Procedural Generation of 2D Terrain to Create a Video Game World

Matthew Kenyon[1]
School of Computing Science, Newcastle University, UK

Abstract
This dissertation presents the development of a procedural content generator for a 2D game world in Unity, using techniques to create a varied and immersive environment. The program employs Perlin noise and cellular automata, as well as other custom algorithms to generate many terrain features, such as biomes, ore veins, cave systems and flora. The project also integrates a detailed lighting system featuring top-down occlusion and simulated light penetration to enhance visual depth.

To improve performance due to the importance of speed, several optimisation strategies were investigated and applied, such as selective tile updates and more efficient data structures, resulting in a much quicker execution time.

The final system demonstrates how procedural generation can be used to build complex game worlds efficiently and dynamically.

Keywords: Procedural Generation, randomness, Perlin noise, cellular automata, light mapping

1. Introduction

A game world is a virtual environment in which a video game takes place [1]. It is a place of imagination whose aim it is to immerse the player and make them feel in control of this environment, with their decisions and actions affecting everything around them [2]. Game worlds can vary widely in their design and structure. For example, Minecraft features a procedurally generated world, offering vast, randomised terrain with each new game. In contrast, The Legend of Zelda: Breath of the Wild presents a handcrafted world, meticulously designed by developers to guide exploration and narrative. Some games, like Skyrim, use a hybrid approach, blending procedural generation with handcrafted elements to achieve both scale and controlled design.

Handcrafted content is the kind of content that is intentionally made. There is a level of quality in handcrafted content that only humans can achieve. A predictable, premade map also leads to environments that feel more natural [3].

On the other hand, procedurally generated content has the potential to create limitless re-playability for a game. It also allows the developer to delegate content creation to a machine and focus on other areas of development. This is especially

[1] Email: m.kenyon2@ncl.ac.uk

useful if the development team is small. However, if not implemented correctly, it can be messy, imprecise and buggy, which is why it's quite difficult to design perfectly.

This paper will explore the ways procedural generation and other randomness techniques can be implemented to generate natural looking 2D worlds. It will first examine procedural generation, and the qualities and attributes needed to create a successful procedural content generation. The way these features are implemented in other games will also be analysed to better understand each feature. The features will then be implemented and tested to ensure they fit the criteria of the generator, and they will be optimised to ensure the content is generated as efficiently as possible.

Aims and Objectives

The overall aim of the project is:

- To design and implement an advanced procedural generation algorithm that is capable of creating an immersive, realistic underwater 2D world.

To achieve this goal, smaller objectives have been included to complete:

- Generate terrain and relevant biomes: Multiple biomes with their own terrain height, background elements and other distinct features should be generated. Each biome should be present in every generation, and transitions between biomes should be smooth to maintain visual coherence.
- Generate ores: Ore veins should be distributed throughout the underground, with different ores being more prominent at different depths. The generation should produce cohesive, naturally shaped clusters.
- Generate caves: Caves should be generated throughout the depth of the map, allowing for plenty of exploration underground, but should be carefully integrated to avoid disrupting the natural appearance of the surface terrain
- Generate light maps: Light maps should be implemented to simulate realistic lighting across the world.
- Generate flora: Each biome should include appropriate foreground and background elements to bring the world to life.

2. Background Research

Procedural Generation

Procedural content generation in games refers to the creation of game content automatically using algorithms [4]. It is a popular method for generating a range of content within game development, and is a preferable method for a number of reasons [5]. It removes the need to have a human designer or artist generate the content, therefore making it possible for indie developers to create content rich games. It also allows for content to be generated as its played, so it can be infinite, and also allows

for personalisation so content can be adapted to the way the user plays. It also encourages more creativity and allows for more control.

Taxonomy of Procedural Content Generation

Due to the variety of procedural content generators that are designed, it is important to have a structure that highlights the differences and similarities between approaches. This highlights the best situations to adapt these approaches.

Online vs offline

Procedural content generation techniques can be used to generate content online, meaning that the player is currently playing the game and in the game world. This allows the generation of endless variations, making the game infinitely re-playable and allowing for player adapted content. It also allows for constant generation, presenting the opportunity for infinite world generation.

Procedural generation can also be performed offline. This takes place during the development of the game, before the player is entered into the game world. This is useful when generating complex content that takes a while to generate, like generating the whole game world at once [5].

Generic versus adaptive

Generic content generation refers to the paradigm of procedural generation where content is generated without taking player behaviour into account, whereas adaptive generation analyses the player's previous behaviour and generates the content accordingly. This makes a much more tailored experience for the user, with potentially scaling difficulty or different experiences, adding to the re-playability factor [5].

Stochastic versus deterministic

Deterministic procedural generation allows the regeneration of the same content given the same starting point and method parameters. This is normally through the use of a randomly generated seed. Stochastic procedural generation does not allow the creation of the same content, even if the parameters are the same at the start [5]. This means the same generation can never be recreated, which makes testing and debugging much more difficult.

Constructive versus generate-and-test

In constructive procedural generation, the content is generated in one pass, whereas generate-and-test techniques alternate generating and testing in a loop, repeating until a satisfactory solution is generation [5]. Search based procedural generation is a special type of generate-and-test. Instead of just rejecting a generation and regenerating a new one, the pieces of content are tested using a fitness function to see

how well they have been generated, meaning each generation can improve on the previous one and eventually be successful [6].

Due to the iterative nature of generate-and-test, it can take longer to generate the correct results, however constructive generation, while quicker, might not generate properly.

Requirements of Procedural Content Generation

As mentioned before there is a precise nature about creating a procedural generation tool. To achieve this task, there are a number of desirable properties all procedural programs should have that makes them efficient and successful.

Necessary vs auxiliary content

Procedural generation can be used to generate 2 types of content, necessary and auxiliary. Necessary game content is the content that is required for the completion of a level, which is required for every generation. Auxiliary is the unimportant content that doesn't necessarily have to show up in the generation. It can be used to generate auxiliary content that can be discarded or exchanged for other content. Necessary content should always be present while this condition does not hold for optional content. Necessary content is required to make the game work function as intended, and if it is not included, the game world may be missing a key part that would hinder the player's progression.

This ties into the importance of constructive and generate-and-test generation techniques, as there needs to be a method to ensure these necessary elements are always incorporated in the generation.

Degree of control

Content generators need to be controllable so a human or algorithm can specify some aspects of the content to be generated. The use of a random seed is one way to gain control over the generation space. This is due to the deterministic nature of procedural generation. A set of parameters that control content generation can also be used to allow control, and adjusted as seen fit to achieve the required generation.

Speed

Requirements for speed vary depending on whether the content is generated during gameplay or during development. For example, a game like Terraria will take longer to generate, as the whole world is being created at once out of the gameplay. However, for Minecraft, the generation needs to be as quick as possible, as it takes place during gameplay and has an effect on the user's active gameplay. However, the generator always should be as quick as possible, as the user doesn't want to be waiting a long time for the game to start.

Diversity

There is also a need to generate a diverse set of content to avoid the content looking like it's all minor variations of the same theme. If the content is repetitive or predictable, it won't be as interesting to interact with, and also takes away from the re-playability feature. This is why it is important to implement many different forms of randomness.

Randomness

Randomness introduces uncertainty in games, enhancing unpredictability and variety. In game development, creation involves introducing something new into the game, and randomisation can be used to alter amounts, quantities and positions of those elements.

Traditionally when creating video games, developers have had to manually design the entire gaming world. This limits how large and detailed that game world can be. To avoid this problem, developers rely on procedural generation where the computer can create new areas based on randomisation. Through procedural generation of new areas, video games can offer new, limitless worlds for players to explore, which increases re-playability of the game [7].

To implement procedural generation, some sort of randomness has to be included. Using a few parameters, the application of procedural content generation ensures the creation of a high number of possible different contents of a game [8].

Computer random number generators rely on a seed value. Based on this seed value, the random number generator algorithm can generate multiple numbers. While these numbers appear random, the limitation is that giving a random number generator the same seed results in the same list of random numbers [7]. However, this follows the deterministic approach explained in the taxonomy of procedural generation previously. The seed is useful for procedural generation, as it can be utilised to work on a singular generation or go back to a previous run and check for adjustments that need to be made.

Perlin Noise

Perlin noise is another form of randomness that is very helpful for procedural generation. It can be used to provide an organic feel to visual effects. It generates a form of randomness that is not uniform or normal. It can be described as coherent randomness, where consecutive numbers are related to each other, providing a smooth feeling between values [9]. This is why it is perfect for generating natural looking patterns, such as 2D landscapes and textures [10].

Perlin noise operates by assigning a random unit vector to each point on a grid. A unit vector is just a direction with a length, or magnitude, of 1. When sampling the noise, the algorithm computes the position's distance from nearby grid points. At each nearby point, it calculates a dot product between the random gradient vector and the offset vector, which is the difference between the position and the grid point.

After calculating the dot product at each corner of the grid square, the values are interpolated based on the sample's position inside the square to produce the final noise value [11]. After many points have been sampled, the program produces an image like the one shown below.

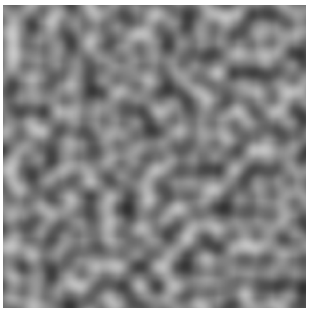

Figure 1: Image of Perlin

As mentioned before, a key concept of procedural generation is controllability, therefore it is helpful that there are several controls that allow customisation of the randomness of noise, all based around the octaves. The number of octaves refers to multiple layers of noise combined together, each with a different frequency and amplitude. Lower octaves offer larger swings in the signal, while higher octaves offer more fine-grained noise. Any range of octaves can be used. The higher the amplitude, the more that octave will influence the final signal. The choice of interpolation is important too; however, the S-curve function is commonly used in Perlin noise [9].

Cellular Automata

Cellular automata is also a procedural generation algorithm that simulates the behaviours of cells in a 2D grid. Each cell has 2 states, which are dead or alive, which depend on the status of their neighbours. Live cells with fewer than 2 live neighbours are underpopulated and die. Live cells with 2 or 3 live neighbours keep living in the next generation. Live cells with more than 3 live neighbours are overpopulated and die. Dead cells with 3 live neighbours become alive again due to reproduction. These rules are powerful enough to generate maps that look like natural environments, such as caves [10].

Success Criteria

There are multiple reasons why it is important to evaluate the procedural content generation [5]:

- To better understand the capabilities of the generator
- To confirm guarantees can be made about generated content. If there are necessary elements in the generated content, it is important to ensure that those qualities are present
- To easily iterate upon the generator by seeing whether it is capable of matching the programmer's intent

The most important concept to remember when thinking of how to evaluate a generator is to make sure that the method used to evaluate the generator is relevant to what it is being investigated and evaluated. There are 2 main ways to evaluate procedural generators; top-down and bottom-up evaluation.

If a content generator is capable of creating thousands of unique generations, it is not feasible to view all of the output to judge whether the generator is performing as desired. However, with enough generations to use as a sample, and the data can be recorded for each generation, you can be relatively confident with the results.

Bottom-up works through acting as the player and experiencing the generations themselves. This is mainly done by just viewing and evaluating the world and checking that the generations look and act as intended.

Light Maps

Light maps contain data about how light works within the 2D world. The best way to implement a light map within Unity is through the use of Shaders.

Shaders are a powerful tool to make stunning landscapes. A shader performs operations that turn graphical data into the desired final image. In this case, it will calculate the appropriate levels of light, darkness and colour of the graphical data. Each shader program has a different scope, and takes in and outputs different types of data. Vertex shaders process individual vertices by transforming their position and preparing data for later stages in the pipeline. Geometry shaders operate on entire primitives, optionally generating new ones based on the input. These are only required if the program needs to manipulate or generate geometry. Most shaders, especially simple unlit or post-processing shaders, skip them. Fragment shaders determine the final colour of each pixel using interpolated data and texture information [12].

Game Examples

To gain a better understanding of these practises, it is important to investigate how they have been implemented in similar games. This analysis can show how these techniques can then be implemented into my own work.

Terraria

Terraria is a 2D open world sandbox / platformer in which the player controls a single character in a pre-generated world [13]. It is a prime example of a game that generates an impressive, immersive procedurally generated 2D terrain.

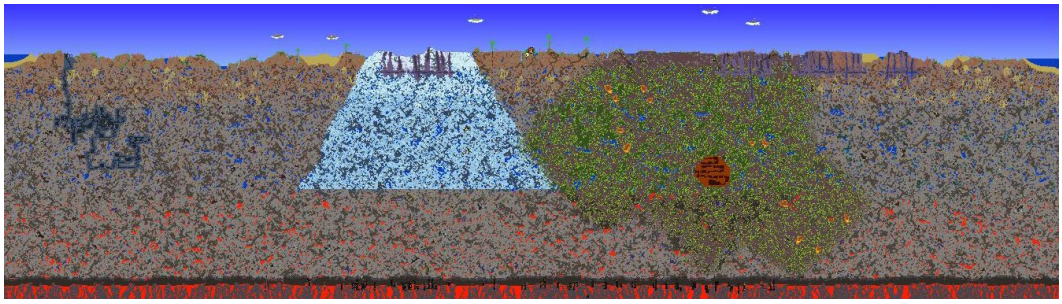

Figure 2: World generation in Terraria

The world in Terraria is generated deterministically through the use of a random seed to ensure no two worlds are identical. The generation is offline, as it takes place before the user leaves the menu. While a new random world is being generated, various texts appear over the loading bar, showing what is being generated in the world. This gives a good idea of the order of stages that go into the creation of the world. The rough terrain is generated first, with hills then being created with noise. Then the deeper layers of dirt and rock are made. This is followed by areas being hollowed out for cave generation, and ores are placed around the map. Plants are then implemented above and underground, then finally the whole world is cleaned up so everything is implemented correctly. It ensures all necessary elements are generated during these stages.

Terraria utilises different world sizes to tailor to the user's needs [14]. The sizes (in tiles) are 1750 x 900, 4100 x 1200, 6400 x 1800 and 8400 x 2400. The smallest world will be used as a base for the program, as it will be far less advanced than the generation found in Terraria. Using a smaller world size ensures that experiments can be evaluated on a more manageable scale. This is ideal, as investigation and execution times are faster, and the data being fetched is still accurate.

Worlds are structured into layers and biomes to provide various content depending on the player's location [15]. Biomes are the different types of areas that any Terraria world can contain [16]. Every biome has its own characteristic terrain blocks, items, backdrops, creatures and so on.

Figure 4: Image of Forest Biome

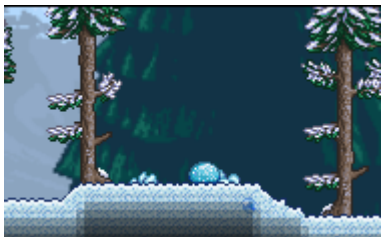

Figure 3: Image of Snow Biome

A world in Terraria is divided into 5 layers, each appearing at different depths. Each section has its own characteristics, and most layers contain multiple biomes [17]. The most important difference between the layers is the population of caves, and also the frequence of ore. On the surface layer, there are very few caves and little ore to mine, encouraging the player to explore deeper. This is because in the cavern layer there are lots more ores, and rarer ones deeper down, as well as more caves and open space. Game balance involves risk vs. reward. The higher the risk, the higher the reward [7]. In Terraria, the deeper the terrain, the higher the reward, due to the structure of the layers.

Minecraft

Minecraft is a 3D sandbox game where players interact with a fully modifiable three-dimensional environment made of blocks and entities [18]. Even though Minecraft is

a 3D world, it will still be useful to investigate the techniques that it uses so that they can be implemented into the generation of the 2D world.

Minecraft generates terrain dynamically as the player explores, allowing the world to expand infinitely. As mentioned previously, procedural generation must ensure the inclusion of necessary features for a playable world, which is online. In Terraria, these must be present from the start, however in Minecraft, the infinite nature of the world means that required features can appear later through continued exploration, making it less of a concern for them to be generated immediately. This is due to them always eventually showing up through probability.
Both Minecraft and Terraria are generic, meaning that they are generated randomly and not in accordance to the user, their generation is purely deterministic based on the seed used.

The Minecraft world is also split into biomes, each with distinct geological features, flora, water, grass and foliage colours. Biomes separate every generated world into separate environments [19]. Some of these environments include underwater biomes, shown below, which are more similar to the project that is intended to be created.

Figure 5: Image of Warm Ocean Biome

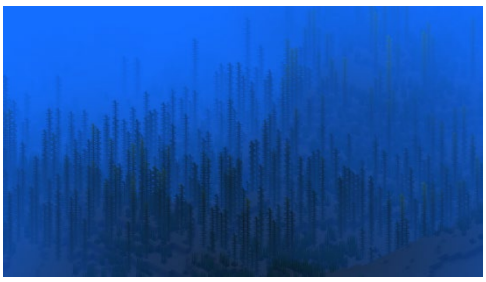

Figure 6: Image of Deep Lukewarm Biome

These biomes, which exist at different depths underwater, show how the lighting reduces with depth, as less light can penetrate through the water. It also shows that there are certain objects that travel a long way off the ocean floor, adding depth to the biome.

Minecraft and terraria are very similar in the way they deal with ores. The areas that are at greater depths in the game world have more valuable treasure and loot. This is shown in the graph below.

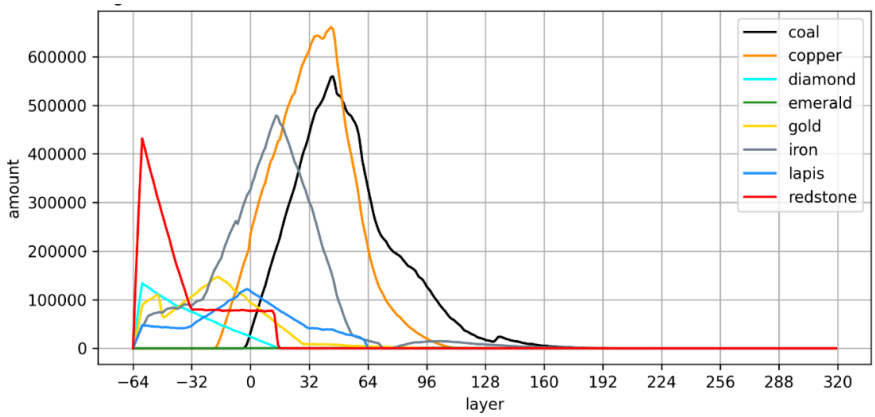

Figure 7: Graph showing distribution of ores at different depths in Minecraft

The graph shows the distribution of ores in a Minecraft world. The rarer, expensive ores like diamonds and lapis are more frequent at lower points, whereas common ores like copper and coal are more frequent at higher depths.

Minecraft has very advanced programming for generating caves, with many different types of caves to add variation within the 3D environment. The most frequent of which are noise caves. These are generated by noise, and by adjusting noise frequency, hollowness and thickness, noise caves can vary in diverse ways. These caves also contain pillars to make caves more realistic [20].

Research Summary

After researching game worlds and all the key principles that are involved with bringing them to life, they can be used to start creating the project world. By investigating how these features are implemented into popular games that are similar to the intended project, it gives me a good idea how to use them within my own work.

I decided that the project would be an offline generation, similar to Terraria, so it doesn't require user data to generate. It will also be deterministic, as the importance of using a seed for generation has been explained throughout the research.

3. Implementation

Structure

The program is set out modularly, with each section being given their own function being called sequentially. This allows for the stages to be separated and easily adapted and monitored. This helps with control, with each section being able to toggle on and off, and their parameters to be adapted to be optimal, which was highlighted to be a very important feature through the research. It also helps to fine tune previous sections without affecting other perfected parts of the generation.

The sequence of generation loosely follows Terraria's, with some of the stages being swapped around to work better with the project. It also has a set world size, with the whole terrain being generated at once, making it an offline generator. It will also be deterministic, using a random seed to create randomness. This makes it easy to extract data and analyse the world and its generation as a whole.

Biome Generation

Biome generation plays a crucial role in creating a diverse and functional game world. It defines the structure and attributes of different regions, directly impacting terrain shaping, resource distribution, and player experience throughout exploration. The biomes generated will have an effect on the terraforming of the relevant areas, therefore I believed it to be necessary to complete this step first.

The game uses 3 different biomes, which each have a specific depth where the terrain tiles are positioned. The biomes are generated from the centre of the game world where $x = 0$. The program then works up and down on the x-axis up until the

size of the game world is reached. The generation process begins at this point to ensure the same starting biome is always placed there. This is similar to Subnautica, where the player spawns in the "Safe Shallows" biome [21]. This is a calm and relatively hazard-free area designed to ease players into the game. Likewise, starting in a designated safe biome ensures a gentle introduction for the player.

Biomes are generated by selecting a random width and assigning that portion of the x-axis to the corresponding biome. Each biome has a minimum and maximum width. The width of the next biome is generated when the previous one ends. This allows the biomes to be an appropriate size so that all the relevant objects can be placed in later on with enough space.

To make changes between biomes appear more natural, there are set transitions between them to make sure that they can only move up or down to the next depth value, so the slopes are not steep. This means that biome 1 and 3 cannot transition to each other, but all other transitions are available.

To evaluate the generation of biomes 1,000 worlds were generated and the presence of each biome in each generation was recorded. The results of the generations are shown below:

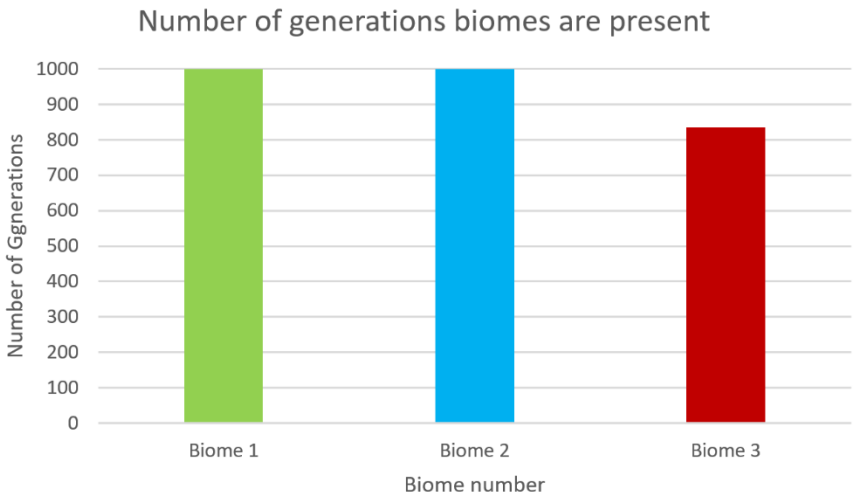

Figure 8: Graph showing the number of generations biomes are present per 1,000 generations

Biome 1 and 2 generated as expected, as biome one is always used at the beginning as the safe area at x=0, and due to the way biome transitions work, biome 2 is always generated second, as it always transitions from biome 1. Biome 3 is evidently a lot less prevalent within the world generations, only showing up 83.5% of the time. Even though this percentage is very high, all biomes need to be present, as it is the assumption that each biome is necessary, shown in the research.

To address this problem, there are 2 approaches that could be taken, which are constructive and generate-and-test, both of which are discussed in the research. Generate-and-test could be used to ensure that all 3 biomes were present by generating the world and verifying their existence. If a biome is missing, the bounds

would be regenerated. The issue with this is that on the small chance that the biomes kept failing to generate together, the time to execute could be made a lot longer, especially if more biomes were introduced and the program was more advanced. On the other hand, constructive could be utilised, which just completes one generation, which could be adapted to enforce all the requirements, in this case all the biomes being present.

The constructive approach was implemented, which meant that the percentages were adjusted that dealt with choosing the biomes. Previously the probability that a biome was chosen was:

$$Biome\ Probability = \frac{1}{Number\ of\ biomes\ available}$$

This evidently enables the small chance for 2 biomes to bounce between each other. To change this, the probability for different biomes changes as the x value gets larger. When the x is small, biome 1 is most likely to be chosen, and its probability decreases as the x increases. Biome 2's probability starts low, then when it gets to the middle of the x-axis it is at its highest, then decreases again as it gets larger. Biome 3's probability also starts small, but increases when the x is largest. Along the x-axis, each biome has a section where it has a 100% chance to spawn, therefore enforcing each one's inclusion.

$$Biome\ 1\ Probability = \begin{cases} 1, & x < 100 \\ 1 - \frac{x - 100}{150}, & 100 \le x \le 250 \\ x, & x > 250 \end{cases}$$

$$Biome\ 2\ Probability = \begin{cases} 1 - 2\left(\left|\frac{x}{350} - \frac{1}{2}\right|\right), & 50 \le x \le 400 \\ 0, & x < 50\ and\ x > 400 \end{cases}$$

$$Biome\ 3\ Probability = \begin{cases} 1, & x > 300 \\ 1 - \frac{x - 300}{100}, & 300 \le x \le 400 \\ 0, & x < 300 \end{cases}$$

This method also allows overlap for biomes which allow for more variation. This also works with the specified biome sizes which always stay in a relevant proportion so the correct biomes are generated and no incorrect transitions happen as a result.

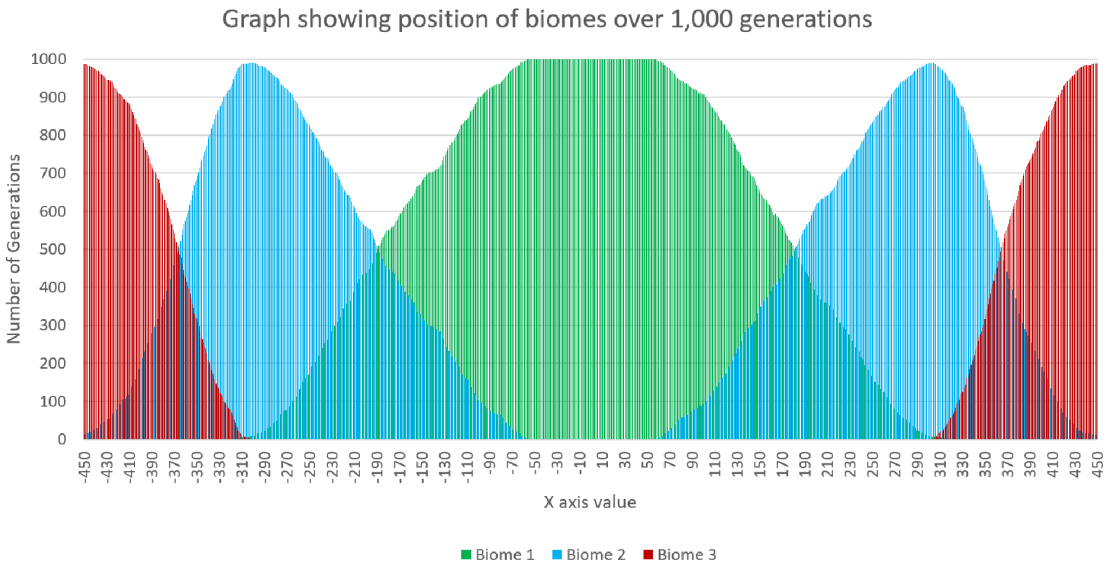

Figure 9: Graph showing number of generations biomes are present per 1,000 generations

The distribution chart shows the average positions of the biomes along the x-axis per 1,000 generations. This shows that the positions of the biomes are still kept relatively random, with some biomes overlapping, but ensures that all biomes are present, as now there are no generations missing any of the biomes.

Ore Generation

In procedurally generated games, the placement and distribution of resources play a critical role in shaping player exploration and progression. Ore generation, in particular, serves as a core mechanic that encourages deeper traversal into the world by rewarding players with valuable materials.

Ore veins are generated through the depths of the world through the use of Perlin noise. By sampling Perlin noise across the horizontal and vertical axes of the world space, regions with similar noise values emerge. If the values of these tiles are above a certain threshold, they are selected as ore tiles. The continuity of Perlin noise ensures that nearby positions in the grid have similar values, resulting in ore veins that mimic natural distributions. These veins are then randomly allocated one of three ores, and a flood-fill algorithm is used to fill the selected area with the same ore, making the veins consistent.

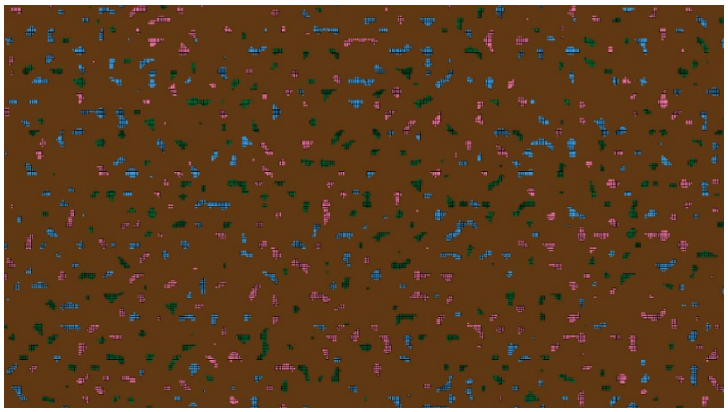

Figure 10: Ore generation with 3 different ore types

The ore generation is fully functional as it is, however, after researching other games, it is apparent that more ores should be more frequent at greater depths due to high risk and high reward [7].

To implement this, the threshold value that is used for Perlin noise should increase with greater depth. This not only causes there to be more ores at deeper points, but due to the consistent nature of Perlin noise in 2D, it causes the ore veins to increase in size, making them much richer. The equation to achieve this is shown below.

$$Threshold\ value = 0.9 - \frac{0.25 * (-y)}{450}$$

To ensure this implementation worked as expected, the frequency of ores at each depth value in the y-axis were recorded over 1,000 generations. The graph below shows the recorded frequencies due to the adapting threshold value.

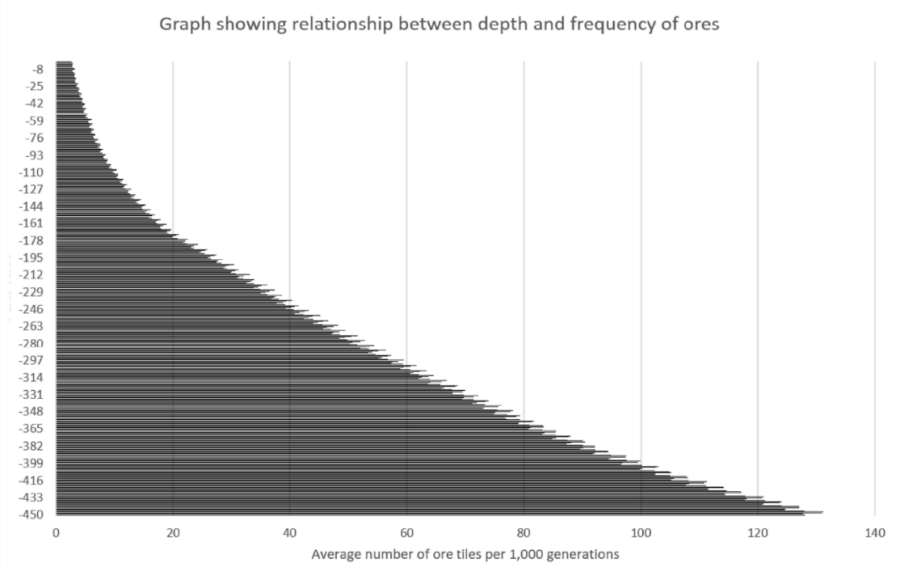

Figure 11: Graph showing relationship between depth and frequency of ores per 1,000 generations

The data clearly shows that the further down the y value is, the more likely it is for ores to be spawned.

To further enhance the ore generation system, more valuable ores are introduced at greater depths, reflecting the high risk, high reward principle. To implement this functionality, the threshold value used to determine whether a tile should contain ore, based on its Perlin noise value, should now be specific to each ore type. This means that at certain points, the threshold value will be met by some ores and not others, making the more common ore more prevalent across the world.

$$Ore\ 1\ Probability = \frac{y}{450}$$

$$Ore\ 2\ Probability = \begin{cases} 0, & y > 100 \\ \frac{y - 100}{350}, & 100 \leq y \end{cases}$$

$$Ore\ 2\ Probability = \begin{cases} 0, & y > 250 \\ \frac{y - 250}{100}, & 250 \leq y \end{cases}$$

The graph below shows how these equations are used to generate ores at their relevant depths over 1,000 different generations.

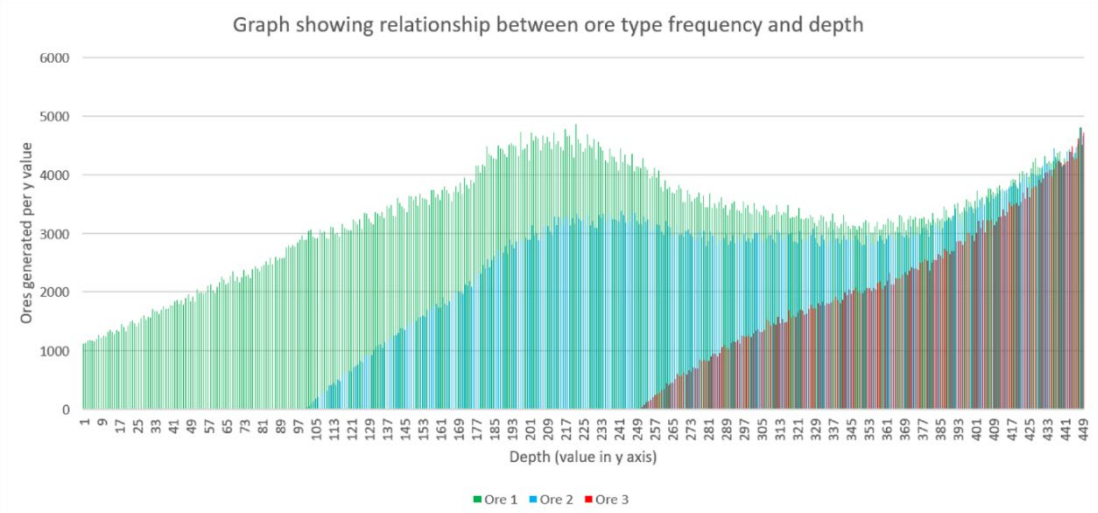

Figure 12: Graph showing relationship between ore type frequency and depth across 1,000 generations

The graph shows how the increasing frequency of ores and changing frequency of type of ores work together. The total number of ores is always increasing, however, due to more ores being available at deeper depths, the probabilities change too, which is why ore 1 becomes less prominent, and eventually why ore 2 becomes less frequent as well. This can be seen in the image below, which shows the distribution of ores in a single world generation.

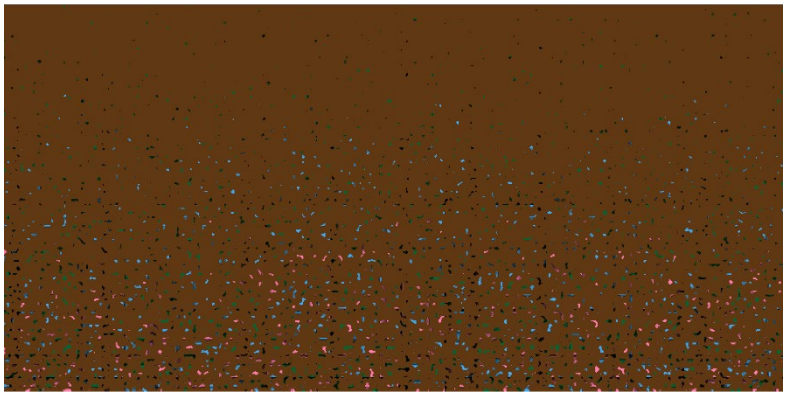

Figure 13: Ore generation with depth adjustments

Currently to generate random Perlin noise each time, an offset is added in the x and y directions so that a different type of noise is used. To ensure that this generation works with a random seed, this offset is changed to the value of the seed. Due to the deterministic nature of Perlin noise, the same values will be generated with the same seed.

As the ore types are randomised separately, it needs to ensure that the ore types are also the same in each generation. This is done by setting the seed for Unity's built-in pseudo-random number generator. This means that the program starts its random number sequence from a specified point by the seed, meaning the same sequence of random values are used.

Terrain Generation

At this point in the generation, a world has been established with appointed biomes and ores generated throughout. The biomes can now be used to determine the height of the terrain. Since each biome has a predefined depth, the terrain can be accurately shaped by traversing the x-axis and adjusting the y position of tiles accordingly.

Due to the adjusted heights, there are steep changes in altitude where different biomes meet. To ensure this isn't the case, the terrain has to transition between biomes so it gradually increases or decreases depth between neighbours. The width of these transition slopes is randomised to produce naturally varied gradients between biomes. This process is shown in the figure below.

To make the world feel even more natural, the top layer needs to be rugged and hilly, rather than just being flat. This feature is implemented through the use of Perlin noise.

For each position along the x value, the function samples the Perlin noise function. The value that the function returns contains a value between 0 and 1, which is the normal return for a Perlin noise function. To allow for vertical displacement both above and below the current terrain, the value is remapped between -1 and 1 using the equation. This provides a value that is then rounded to the nearest integer, which can be added to the terrain's current y value to move the surface up or down

smoothly so there are hills and dips. The strength of the noise can also be controlled, which is multiplied by the noise value to increase the changes by larger amounts and make the terrain more intense. This allows the programmer to have more control over the terrain generation, which was specified in the research as an important consideration of procedural content generators. In addition, due to the continuous nature of Perlin noise, even if the strength of the noise is increased, the hills are still smooth and natural, and there are no big drops or changes in height. This feature allows the terrain to have a little bit more variation, and not just be the same minor variation of the same theme, which was to be avoided, as mentioned in the research. These features work together to generate terrain with different heights, shown in the figure below.

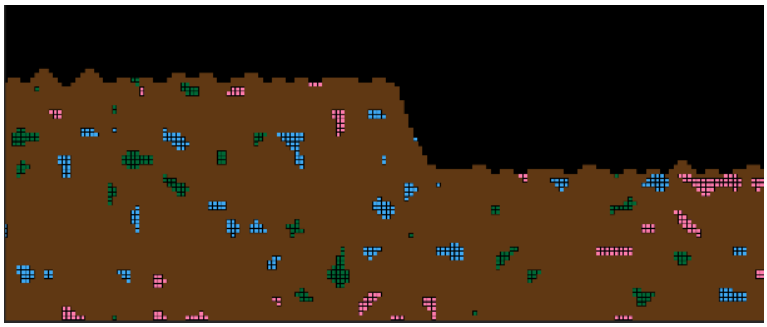

Figure 14: Image of adjusted terrain

To ensure this Perlin system works with the seed, it is done similarly to how ores are. However, as this only works in one dimension along the x-axis, it is a lot easier to configure. The y value of the Perlin noise is replaced by the seed, so for every individual seed, there is a different row of Perlin noise being sampled.

Cave Generation

Cave systems play an important role in shaping the underground landscape of procedurally generated terrain, providing exploration opportunities and influencing gameplay elements such as resource placement.

Caves generation works very similarly to ore generation. The terrain should be allocated sections that can then be cut out to be made into large cave systems. Perlin noise can be used to achieve this effect, just like it was used for the ores, however cellular automata can also be used as another type of procedural generation. Therefore, I believed it would be best to experiment with both methods to generate caves and compare the results, as both systems have advantages and disadvantages in this situation.

Perlin Noise Approach

When generated using Perlin noise, the program uses a threshold value to check whether the value in noise should be used to represent a cave. If it is over this threshold value, the tile is removed from the tilemap. Similar to ore generation, the

continuity of Perlin noise ensures that nearby positions in the grid have similar values, so large open spaces start to form and winding formations of caves are carved out of the existing terrain.

The Perlin noise can also be adapted depending on the depth so the caves can open up towards the bottom of the terrain and create large winding systems that have more potential for spawning rare loot. This system also provides the advantage that the surface isn't torn up with caves, so more items can be spawned on top of the surface when items are added. It also keeps the surface looking clean and natural.

Figure 15: Cave system generated with Perlin noise with depth adjustments

The approach with Perlin noise successfully creates caves, with more systems showing up at lower depths. However, the caves are very regular, and show up in quite predictable locations. This is due to noise giving a more uniform result than the cellular approach [22].

The seed is integrated with the Perlin noise using the same approach that was used with the ore generation, with both the x and y using the seed as an offset value.

The Cellular Automata Approach

The cellular automata system starts by generating a map with randomised coordinates filled in. This then goes through various generations with tiles being emptied or filled based on the specified criteria.

As mentioned in the research, the cells status depends on the neighbouring cells. If the cell has fewer than 2 or more than 3 live neighbouring cells, it will die, but if there are 2 or 3, then it will live in the next generation. All cells are evaluated simultaneously so they all use consistent values from the previous generation. After the specified number of generations are completed, the generated tiles are used as caves in the terrain, and are cut out.

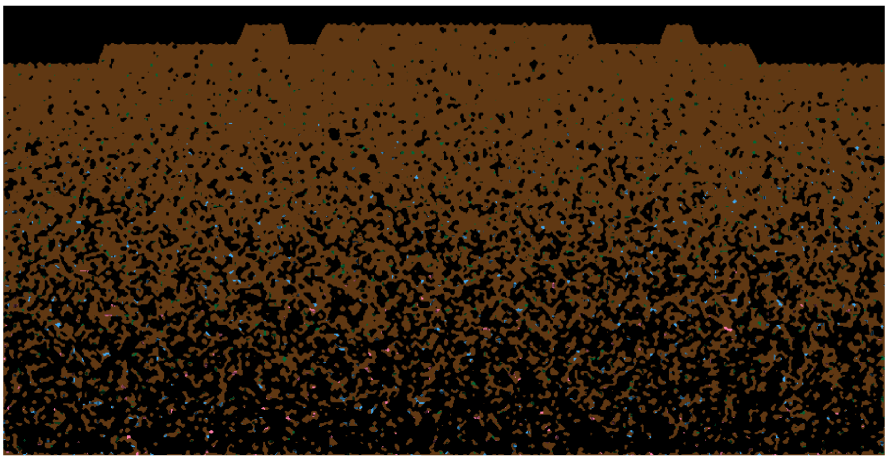

Figure 16: Cave generation with cellular automata and depth adjustments

Cellular automata is a deterministic algorithm too, meaning the same original grid will always produce the same result. By using a seed to initialise the sequence of random numbers used in Unity, the starting conditions of the grid can be reproduced. This ensures the same points on the grid are always generated, so the same caves are eventually created.

To test which approach works best in this scenario, the number of tiles allocated for caves were recorded for each y value through 1,000 generations. They are both tested with the same adapting threshold value so the results are fair.

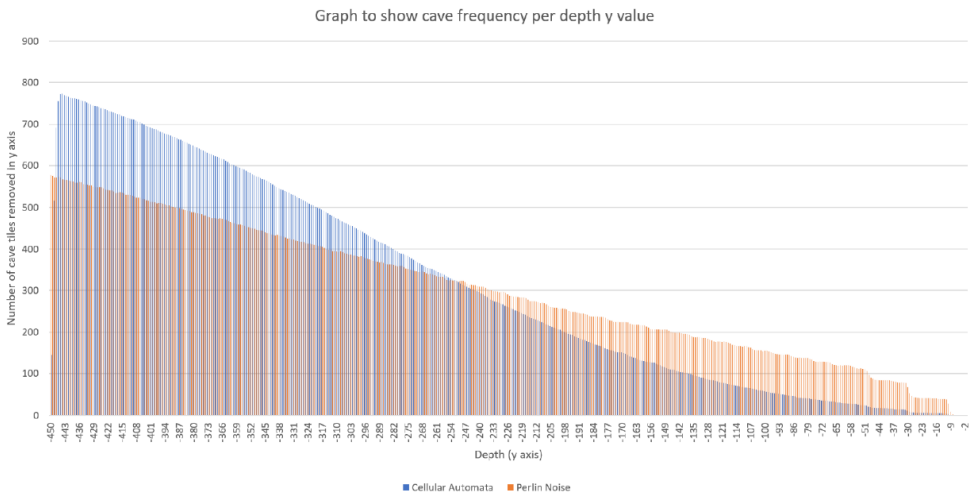

Figure 17: Graph to show cave frequency of generation methods per depth y value

Overall, after trying both approaches, it is clear that both successfully generate caves in the terrain.

The Perlin noise approach generates caves that have smooth, natural shapes. The approach is also very quick, only needing one iteration to generate and implement the

cave shapes. It is also very easy to control, due to the uniformity of the caves generated, which helps for making bigger caves deeper down and enforcing the high-risk-high reward concept. However, this uniformity makes the caves look regular and unnatural. The caves are also not interconnected at shallow depths, as they are individual shapes with relevant space in between.

The cellular automata approach sorts the problem of regularity, as the caves don't follow visible patterns due to the random grid generated at the start of the process. Caves are also generated with links and interconnected networks with how the generation stages work. This approach also yields an unintended but beneficial side effect: it naturally ensures that the edges of the world are sealed off, preventing players from exiting the intended playable area. However, due to the multiple rounds of generation, it does take longer to make the shapes of the caves. In addition, due to the random nature of the cave generation, it is slightly more difficult to control the system of caves, which is a problem as that is one of the requirements of a successful procedural generation system. The gradient of the graph shows the frequency of caves is smooth, and doesn't immediately decrease when transitioning between biomes, which shows it's more natural.

Due to the more random nature of cellular automata and the interconnected systems, I believed the cellular approach was the better approach to take.

More variation for caves

As mentioned in Minecraft, there are lots of caves that need to be generation to add a lot of variation. Even though the caves currently generated have a lot of difference in size and shape, they all follow the same patterns. In terraria, there are some surface caves that travel down like snakes and go deep into the world. Like in terraria, the expectation is that the caves don't need to be visible from the top, as the user can use tool to dig into the terrain, however these caves stand out and encourage exploration, as they are an easy means of access.

To simulate naturally winding, smooth cave systems working from the surface of the terrain down into the deeper points of the world, a new cave carving algorithm was implemented. The method iteratively moves down through the terrain, each time carving out a section of the terrain with a random radius. The algorithm uses Perlin noise to change the size of the area that is carved out, so it feels more natural. With each iteration, there is also a slight chance to move the direction of the cave, so it has a natural curvature. The length is also randomised between a minimum and maximum, between which it is terminated, adding to the randomness.

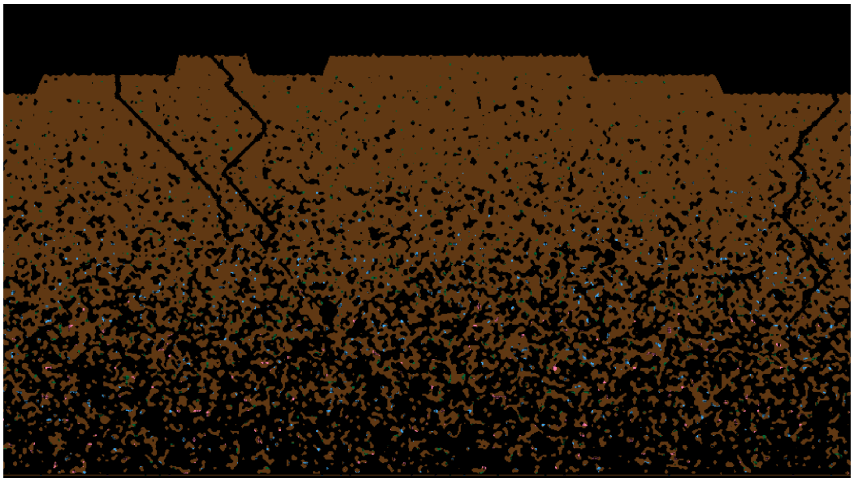

Figure 18: Terrain with 3 additional winding caves for variation

Tiling

At this stage in the generation, all the tiles are placeholders for where the blocks are going to be, there are no detailed blocks that have any orientation or surfaces, they're just monochromatic. To add life to the terrain, an adaptive tileset needs to be implemented.

When implemented, a process called auto tiling is used. Auto-tiling converts a matrix or array of information about a map and assigns the corresponding tile texture to each tile in a manner that makes sense visually for the tilemap level. This uses a tile's position, relative to its neighbour tiles to determine which tile sprite should be used [23]. This ensures the rapid creation of visually coherent environments [24].

Figure 19: 128 combinations of auto tiling using a grass tileset

Depending on which tiles are present and which aren't, the current tile is allocated a sprite from the tileset. This provides 128 potential layouts for the tiles in the 3x3 local area. To achieve these 128 combinations, the user found there would be to be 47 individual tiles designed, which were kept at a low resolution of 8x8 for simplicity. The diversity and frequency of tiles within the tileset make the environment look varied, and therefore meets one of the requirements of procedural generation set out in the research, as it does not look like minor variations of the same theme.

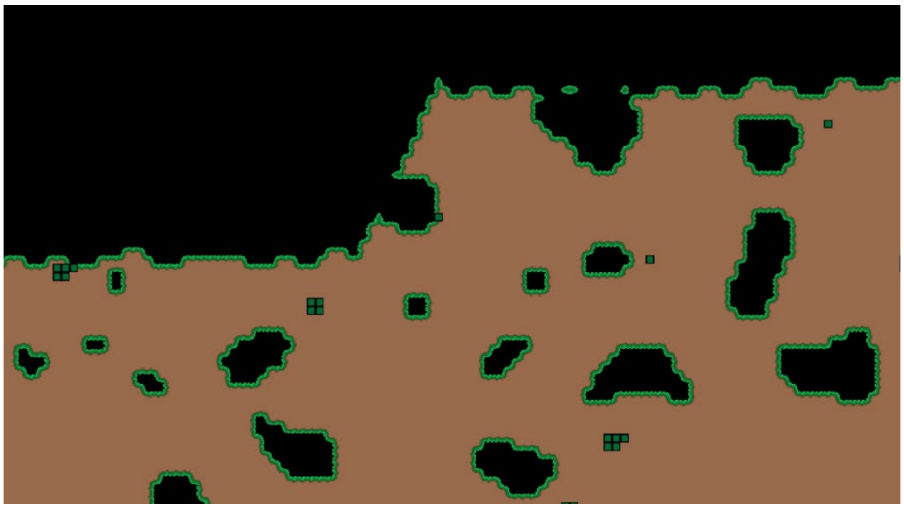

Figure 20: Terrain using grass tiles

To visually demonstrate where the different biomes lie, 2 new tilesets can be designed and added. These are just resprites of the previous tileset, but are used at different altitudes to show where one biome starts and the other ends. When auto tiling, the depth is now checked, as that is a key characteristic of the different biomes. This will then determine what tileset should be used, and successfully generates the different biomes.

To add smoothness between the tile transitions, Perlin noise is used to provide an offset for the tiles, so they do not all fall along the same y value, so the changes feel more natural.

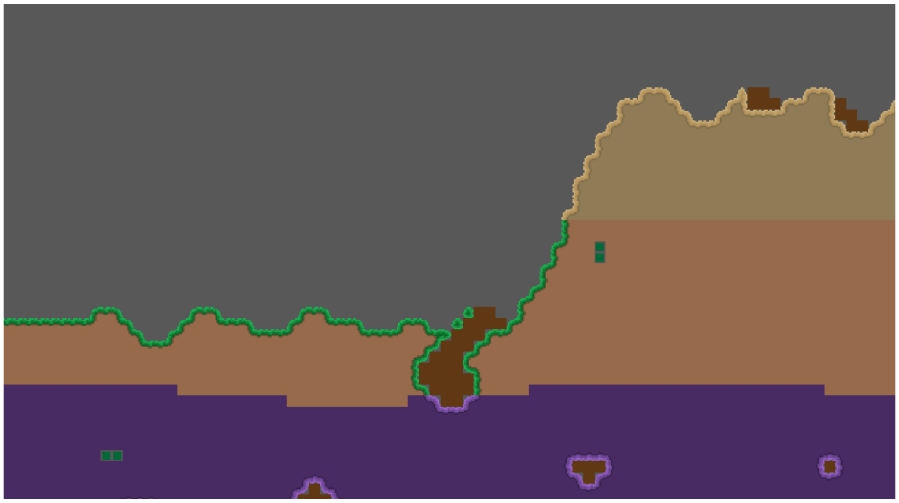

Figure 21: Terrain using biome specific tiles

Light Maps

Lighting plays a critical role in enhancing immersion in procedurally generated worlds. It brings the world to life by adding a sense of depth and dimensionality, making the current flat, basic environment feel more dynamic and layered.
The light map system simulates top down sunlight, with the light coming down onto the terrain and penetrating through the water, done through the use of shaders and textures. The light map works through a Texture2D object set to the world's dimensions, with each tile corresponding to a tile in the generated terrain, maintaining the pixelated art style. The function deals with 3 different areas: light penetrating through the water, the terrain and through caves.

Water penetration

To provide depth within the open water, the light map simulates the light travelling through the water above the terrain, with brightness gradually decreasing at greater depths. Each tile is provided with a shade of blue depending on its y coordinate, ranging from light shades on the water's surface, to dark blue deeper down. The shade for the tiles are then used to determine the shade of the terrain, due to them being at the same depth.

Terrain penetration

To ensure that the terrain is lit, a breadth-first flood fill algorithm is used to cascade light through different levels. It checks all tiles that are in contact with open water, where the light map has already filled, and takes this value, then all tiles connected to these tiles that have not been provided with a light are made slightly darker than those tiles. This loop continues until the tiles are completely opaque, making a gradient from light to dark, allowing a few of the top tiles to be visible, before making the underground hidden.

This system works similarly to surface caves too, however the amount the visibility reduces is smaller, so it simulates the light travelling and spreading out into the cave. The rest of the terrain that isn't affected by light due to it not being in view of the surface is completely opaque.

To render the generated light map onto the world, a custom shader is used, which applies the lighting effects mentioned by multiplying the light map texture over the visible terrain.

The lighting system uses a custom unlit shader to blend the generated lightmap. The shader is specifically designed to multiply the tilemap's base colour by the lightmap's colour values, darkening the terrain according to how much light reaches each tile.

The lightmap is drawn through the use of a multiply blend shader, which multiplies the destination colour by the colour of the lightmap, making areas of dark and light. The shader process is as follows:

- Vertex shader: passes through vertex positions and UV coordinates unchanged
- Fragment Shader: Samples the lightmap texture and multiplies it by a colour tint

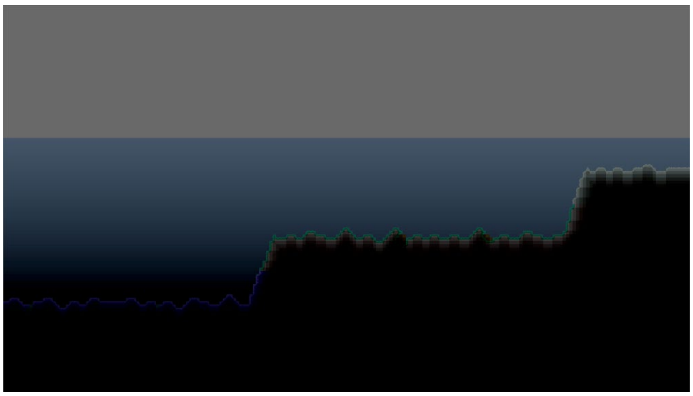

Figure 22: Light map showing light penetrating through water at different depths

Foreground / background elements

The final stage of terrain generation involves placing foreground elements—decorative or interactive objects that sit in front of the terrain tiles. These elements play a key role in enhancing the visual depth and environmental storytelling of the game world.

Foreground elements are implemented in the last stage, as if they emit light, it is easy to adjust the light map to accommodate for this, as the tiles surrounding the object are the only part that needs to change.

In contrast to the full biome generation, all the individual items are not necessary to be spawned in, so there doesn't need to be a requirement in the code to include them all in every generation.

At this stage of the generation, the program searches through the terrain to see if there are any free spots on the relevant tile. Items will spawn on all 4 sides of tiles, so it is important to check if the relevant space is free to ensure the correct item is spawned. Some items will be larger than just 1 tile, as some are extremely tall and some are wide, so these spaces need to be checked too. All objects that can fit in the available size are added to a list. An item is then randomly picked out of this list, which creates more variation in the world and makes the world feel more natural.

At deeper points in the terrain, some objects need to emit light so they are visible in the dark areas of the light map. When the objects are spawned in, any objects that are meant to emit light are given a colour and a radius, and this is used to generate the scope of the light that is produced, which is shown in the image below.

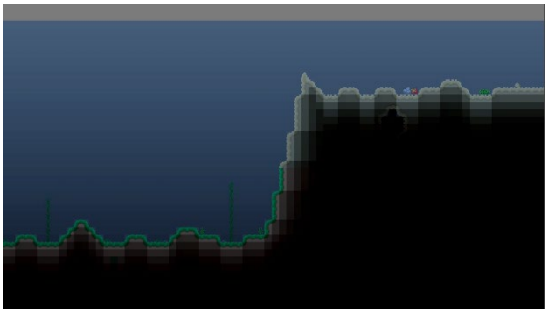

Figure 24: Biomes with their foreground items

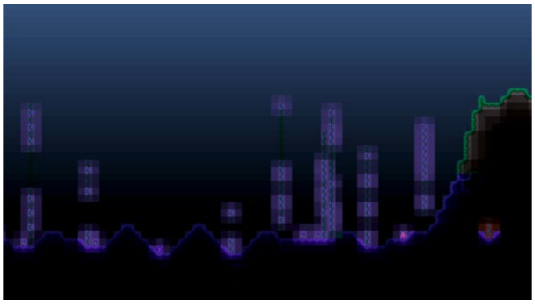

Figure 23: Biome with lit foreground objects

To ensure that the items are spawned the same for the same seed, the random functions once again follow the initialised sequence so they pick out the same random numbers every generation.

4. Testing and Analysis

Speed

As mentioned in the research, speed is one of the most important qualities of procedural content generation. Even though the generation is not in real-time, like Minecraft, it still needs to work as efficiently as possible so the user is not waiting a long time for world generation.

The speed of the generation is also important for scalability. Compared to Terraria, which has much larger worlds with more detailed generation, this project is quite small. This means that when scaled up to larger sizes, it will take much longer to generate, which is why at this point the time taken needs to be reduced as much as possible.

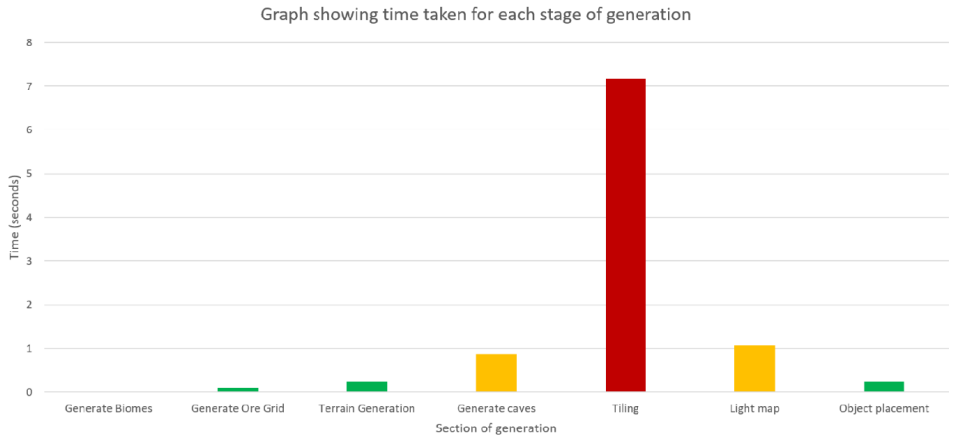

Figure 25: Graph showing the execution time for each stage of generation

To ensure that all parts of the project are efficient, their duration was tracked. The overall execution time was recorded at 9.69 seconds. The individual sections have been broken down and their times are shown in the graph below.

Bitmask-based Tiling

By analysing the graph, it is evident that the tiling stage of generation takes the longest amount of time, and therefore is the priority to deal with. The current implementation uses multiple calls to check the state of all 8 neighbouring tiles in the tileset to determine the sprite of the current tile. This is a very expensive procedure, as fetching data repeatedly from the tilemap increases cache misses and memory latency, especially when considering the size of the terrain.

To deal with this problem, the tilemap is represented in a lightweight array, with a Boolean showing whether there is a tile present or not. From these Boolean values, the program can use Bitmask Generation. For each tile, an 8-bit integer is constructed. Each bit represents the presence or absence of a neighbour in one of the 8 directions. Following this, tile classification is performed through integer comparison, rather than expensive memory lookups, causing the process to be completed quicker. This also allows the function to be scaled up to larger, more detailed maps, or more detailed tilesets.

After applying these changes and generating the world again, the overall time of generation had almost halved, going from 9.69s to 5.06s, with the tiling stage going from 7.17s to only 1.28s.

Bitmask-based lightmaps

As the generation of the light map uses a similar style to the tiling, with multiple unnecessary checks of the tilemap being carried out, it was evident this functionality could be made more efficient too.

The checks to the tilemap have once again been replaced with a 2D array of Booleans. Another array of Booleans was also implemented to replace a hash

function, which was used to track which tiles were already visited during the flood fill of caves that were not directly accessed by simulated sunlight. This change was made because hash set operations are slower than dealing with arrays.

These changes resulted in a significant performance improvement without affecting the quality or resolution of the lightmap. It also reduced the execution time of the light functionality from 1.07s to 0.33s.

Bitmask-based object placement

The object placement functionality, while not taking up too much execution time, does face the same issues with accessing the tilemap as the lights and tiles. This means that it can be made to be more efficient, saving crucial time during execution. As a result, the execution time has been reduced from 0.24s to 0.16s for object placement within the world.

Cave cellular automata investigation

The caves implemented were selected to use cellular automata for their generation. As mentioned in the implementation and research, cellular automata uses multiple generations to create the final result. The tiles in a generation are done at the same time, but looping through such a big set of tiles multiple times increases time spent. When using 5 steps, which was the original number, the processing time is 5x longer.

When the times were analysed at different generation numbers, it was found that it had little to no effect on the execution time. This can be seen as a positive, as the number of generations can be adapted to fit the needs of the game, without worrying about execution time, meaning the feature can be more controlled. Other methods were implemented to try and speed up execution time, such as multithreading and experimenting with buffers, however this had no effect on the execution time. By separating the function into sections, it was found that the cellular automata only took 8ms, whereas the actual removal of the tiles took around 800ms. This was attempted to be reduced, however due to the requirement of tile specific placement, this was the shortest time that could be used.

Overall Analysis of Optimisations

After analysing the original execution times and adapting the program to make it more efficient, the execution time of the program was greatly reduced, shown in the graph below.

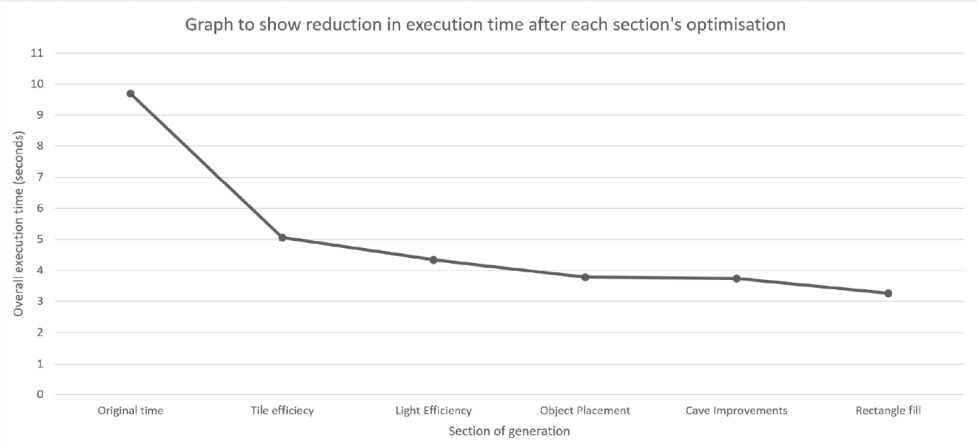

Figure 26: Graph to show reduction in execution time after each section's optimisation

By implementing the more efficient methods explained, the overall execution times reduced from an average of 9.69s to 3.26s, meaning the user had to wait a lot less time to generate their content, and also allows for better scalability for when the project expands and becomes increasingly advanced.

5. Evaluation and Conclusion

Aims and Objectives Evaluation

To check how well the program was implemented, it needs to be evaluated in accordance with the aims set out at the start of the project.

Objective 1: Generate terrain and relevant biomes

The program successfully generates 3 different biomes, and ensures they are all present in every generation due to being necessary aspects. They each have their own depth, and have smooth transitions between them. The terrain is also made more natural through the use of Perlin noise, which generates hills and dips.

Once tiling was added along with foreground items, the biomes became a lot more pronounced and prominent, each having their own distinct feel.

Objective 2: Generate Ores Properly

The ore generation works well, with Perlin noise providing the ability to create uniform, separated veins of ore. The program enables more ore to be spawned at greater depths, with rarer ore being found lower down, demonstrating the high risk, high reward feature shown in games like Terraria.

Objective 3: Generate caves

The cave generation was very successful, with investigations into different techniques to ensure the best method was being used to create the system of caves. There are larger caves at greater depths to align with the high risk, high reward principle. The

use of cellular automata also allows for the creation of large systems of caves and allow for exploration in the underground in the winding cave systems. Three winding caves were also added, to create variety in the underground and draw the player into the depths and further encourage exploration.

Objective 4: Generate light maps
The development of the light maps went far better than expected, and brought the world to life. The method of light penetrating through the water and getting darker gave a sense of realism to the body of water, and the way the terrain blocked out of light gave a sense of mystery and covered up the parts of the environment that the user isn't meant to see without exploring further.

Objective 5: Generate flora on top of terrain
The procedural generation successfully used randomness to add in biome specific foreground elements to the terrain to add more depth. Objects with different shapes and sizes were showcased to add variation and show how different objects would be spawned. These objects also worked very successfully alongside the light system, with some items producing their own light in dark areas.

Evaluation

Time Scale
I was very generous with the time allocated to each task at the start of the project, as I had a short timespan to work with. I wanted to get the main functionality of project completed as quickly as I could, while still ensuring everything functioned properly. Once completed, I could easily go back and refactor the code and perfect it. In addition, I allocated a lot of time for researching light maps and shaders, as I have struggled understanding functionality like this in the past. I managed to quickly pick up the use of shaders for the lighting and was able to implement it with no issues, therefore completing the section a lot quicker than I planned.

What was learnt
The whole process of procedural generation was new to me, so it was a very useful skill to learn. It was useful to educate myself about the important qualities and attributes of content generators too. As mentioned in the research, procedural generation is helpful for indie developers, as it allows for the creation of large, dynamic worlds without the needs to design every element by hand. This will be beneficial in future projects as it allows me to generate large world efficiently, giving me more time to focus on other areas that might require more time.

As previously mentioned, I have struggled working with shaders in the past due to a lack of understanding. However, this project allowed me the time to research them

in depth to develop a solid foundational knowledge and build on this to successfully implement a light system in Unity.

Working with cellular automata was also an interesting aspect of the project. I was introduced to this concept in a previous university module, where I had to code Conway's Game of Life. It was interesting to properly implement this feature and to put it into practise in a game scenario to see its true capabilities.

What Went Well

The project overall was a success, with certain aspects standing out as particularly strong. As mentioned, I had struggled to work with shaders in previous projects, and was unsure about how successful their implementation within my project would be. However, when I started, I grasped a good understanding of how they worked and managed to implement them into the light system, which worked extremely well.

The investigation into different types of noise was a major positive in this project too, and I believe best type of noise for procedural generation was utilised at the different stages of the program, such as Perlin noise for ores and terrain and cellular automata for cave generation.

What Could Have Been Done Better

At the beginning of the project, I didn't have the best understanding of how I could implement seeds. This meant once I was mostly done with implementation I had to go back through the different sections of generation and rework them so they were deterministic through seeds. However, this was made easier with the modular approach I used, as I could easily go back and make changes, which were self-isolated from the other functions.

Further Work

I intend to continue development on the project as there are a few areas that I want to expand on. One key improvement would involve increasing biome variety beyond the 3 initial examples. I aim to introduce underground biomes and redesign the generation system so biomes can appear both above and below the surface, which would result in greater world diversity and a more immersive environment.

I also intend to enhance the lighting system. While the current implementation is great, I intend to add different types of lights, with different colours and intensities for a bit more variation. I also intend to make the lights function in real time, with lights being added and calculated during the gameplay, rather than just when the world is generated.

Finally, I aim to add more foreground elements to the biomes to add more diversity. This includes bigger structures like buildings or ruins. These would be implemented the same way as the flora, but more checks would have to be done to ensure the space is free.

References

[1] M. Filimowicz, "Game worlds, dimensionality & time." *Narrative and New Media*. Accessed: Apr. 17, 2025. [Online]. Available: https://medium.com/narrative-and-new-media/game-worlds-dimensionality-time-d6924bae7f95

[2] D. Cole, "What is a game world?" in *Game Design & Development 2021*, A. A. Hassen et al., eCampusOntario, 2021. [Online]. Available: https://ecampusontario.pressbooks.pub/gamedesigndevelopmenttextbook/chapter/what-is-a-game-world/

[3] C. Pitt, "Hand-Crafted Content vs. Procedural Content," in *Procedural Generation in Godot*, Berkeley, CA: Apress, 2023, pp. 1–9. doi: 10.1007/978-1-4842-8795-8_1

[4] J. Togelius, E. Kastbjerg, D. Schedl, G. N. Yannakakis, "What is Procedural Content Generation? Mario on the borderline," Proceedings of the 2nd International Workshop on Procedural Content Generation in Games, pp 1-6, June 28, 2011. doi: https://doi.org/10.1145/2000919.2000922

[5] N. Shaker, J. Togelius, M. J. Nelson, Procedural Content Generation in Games. Cham, Switzerland: Springer, 2016. doi: 10.1007/978-3-319-42716-4. [Online] Available: https://link-springer-com.libproxy.ncl.ac.uk/book/10.1007/978-3-319-42716-4

[6] I. McGathey, "Search-based Procedural Content Generation in Games," UMMCSciSenior Seminar Conference, Morris, Minnesota, USA, Dec. 2014. [Online]. Available: https://www.semanticscholar.org/paper/Search-Based-Procedural-Content-Generation-in-Games-McGathey/ee77838c95e283dc8820d3e9ae002655d9dcc7d1

[7] W. Wang, The Structure of Game Design, ed. 1. Switzerland: Springer Cham, 2023. DOI: https://doi.org/10.1007/978-3-031-32202-0

[8] O. Korn, N. Lee, Game Dynamics – Best Practises in Procedural and Dynamic Game Content Creation, ed. 1. Switzerland: Springer Cham, 2017. DOI: https://doi.org/10.1007/978-3-319-53088-8

[9] S. Rabin, Game AI Pro, ed. 1. New York: A K Peters/CRC Press, 2013. DOI: https://doi.org/10.1201/b16725

[10] D. Grauer, Unity Artificial Intelligence Programming, ed. 5. Packt Publishing, 2022. Accessed: Apr. 17, 2025 [Online]. Available: https://app.knovel.com/web/view/khtml/show.v/rcid:kpUAIPE00D/cid:kt0134O6FB/viewerType:khtml

[11] J. Smed, H. Hakonen, Algorithms and networking for Computer Games, ed. 1. Chichester, UK: Wiley, 2006. DOI: 10.1002/9781119259770. Available: https://onlinelibrary-wiley-com.libproxy.ncl.ac.uk/doi/book/10.1002/9781119259770

[12] R. Davison, G. Ushaw (2024). CSC8502 Advanced Graphics for Games – Introduction to Graphics. [pdf]. Available: https://ncl.instructure.com/courses/55379/pages/tutorial-notes?module_item_id=3517648

[13] "Terraria." Terraria Wiki. Accessed: Apr. 20, 2025. [Online]. Available: https://terraria.fandom.com/wiki/Terraria

[14] “World size.” Terraria Fandom Wiki. Accessed Apr. 3, 2025. [Online]. Available: https://terraria.fandom.com/wiki/World_size

[15] “World.” Terraria Wiki. Accessed: Apr. 17, 2025. [Online]. Available: https://terraria.wiki.gg/wiki/World

[16] “Biome.” Terraria Wiki. Accessed: Apr. 17, 2025. [Online]. Available: https://terraria.wiki.gg/wiki/Biomes

[17] “Layers.” Terraria Wiki. Accessed: Apr. 17, 2025. [Online]. Available: https://terraria.wiki.gg/wiki/Layers

[18] “Home.” Minecraft Wiki. Accessed: Apr. 17 2025. [Online]. Available: https://minecraft.fandom.com/wiki/Minecraft_Wiki

[19] “Biome.” Minecraft Wiki. Accessed: Apr. 17 2025. [Online]. Available: https://minecraft.fandom.com/wiki/Biome

[20] “Cave.” Minecraft Wiki. Accessed: Apr. 17 2025. [Online]. Available: https://minecraft.fandom.com/wiki/Cave

[21] "Safe Shallows," Subnautica Wiki, Fandom. Accessed: Apr. 17, 2025 [Online]. Available: https://subnautica.fandom.com/wiki/Safe_Shallows

[22] Insane Scatterbrain, "Cave-like Maps," MapGraph Manual, Accessed: 24 Apr. 2025 [Online]. Available: https://mapgraph.insanescatterbrain.com/manual/cave-like_maps.html

[23] J. Young, "Autotiling Technique," Excalibur.js, Jul. 2, 2024. [Online]. Available: https://excaliburjs.com/blog/Autotiling%20Technique/

[24] T. Ihor, "Optimization of Procedural Generation in Godot Engine," Logos Science Conference Proceedings [Online]. Available: https://archive.logos-science.com/index.php/conference-proceedings/article/view/2621/2658