer tiles. Finally, Map Generator, here we attach the same game object attached to the **MapGenerator** script, this is here so that the two scripts can function together.

Also, to mention, I added the functionality of generating a new map with the spacebar, saving maps with the 'S' key and loading maps with the 'M'.
Here is an image of some of the maps saved to a folder:

*Figure 26: Saved maps (from my project)*

If I was to do the project again, I might try to add more functionality and options for a user to give them further control, as well as integrating the user interface into the game itself through the implementation of a menu.
Overall, I believe the result of the project has sufficiently demonstrated each of the areas outlined in my projects brief, of researching into the various techniques and tools that are currently used (with a particular focus on the virtual environments such as found in games) followed by the creation of a tool/application that demonstrates PCG in operation. I also completed the task I set out to do myself, by allowing the user to have an influence on the PCG in operation as well as being a learning tool to the user.

## Conclusion

In conclusion, the exploration of procedural content generation techniques and their implementation in Unity has been an invaluable learning experience for me. This project has not only resulted in giving me a better understanding of Unity's capabilities and tools, but it has also fulfilled the goal and ambitions that I set out for myself at the beginning of this project. The process of development of my 2D map that I undertook has honed my coding skills and improved my overall game development proficiency, which will undoubtedly be beneficial in my future endeavours as a computer scientist.

Through extensive research and developing this 2D map generator that makes use of PCG algorithms has led me to have a greater understanding of PCG as a whole and its importance in the gaming industry. The insights I have gained from this research will serve me as a solid foundation for future projects and could potentially be applied in innovative ways to create immersive and engaging game experiences.

The completion of this project highlights the power of PCG. By leveraging these techniques and tools, the possibilities for generating unique and dynamic game content are virtually endless. As a result, this project has further ignited my passion for procedural content generation and its potential applications in the gaming industry.

## References:

Aidan McInerny (2015) *Generating A Forest with L-Systems in 3D*, *https://www.csh.rit.edu/~aidan/portfolio/3DLSystems.shtml*.

AlgoMonster (no date) *Flood Fill*, *https://algo.monster/problems/flood_fill*.

ArMM1998 (2017) *Zelda-like tilesets and sprites*, *https://opengameart.org/content/zelda-like-tilesets-and-sprites*.

Bay 12 Games (2006) 'Dwarf Fortress', *Bay 12 Games* [Preprint].

Browne, C. (2008) *Automatic Generation and Evaluation of Recombination Games*.

Chudasama, D. *et al.* (2015) *Image Segmentation using Morphological Operations*, *International Journal of Computer Applications*.

Cornell University (no date) 'Conway's Game of Life', *http://pi.math.cornell.edu/~lipa/mec/lesson6.html* [Preprint].

Costa, L. and Oliveira, P. (2002) 'An evolution strategy for multiobjective optimization', in *Proceedings of the 2002 Congress on Evolutionary Computation, CEC 2002*. IEEE Computer Society, pp. 97–102. Available at: https://doi.org/10.1109/CEC.2002.1006216.

digitalfreepen (2017) *THE RANGE OF PERLIN NOISE*, *https://digitalfreepen.com/2017/06/20/range-perlin-noise.html*.

Eric Frederiksen (2021) *Minecraft Biome Guide - All The Warm Biomes*, *https://www.gamespot.com/articles/minecraft-biome-guide-all-the-warm-biomes/1100-6495679/*.

Francesco Berto and Jacopo Tagliabue (2017) *Cellular Automata*, *https://plato.stanford.edu/entries/cellular-automata/*.

Gardner, M. (1970) 'Mathematical Games', *Scientific American*, 223(4), pp. 120–123. Available at: https://doi.org/10.1038/scientificamerican1070-120.

Gazmeh Khaje Nasir, H. *et al.* (2014) *Spatio Temporal Forest Fire Spread Modeling Using Cellular Automata Honey Bee Foraging and GIS*. Available at: http://www.bepls.com.

Gearbox Software (2009) 'Borderlands', *2K Games* [Preprint].

Goldberg, D.E. and Holland, J.H. (1988) ‘Genetic Algorithms and Machine Learning’, *Machine Learning*, 3(2/3), pp. 95–99. Available at: https://doi.org/10.1023/A:1022602019183.

Gustavson, S. (2005) *Anti-Aliased Euclidean Distance Transform View project WebGL-noise View project*. Available at: https://www.researchgate.net/publication/216813608.

Harold Abelson and Andrea Disessa (1981) *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*.

Infinity Ward (2007) ‘Call of Duty 4: Modern Warfare’, *Activision* [Preprint].

Julian Togelius *et al.* (2013) ‘Procedural Content Generation: Goals, Challenges and Actionable Steps’.

KEVIN PURDY (2022) *Dwarf Fortress on Steam gets release date, trailer, and graphics beyond ASCII*, *https://arstechnica.com/gaming/2022/11/dwarf-fortress-on-steam-gets-release-date-trailer-and-graphics-beyond-ascii/.*

Mathuranathan (2020) *Statistical measures for stochastic signals*, *https://www.gaussianwaves.com/2020/08/statistical-measures-for-stochastic-signals/.*

Mojang (2009) ‘Minecraft’, *Mojang AB* [Preprint].

Perlin, K. (1985) ‘An image synthesizer’, *ACM SIGGRAPH Computer Graphics*, 19(3), pp. 287–296. Available at: https://doi.org/10.1145/325165.325247.

Prusinkiewicz, P. (no date) *Graphical applications of L−systems*.

Prusinkiewicz, P. and Lindenmayer, A. (1990) *The Algorithmic Beauty of Plants*. New York, NY: Springer New York. Available at: https://doi.org/10.1007/978-1-4613-8476-2.

Shaker, N., Togelius, J. and Nelson, M.J. (2016) *Procedural Content Generation in Games*. Cham: Springer International Publishing. Available at: https://doi.org/10.1007/978-3-319-42716-4.

Smith, G. (2021) ‘The Future of Procedural Content Generation in Games’, *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 10(3), pp. 53–57. Available at: https://doi.org/10.1609/aiide.v10i3.12748.

Summerville, A. *et al.* (2018) ‘Procedural Content Generation via Machine Learning (PCGML)’, *IEEE Transactions on Games*, 10(3), pp. 257–270. Available at: https://doi.org/10.1109/TG.2018.2846639.

Thomas H. Cormen *et al.* (no date) ‘ Introduction to Algorithms, Fourth Edition’.

Togelius, J. *et al.* (2011) ‘Search-Based Procedural Content Generation: A Taxonomy and Survey’, *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, 3(3). Available at: https://doi.org/10.1109/TCIAIG.2011.2148116.

Unity technologies (2017) *2d-extras*, *https://github.com/Unity-Technologies/2d-extras*.

Weizenbaum, J. (1983) ‘ELIZA — a computer program for the study of natural language communication between man and machine’, *Communications of the ACM*, 26(1), pp. 23–28. Available at: https://doi.org/10.1145/357980.357991.

Yannakakis, G.N. and Togelius, J. (2018) *Artificial Intelligence and Games*. Cham: Springer International Publishing. Available at: https://doi.org/10.1007/978-3-319-63519-4.