Methods for Procedural Terrain Generation

Master of Science Thesis
University of Turku
Department of Computing
Computer Science
2022
Kasimir Piispa

TURUN YLIOPISTO
Tietotekniikan laitos

Kasimir Piispa: Methods for Procedural Terrain Generation

Pro gradu -tutkielma, 61 p., 2 liites.
Tietojenkäsittelytieteet
Huhtikuu 2022

Proseduraalista generointia on hyödynnetty datan automaattisessa tuottamisessa jo pitkään. Tätä automatisoitua prosessointia on niin hyödynnetty viihdeteollisuudessa kuin tutkimustyössä, jotta ollaan voitu tuottaa nopeasti suuria määriä juuri sellaista dataa kuin tarvitaan esimerkiksi järjestelmän testauksessa.
Tässä tutkielmassa tarkastellaan erilaisia tapoja hyödyntää proseduraalista generointia erilaisten synteettisten maastojen tuottamiseksi. Aluksi tutustutaan hieman tarkemmin siihen mitä proseduraalinen generointi on, mistä se on alunperin lähtenyt ja mihin sitä on hyödynnetty. Tästä siirrytään tarkastelemaan miten kyseistä tekniikkaa hyödynnetään maastojen luomisessa ja mitä maastoilta yleensä visuaalisesti vaaditaan. Tästä siirrytään tarkastelemaan eri tapoja toteuttaa maaston generointia.
Osana tätä tutkielmaa, on valittu kolme menetelmää ja laadittu niistä kullekin oma toteutus maaston generointiin. Työssä tarkastellaan näiden toteutusten suoritustuloksia, ja mitä mieltä testiryhmä on kyseisistä synteettisistä maastoista. Saadut tulokset ja niiden analyyysi esitellään tutkielman lopussa.

Avainsanat: proseduraalinen generointi, automaattinen, maisema, maaston generointi, virtuaalinen, metodit

UNIVERSITY OF TURKU
Department of Computing

Kasimir Piispa: Methods for Procedural Terrain Generation

Master of Science Thesis, 61 p., 2 app. p.
Computer Science
April 2022

Procedural generation has been utilized in the automatic generation of data for a long time. This automated processing has been utilized in the entertainment industry as well as in research work in order to be able to quickly produce large amounts of just the kind of data needed, for example, in system testing.
In this thesis, we examine different ways to utilize procedural generation to produce different synthetic terrains. First, we will take a closer look at what procedural generation is, where it originally started, and where it was utilized. From this we move on to look at how this technology is utilized in the creation of terrains and what terrain is generally visually required. From this we move on to look at different ways to implement terrain generation.
As part of this thesis, we have selected three methods and implemented our own implementations for terrain generation. We look at the performance of these implementations, and what a test group thinks about those synthetic terrains. The results obtained from this are analyzed and presented at the end of the thesis.

Keywords: procedural generation, automatic, landscape, terrain generation, virtual, methods

Contents

1 Introduction

Procedural generation refers to the process of creating data through an automated process as opposed to doing it manually. This process is often linked to computers for creating 3D models and textures, but the creation of the models and textures is not the focal point for this thesis. Instead, we are going to focus on techniques used in procedural generation to create artificial terrains.

The goal of this thesis is to analyze different types of procedural terrain generation methods and to find out their strengths and weaknesses. This includes an in-depth comparison of a few selected generation techniques by subjecting them to a set of prepared terrain generating scenarios. The end results are evaluated using different metrics and presented separately.

Chapter 2 covers the background, giving the reader a basic understanding of what is procedural generation, what is computer generated terrain and how can it be used to generate terrains through various different methods and techniques. We introduce four different groups of generation techniques, explaining what they are and what can be done with them. Each technique group contains multiple terrain generation methods, and each one of them will be given an overview for better understanding how they function and can be utilized properly.

In Chapter 3, we will introduce the research questions for the thesis. How well do these terrain generators perform under duress for one minute, and how good do the generated terrains look like? For the first research question, we will be creating a

set of three different implementations for testing purposes. For the second question, a set of testing scenarios have been planned for the generators, and those results are then presented in a survey form to volunteering participants for evaluation. This chapter presents the testing scenarios and their explanations, along with the questions and other criteria for the evaluation task.

Chapter 4 takes a more in-depth look at the process of procedural terrain generation and introduces the three selected procedural terrain generation techniques, which will be used to solve the research questions presented in Chapter 3. The implementation processes of those techniques are briefly discussed. This chapter will also introduce some other procedural terrain generation techniques that were going to be included in the problem solving process, but were found unsuitable for the task. We will also line out the reasons for leaving them out.

Chapter 5 will showcase the primary results created through the procedural terrain generation techniques introduced in Chapter 4. We give the results a performance assessment based on the metrics gathered during the techniques' run-time. The same technological results will also go through a secondary subjective evaluation, providing graded ratings for each of the primary results. These secondary results will tell which of the generated terrain results is the most and the least appealing to group of selected evaluators. These terrains will be presented in a ranked list based on the survey results presented to the evaluators.

Chapter 6 will cover the analysis of the results, starting with the results gathered from our procedural terrain generation implementations presented in Chapter 5. After this, we proceed to analyze the gathered survey results.

Chapter 7 will contain discussion regarding the procedural terrain generation as a concept, various differences between the used techniques and what we could have done differently during our generator implementation process, and overall thoughts about the end results.

2 Procedural Generation

In computing and electronic content creation, procedural generation is a method of creating content and data through different algorithms instead of doing the process entirely manually. It primarily combines the use of assets created by a user and various algorithms that provide a certain amount of randomness, but the generation process can be automated to work without involving the user in the process. [1], [2]

Procedural generation has been used ever since the 1980s to produce content for video games and movies [2], [3]. In general, various video game developers had the problem of limited resources and to solve this problem, many of them resorted to creating algorithms that would create the content using different generation methods. These algorithms were used to create various types of different assets such as 3D models, textures, levels and non-player characters. All of this could be done by including specific instructions in the algorithms and a seed value to replicate the wanted generation outcome when needed. These algorithms used a lot less memory space on a video game cartridge or a floppy disk than saving human-made assets directly onto the device. This meant that the developer could include multiple different algorithms for various things using the same amount of memory space that it would have taken to include a single asset in the game. By combining these algorithms, the developers could create a lot more variety in assets than would have been possible otherwise. [1], [4]

This method is also known as *Procedural Content Generation* (PCG). Although

procedural generation has been used widely for a long time, it has become a research topic only during the last decade. This thesis will focus less on the various types of content that can be generated through PCG and more on the various methods used for *Procedural Terrain Generation* (PTG). Although terrain generation is a part of content generation, simply focusing on the PCG would be too large of a topic for this thesis alone to cover it properly. [2]

2.1 Procedural Terrain Generation

The main objective of procedural generation is to generate an endless amount of randomized data, which is then utilized for other purposes. In PTG, this randomized data, depending of the approach and technique, is used to create a set of desired attributes for the terrain. A generator does not always have to include a set of specific instructions on how the world should be generated, but it allows more favorable outcomes for the user when said instructions are provided for the generator.

In most cases, the data is first used to create a height map to work as a basis and afterwards other randomly generated features are added onto the height map or they change the map in some other ways. Creating a height map is the most common form of terrain representation, because of its low demand of resources during the generation process [1]. The main goal of randomization is to prevent a repeating pattern from occurring when the map is being generated further, but it is crucial to find a suitable balance between how much things are getting randomized. In video games, a game level should have enough random decision-making in the level generation to make it feel new for the player whenever a new level is created, but the randomization process should still stay within certain boundaries for the player to be able to enjoy the level and eventually solve it to move forward in the game. This topic about limiting randomization in PTG will be discussed further in Section 2.2.1.

2.2 Generation Techniques

Generation techniques are different approaches to generating virtual terrain on a computer system, be it two-dimensional or three-dimensional terrain. Some of the approaches tend to have some things in common between them when it comes to building the system for the terrain generation, but their differences on the technical side sets them apart from each other.

To describe the various types of commonly employed terrain generation methods and differentiate them efficiently, they need to be categorized by their respective key attributes. In most cases, a particular method is always selected for a specific way of approaching a problem. The problem is usually what kind of terrain needs to be generated and for what purpose. Finding suitable techniques for algorithms is a challenging process, as each of them have their own strengths and weaknesses. Fischer et al. [5] list the four most essential requirements for a PTG system:

- *Realism:* realistic looking terrain
- *Performance:* does not use too much time or computational resources to generate the terrain.
- *Usability:* fast and not too complex to use, scalable in size.
- *Flexibility:* allows multiple types of terrain biomes and features during the generation

Doran and Parberry [6] add *Novelty* and *Interest* into the list of requirements for the system. Novelty means that the generator "contains an element of randomness and unpredictability" and Interest is "a combination of randomness and structure that players find engaging" [6].

Many methods share the same strengths, but it is usually the weaknesses that set them apart when deciding which approach to use. Because most system solutions

are designed on a case-by-case basis, those solutions cannot be reused as efficiently in other systems. For this reason, modularity is challenging to implement and the choice of methods is of a great importance for terrain generation.

It is debatable whether there are three or four or even more families of techniques, as some techniques are sometimes combined together into a singular family of techniques. In some cases, the viewpoint on how these techniques should be categorized is also different. When it comes to varying viewpoints, Fischer et al. [5] tend to categorize numerous PTG algorithms into the following types: *synthetic*, *physics-based* and *example-based* approaches. Although it is an efficient way of categorizing algorithms, in this thesis we follow the categorization provided by Valencia-Rosado and Starostenko [1], where the various PTG families are categorized into *stochastic, simulation, learning and sketch* methods. This allows the methods to be approached separately and coherently without them getting mixed up in the process, as some methods tend to be combined in order to achieve the wanted result.

2.2.1 Stochastic Approach

Stochastic approach to procedural terrain generation involves methods that try to mimic the natural randomness of real world by using specifically created parameters and generation instructions. These parameters and instructions control the generation process, and because these stochastic methods perform their task fast, they can be applied recursively multiple times to increase details in the terrain. But because these methods rely on the natural randomness, they do not provide enough control over their generation, so the provided features will always be placed randomly on the terrain map. Since stochastic methods are quick to generate a terrain, they are often used to generate base maps for other methods to use as a resource for further generation. They are also most suitable for real-time generation because of their speed. [1]

Brownian motion

According to Blatz and Korn [7], Brownian motion or Brownian movement, is an important historical concept for procedural generation algorithms, as it was one of the first documented phenomenons of observable stochastic processes in nature. The discovery was originally made by the botanist Robert Brown (1773-1858), after whom the phenomenon was named. The phenomenon he observed in 1828 was caused by the irregular movement of indiscernible molecules in liquids and gasses with random velocities. These molecules would then randomly collide with pollen grains, causing them to disperse in random directions. [7], [8]

Brownian motion has been used later as a basis for different types of research. According to Karatzas and Shreve [8], Brownian motion has been used, for example, to model stock prices, thermal noise in electrical circuits and varying types of management systems. In mathematics, Brownian motion is described by the Wiener process, named after Norbert Wiener. The process describes the motion as continuous-time stochastic process, which is a mathematical model for phenomena and systems that seem to contain varying amounts of randomness in them. [9]

Fractals

Fractals are important for stochastic generation methods as they are geometric figures, where each of the parts possess the same statistical attributes as the whole. They are created through a repetition from a single base value [1]. Most often these fractals are presented in the geometric shape of a triangle or a square. They can increase or decrease in size quite efficiently depending of the amount of detail demanded by the user. Because natural looking terrain is not truly created from Euclidean shapes like previously mentioned triangle and square, they are treated as fractal in nature [10]. When combined with a midpoint displacement algorithm (also known as diamond-square algorithm), a PTG method can be utilized to cre-

ate realistic looking landscapes. As the name implies, the midpoint of a triangle gets displaced by a random value in the method to provide more randomness in the generation outcome to make it look a lot more natural [11]. Although fractal-based methods can create large-scale terrains with unlimited amount of detail, they often do not provide enough control for limiting where specific terrain features should be placed. They also lack the capability to create erosion and weathering in the terrain, which means the terrain will always look freshly created.

According to Rose and Bakaoukas [10], fractals have two important traits that make them a lot more suitable for terrain generation than regular Euclidean shapes: self-similarity and chaos. The first trait allows the fractal to divide itself into smaller copies of itself, and being chaotic in nature allows them to have infinite complexity. When combined with a computer system, fractal terrain can be generated when utilizing a *noise function*. The concept of noise will be explained later in Section 2.2.1. Noise function refers to a set of instructions and parameters for generating pseudo-random noise data for terrain generation. As previously mentioned, this can be used for creating a height map or rendered directly, depending of the user's needs. A visualized example of a generated noise height map using Perlin noise function can be seen in Figure 2.1. The height map is formed from a grid of decimal values, where the shades of black and white determine the height values between 0 and 1. Those values closer to 1 are lighter in colour and on the opposite darker shades are closer to 0. This also means that the higher value on the grid, the higher the terrain would be. Figure 2.2 presents a the colourized version the height map of Figure 2.1 to represent mountainous terrain between lakes.

To create something called *fractal noise* to provide more variety and increase complexity in generation, the developer needs to combine different sets of noise data with different frequencies to create said noise data. Although the use of noise for procedural terrain generation is popular due to how easy it is to use and how little

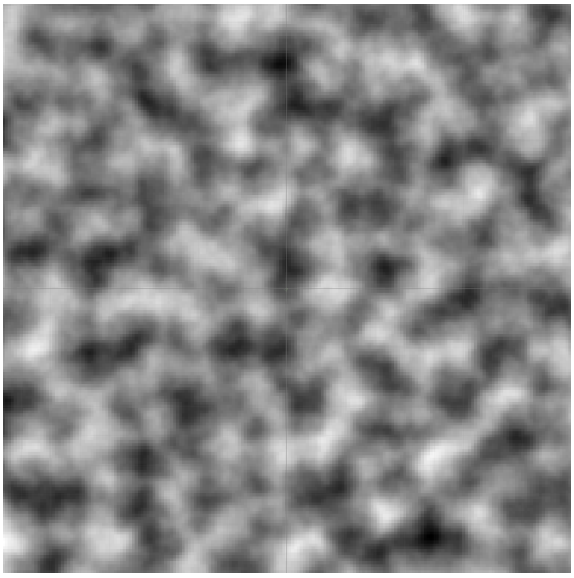

Figure 2.1: Visualized height map of generated Perlin noise

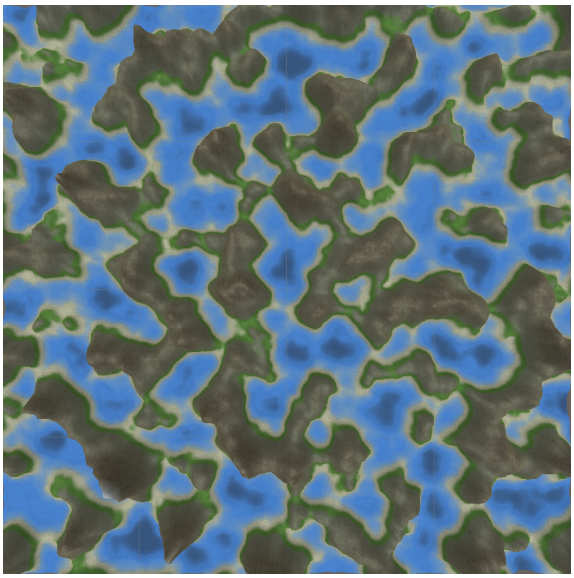

Figure 2.2: Colourized version of the noise height map of Figure 2.1

it requires computational power to perform, its lack of control and ways to adjust noise parameters make it challenging to create realistic looking terrain. [5], [12]

What is *noise* in the context of procedural terrain generation? Noise can be seen as a set of multiple sound-waves created from random numbers between a set of minimum and maximum wave height value, or simply as the height value, which are then arranged into an integer grid. This is known as value noise [4], and most often the range is between -1 and 1, or 0 and 1, depending of the method to create noise for data. As a task, it is way too complex for all noise methods to be covered in this thesis, so instead three well-known noise generation methods will be used as examples to provide sufficient context and understanding of the nature of noise methods: Perlin Noise, Simplex Noise and Worley Noise.

Perlin noise was originally a part of an image synthesizer developed by Ken Perlin in 1983. The purpose of the synthesizer was to be able to design "highly realistic Computer Generated Imagery" (CGI) [13] quickly based on the requirements. This began with experiments regarding the creation of natural looking textures, but Perlin found this process to be a time-consuming process since he had to rewrite and rebuild the code every single time when he wanted to try out a new combination of functions. This led Perlin to develop a pixel stream editing (PSE) language to reduce the amount of used time for trying different generated textures. [13]

To maximize the potential of PSE and the solid textures, Perlin developed a set of stochastic functions to provide more complex imagery and textures, which would later be come to known as Perlin noise function. This function was described as "a scalar valued function which takes a three dimensional vector as its argument". [13] Currently, it is categorized as gradient noise. To utilize the function properly, an implementation of a Perlin noise generator usually involves the following three steps in the following order [14]:

- *Creation of a vector grid:* An n-dimensional grid of random gradient vectors

- *Performing dot product:* Performing mathematical dot product operation using the gradient vectors and their respective offset values. This is done by taking two number sequences that are equal in length from the grid and returning a single value after the operation. [15]

- *Performing interpolation:* Interpolation function is then performed on all the gradient vectors and their respective dot product values on the grid.

Most often Perlin noise is implemented this way to PTG functions of two to four dimensions, but because of the noise function's flexibility, it can be implemented to work on any number of dimensions if required.

Simplex noise is an upgraded version of Perlin noise, which was also made by Ken Perlin in 2001. Simplex differs from its earlier version by having more simplified and optimized mathematical functions in the algorithm, when calculating gradient distributions for the noise map. This results in improved method efficiency and overall better visual quality for the generated noise by having more uniform distribution of gradient values. Simplex also corrects other minor errors that were present in the original version. [14], [16]

Worley noise was developed and published in 1996 by Steven Worley [17]. The noise method differs from Ken Perlin's works in a significant way. Whereas previously mentioned gradient noise methods rely on the idea of using a lattice and have the mathematical functions calculate single values for each location in its space, Worley noise functions on the idea of distributing random points in three-dimensional space. These points are used to represent various features in the space, and they are also used to partition the three-dimensional space into cells. These cells do not have fixed sizes.[17], [18]

According to Hettiga et al., "Worley noise is not a traditional noise function, but a rather a texture basis function, producing cellular-like textures similar to Voronoi cells." [19] A representation of a Voronoi cell diagram can be seen in Figure 2.3.

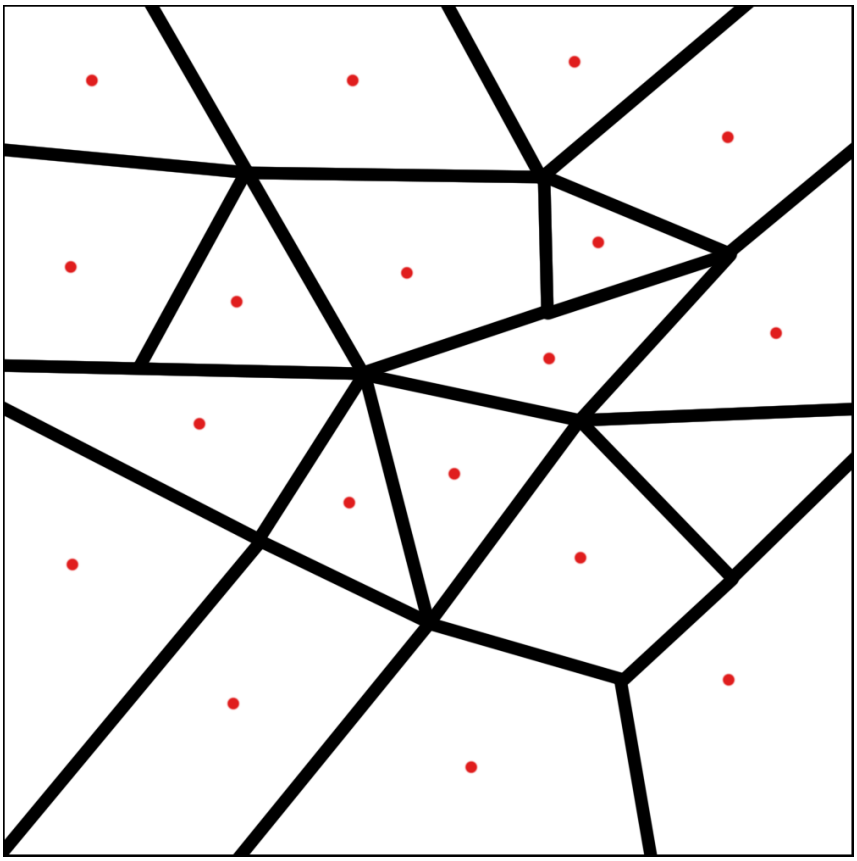

Figure 2.3: An example of a Voronoi cell diagram. Cells begin to expand from the red dots, partitioning the plane into their respective areas which are bound to their cells.

Parametrization

Parameterization is an important part of the stochastic approach to PTG, because it is used to allow the user to influence the appearance and outcome of the generated terrains, although the control is more akin to general instructions than directly controlling feature placements. Depending of the amount and scale of parameters introduced to the user in a PTG, the overall amount of control can vary a lot between different PTGs. Often these parameters are used to limit e.g. the amount and size of different terrain biomes, or ensure that specific features will always be present in the generated terrain. Depending of the PTG's implementation, these instructive parameters can be applied recursively multiple times to increase the amount of detail present in the terrain, as was mentioned earlier in Section 2.2.1. [1], [20], [21]

Tiling method

According to Valencia-Rosado and Starostenko, "tiling is a technique that improves terrains by dividing them into smaller areas. A different set of parameters can be used at each cell, in this way the monotony of noise algorithms is broken and transition between areas look more natural." [1]

Because tiling is used on already generated terrains, it needs to be combined with other PTG methods to function efficiently. An example of a PTG function that essentially utilizes tiling in its operation is the previously mentioned Worley noise in Section 2.2.1. This means that Voronoi diagrams can be viewed as a visual representation of the use of tiling in action. [22]

Grammar-based generation

Grammars are a set of production rules that a PTG will follow when generating a terrain. Each rule transforms a feature in the generated terrain into one or more other features. As an example, a PTG has a rule that at least one terrain biome

needs to be generated, so it generates a plain prairie biome. In a separate rule that applies later in the PTG process, the biome gets added a house into it as an additional feature. Grammars are a part of formal languages, that are used to set the production rules for the generation process. [1], [23]

Marák et al [23] uses a Lindenmayer system (L-system) to propose a way to model terrain erosion. It consists of a rewriting system and array grammars to create the set of production rules for the erosion process. The grammar is formed from an alphabet of symbols, where each symbol represents a production rule, which then are set in strings to arrange the order of required rules. These strings were then set in a specific order to get the wanted erosion process. [23]

2.2.2 Sketch Methods

Sketching is understood to be the process of creating a freehand drawing rather quickly. Provided sketches are then used as a basis for a more professional looking end product once they have been processed properly. In the context of PTG systems, sketch methods take advantage of these handmade sketches when generating the terrain. Such methods offer a better control over the generated terrain, as the user has the possibility to place any terrain feature around the sketch map. These sketch maps will then be rendered as requested based on the set parameters, although due to increased control for the user, the end result has less realism in terrain when compared to other PTG methods. [21], [24], [25]

2D Sketch

As the name implies, this PTG approach revolves around the use of two-dimensional sketches in terrain generation. To better explain how 2D sketching works, let us examine a sketch method proposed by Yin et al. [21]. This two-dimensional sketching method generates the terrain based on a colour-coded handmade sketch, which con-

tains multiple different pen strokes, the user would draw for the algorithm. All the required terrain features would be represented by separate colours on the sketch before it would be scanned for the computer to be processed further.

Yin et al. aim at providing a sketching method for terrain generation, which would have low skill requirements, so it could be used efficiently by anyone that does not possess any professional understanding about 3D modeling or related skills. This approach to PTG via sketching allows precise placement of terrain features onto the terrain as briefly mentioned earlier in Section 2.2.2.

The proposed method's generation process has been parameterized, which can be adjusted by the user if necessary. The process has been split into three steps [21]:

1. *Initialization:* The first step involves reading and scanning the user's sketch, determining the feature placements and respective feature classifications. Each pixel in the scanned image is processed and RGB values gets compared to the list of predetermined values that define which range of values represent which terrain feature.

2. *Generate terrain skeleton:* The second step generates a height map based on the terrain features and their respective control parameters.

3. *Extension:* The third step involves the visual improvement of terrain features using the corresponding extension parameters for each feature.

This approach allows the user to generate a suitable terrain for their needs, but as Yin et al. [21] mention in their conclusions, their approach requires more refining as the method has slight problem when it comes to generating lakes and rivers into the terrain along with canyons, terrain features that extend downwards into terrain surface. They also proposed doing more research into automating their sketch map creation process and skipping the part where the user has to draw the

sketch themselves. Other types of 2D Sketch methods can be found in other works such as [25] and [26].

3D Sketch

According to Valencia-Rosado et al. [1], three-dimensional sketching differs from two-dimensional sketching in a way that instead of using 2D lines, the developers could use 3D vectors from the start in their PTG process when using a sketch as a basis for terrain. This is not necessarily the only way to do 3D sketching, as can be seen in [27], but we are going to look at the 3D vector approach as an example of how it can be done.

Becher et al. [28] propose in their solution to use terrain height map tools for creating 3D volumetric terrain. Becher et al. mention being inspired by *Feature Curves* (parametric 3D spline curves), which are used for modeling terrain and constraint placement in the terrain view.

Because normal 2D height maps alone cannot be used to create overhanging terrain features or underground structures without additional tools present in the PTG solution, Becher et al. take the 3D approach to the problem since such limitations do not apply to three-dimensional environment when it comes to representation of generated terrain. By placing descriptive constraints around the terrain scene, the surrounding terrain surface can be propagated when required. These descriptive constraints are called *primitives*. Using the previously mentioned Feature Curves as the primitives to describe and parameterize required terrain features for 3D space, Becher et al. were able to present terrain features that would not be as easily done in 2D space. [28]

Figure 2.4 illustrates the workflow created for the GPU computational pipeline for Becher et al. terrain solution. The alphabetical letters N, R and B represent different vector fields with varying vectors and other parametric information. The

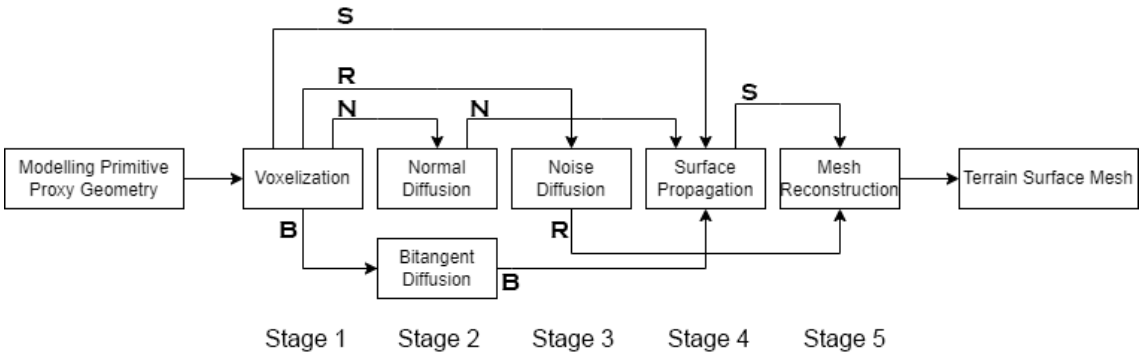

Figure 2.4: GPU computational pipeline for 3D sketching method proposed by Becher et al. [28]

letter S represents a scalar field for storing terrain surface data. These fields will then go through the pipeline stages in order to create the surface mesh. [28] We briefly explain what each stage does in the Figure 2.4 pipeline:

- *Stage 1:* In this stage, the primitives responsible for input modelling are converted suitable for the computational domain.
- *Stage 2:* This stage takes care of diffusion-computing normal vector fields. If there is a bi-tangent vector field present, that will also be processed as well.
- *Stage 3:* This stage is used to calculate noise parameters with a diffusion algorithm.
- *Stage 4:* In this stage, the values of scalar field S go through propagation along with other Stage 1's data.
- *Stage 5:* This stage combines all the processed surface field data into the final terrain mesh.

This approach of using Feature Curves allows the user to create 3D volumetric terrain with vertical features, which can be quickly rendered and edited when

needed. Although Becher et al. point out that their approach could be improved by introducing additional constraint properties for additional surface features, a larger variety of modelling primitives or other types of noise to make the generation process more complex, thus more suitable for terrain models that focus on geological accuracy. [28]

2.2.3 Learning Methods

Learning methods are a category of PTG methods that focus on the idea of teaching the methods how to generate terrain in specific ways by giving them data of real-world to be learned before trying to imitate the look of real terrains. This learning data is usually given in the form of image files and digital models of various terrain types. [1], [29]

Inverse procedural generation

As a PTG method, inverse procedural generation (IPG) takes advantage of the idea of using parameters by learning how to estimate object functionality based on given user-input. In this approach, the method learns those parameters from examples given by the user, for example, pictures of trees. Emilien et al. [29] explain the idea of IPG methods as follows: "inverse procedural methods aim at easing the use of procedural models by automatically inferring input parameters from user-defined output or constraints."

In this context, object functionality refers to how an object acts in the given context. Using the previously mentioned example of using pictures of trees as data, the method would learn of tree objects and how the trees are placed around in the pictures. From this data, the method would make a rough imitation of how trees would be placed around in the generated terrain.

Finding and calculating the near-optimal parameters from the data for the

method is not without problems. Yeh et al. [30] utilize a Markov Chain Monte Carlo (MCMC) solution to calculate probability from the sample data for object distributions around the generated terrain scene.

Example-based learning

In this approach to PTG, example-based learning methods utilize the extraction of terrain features from different sets of example data. These data sets could, for example, be digitized versions of real-world height maps which are turned into elevation models for the method to be utilized later in the PTG process. [31], [32]

This type of method can be combined with other methods to achieve more complex terrain generation. Zhou et al. [31] propose an example-based PTG system which utilizes sketched feature maps made by the user as a basis for the new terrain. They extract terrain features with extreme height differences (mountains and canyons). These extracted terrain features are applied on top of the feature map with matching markings, which were later connected together with the use of splines and other terrain warping and blending techniques to finish the terrain.

Another use-case is presented by Génevaux et al. [32], who create new terrains by combining together extracted terrain features and computer-generated features. The overall generation process involves multiple small-scale feature combination processes, which will be combined together in the end to form the final terrain.

Search-based learning

According to Valencia-Rosado et al. [1] when talking about search-based methods, "most works are based on evolutionary algorithms (EAs), which rely on a set of examples that are evaluated using a fitness function." These *fitness functions* are automated functions that are used to test and give grading to generated terrains based on the given parameters or other criteria [33]. These functions constantly

generate and test new terrain, comparing the new iteration's grading to the old one's grading, always trying to improve the grading until the set criteria are met.

Walsh and Gade [34] explain about the fitness evaluation process the following way: "The fitness evaluation is a key aspect of the search heuristic and is commonly based on an objective measure within the domain of interest. - Fitness is evaluated by the designer by selecting one or two desired terrains. If two individuals are selected then crossover and mutation are performed by selecting two individuals as parents. If one individual is selected then they introduce a mutation operator for generating new individuals. This process is continued until designer is satisfied with generated terrains."

To efficiently setup a search-based method, the user needs to create a search space with the appropriate terrain parameters. These will be used by the fitness function to provide a grading for the generated terrain. The evaluation process does not always have to be fully automated, as it is also possible to create a search-based PTG method that allows the user to guide the evaluation process by selecting the favorable terrains as they are being generated. This directs the function towards more favorable outcomes. Although user-guidance can be useful, if the method is not capable of automating the fitness evaluation process for fully textured landscapes. [34]

2.2.4 Simulation

In PTG, terrain simulation methods aim at recreating natural phenomena that shape the terrains to create new synthetic terrains. These synthetic terrains can be, for example, used to predict how real life environment can change and develop over time, although this approach is mainly used by ecologists only. [35]

Geomorphological simulation

With geomorphological simulation, the primary objective is to create a simulation that utilizes natural erosion process. The erosion can be caused by multiple different factors. Most often the erosion process utilizes at least one or more natural agents; running water being one of the most common agents. Water can cause natural erosion of terrain mass along with deposition and deformation of land materials. Figure 2.5 illustrates simulated rain water eroding and exposing what lies beneath terrain layers.

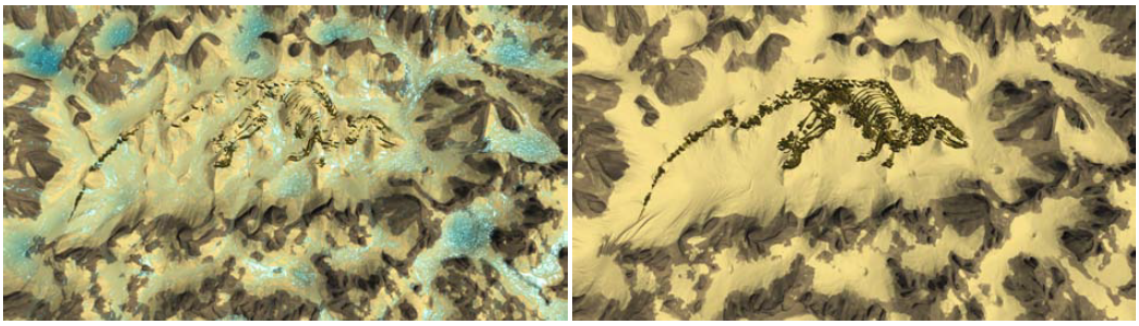

Figure 2.5: Erosion simulation revealing a fossil, created by Šťava et al. [36]

Just by having water used in the simulation process is enough to create basic erosion, but better simulations include a larger roster of natural agents to create more realistic end results: ice, glaciers, wind, rain and water pockets beneath earth, just to name a few. Almost any real life natural phenomena can be utilized in erosion simulation, but because there are so many of these phenomena, simulating all of them is virtually impossible. [36], [37]

Geomorphological simulation using erosion can be done in multiple ways. One option is to use a 3D height map as the initial landsurface, which contains multiple nodes of different material (rock, sand, and dirt) with varying parameters of resistance to different natural agents. These nodes are spread around on different layers. This acts as the 3D geological model, which will be put through an iterative

geological simulation process which applies the erosion agents onto the 3D model. Each iteration creates a new landsurface which will be then which will be exposed to the agents. This approach also allows the user to utilize Digital Mapping Agency (DMA) files as initial landsurface instead of creating their own as the starting point. [37]

Another way to do geomorphological simulation is to simulate tectonic plate activity, as suggested by Cordonnier et al. [38] in their proposal to erosion simulation. This adds tectonic uplift into the geological simulation process, lifting certain terrain features higher during each iteration and thus affecting how certain natural agents perform during the simulation.

Ecosystem simulation

Ecosystem simulation is an approach to PTG process, which is used to create new terrain or alter and improve already existing terrains by introducing climate and vegetation into the simulation process. Just like the tiling method mentioned in Section 2.2.1, ecosystem methods can be paired together with other methods to improve their functionality, but can also function as their own independent simulation engines. [35], [39]

Cordonnier et al. [39] propose combining ecosystem simulation with geomorphological simulation in order to create a system to simulate "the spatio-temporal evolution of landscapes, in other words, how a given terrain would look after some period of time" by introducing vegetation as a natural agent in the erosion process. By including vegetation into the simulation process, vegetation can limit the effects and speed of other geological processes by interacting with them, but in turn these processes will also affect the vegetation present in the simulation. This makes the simulation process more complicated, but the end result would be more realistic.

Cordonnier et al. also include a life and death cycle for the vegetation, along with

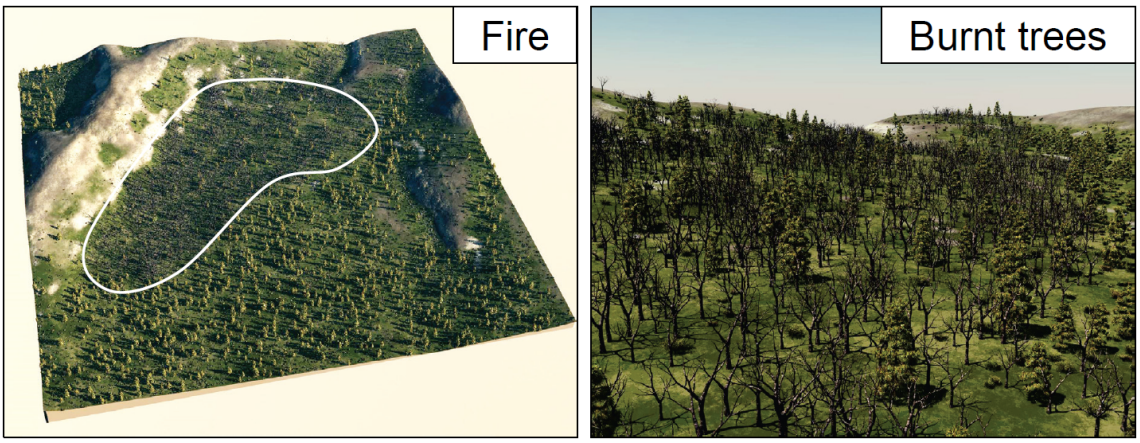

Figure 2.6: A forest fire in simulation proposed by Cordonnier et al. [39]

other complicated natural agents such as lightning and fire, of which an example can be seen in Figure 2.6. They also mention the difficulties of simulating the spatio-temporal evolution, as many geological and ecological agents have varying speeds at which they start affecting the terrain. Some agents act faster than others. This makes it a realistic problem, that if the simulation is not being run for a long enough time, some agents results might not ever be seen despite them being in the simulation process. [39]

Agent-based simulation

Although the term 'agent' in both simulation types, geomorphological and ecosystem simulation, the term 'agent' is slightly different in the context of agent-based simulation systems. Russell and Norvig [40] clarify that agents are autonomous subjects that exist in a system where they can create causes and respective effects. While Doran and Parberry [6] state there is no single agreement on what a software agent is, they describe their list of software agents for their PTG method based on Russell and Norvig's definitions: "our agents are accessible, are deterministic in the results of their effectors, are nonepisodic, work in a dynamic environment, and work

in a discrete environment." [6]

Doran and Parberry [6] in their proposal for using software agents in controlled PTG method, give us proper definition for their groups of agents used in their simulation method:

- *Coastline agents:* These agents are responsible for creating the initial landmass in the middle of a body of water.
- *Landform group:* This is a group of varying agents are responsible for creation of plains by smoothing terrain, flattening coastal areas in order to have beaches and raising terrain features to include mountainous terrain.
- *Erosion agents:* These agents are used to erode parts of the terrain to have rivers running around.

With this gathered information, we can make a deduction that agent simulation is about simulating multiple simultaneous agent operations while these agents can affect and change other agents results.

2.3 Relevant Techniques

The PTG techniques previously mentioned in Section 2.2.1 are especially important for our research, as they are going to be utilized in our practical PTG generator implementations. We are primarily going to utilize stochastic methods. The sole exception being the grammar-based approach which will not be implemented into a PTG generator. The other methods will be combined between each other to synthesize more detailed terrains and have some level of control over our terrain generation process. This means all our different implementations are going to be parameterized and include noise generators based on Perlin noise.

These implementations are discussed more in better detail in Chapter 4, where we go through each implementation separately to understand what they contain and how they function. We briefly also talk about a few simulation methods we tried to create, and what problems we encountered with them.

3 Research Question

In this thesis, three procedural terrain generation methods have been implemented into proper terrain generators. Each of these terrain generators will go through a set of four planned scenarios in order to see how well they perform under duress and what will their outcome look like. Chapter 4 will cover the selected generation methods in detail and explains those methods that did not get involved into this measurement task.

3.1 Scenarios

In this section are listed and described all the four scenarios that each procedural terrain generation method will be attempting to generate successfully.

Scenario 1 - Flat

The first scenario for the procedural terrain generation is *a flat.* It is a basic scenario for testing how well the method can produce terrain with a minimal amount of height difference. To make the terrain a bit more challenging for research purposes, the scenario will include the generation of trees around the terrain. It is expected for all the generation methods to be able to create the easiest of landscapes.

Scenario 2 - Woodland

The second scenario for the procedural terrain generation is *a woodland with lakes.* This scenario will be more challenging for the generation methods, as now they have more features required to be created successfully in order to have an acceptable outcome. The height difference value is also increased to provide a lot more visually interesting landscape, and to see if it impacts the generation process. If a single feature is missing from the requirements (basic terrain, trees and lakes), the result will be considered as a failure since it could not deliver all three required components. The result will be shown and discussed regardless of the scenario's outcome.

Scenario 3 - Mountains

The third scenario for the procedural terrain generation is *mountainous terrain.* This scenario's purpose is to test if the large height variance has any impact on terrain generation on hardware performance while using the selected methods. Scenario 3 is also expecting the terrain generators to provide the same three components presented in of description Scenario 2 and will also be considered as a failure if any of the requirements is missing.

Scenario 4 - Island and Ocean

The fourth and final scenario for the procedural terrain generation is *island and ocean terrain.* This scenario has slightly different requirements compared to the previous scenarios. In this scenario, each of the terrain generators are only required to create an ocean that is approximately on same height level and at least a single island in the ocean to be considered successful. The inclusion of trees and other additional features on the terrain are not required to pass this scenario.

Other types of terrain and landscapes were considered to be included and tested in the list of scenarios. But because of some scenarios being too similar to those

that were already chosen to be tested, they were dropped from the list due to time constraints.

3.2 Hardware Performance

This section will cover the first research question for the thesis:

RQ1: How well do these selected terrain generators perform when under duress for sixty seconds?

Duress refers to the process of continuously generating new terrain. Originally, the intention was how quickly these implementations can produce terrain, but it had to be changed due to encountered challenges while developing the implementations. All three terrain generators have been implemented in Unity game engine and run on the same hardware to provide a stable and equal testing ground. The test machine has an Intel i9 7900X processor at 3.30Ghz, 32 gigabytes of DDR4 RAM and NVIDIA GeForce RTX 3080 GPU. To gather the necessary hardware performance data, Unity's own Profile Analyzer module will be used for data handling and storing for later data management. The goal of this performance check is to see which of the methods performed the best for each of the scenarios or if they encountered issues during the generation process.

3.3 Evaluation

This section will cover the second research question for the thesis:

RQ2: How good do the generated terrains look like under the grading of a group of evaluators?

Each evaluator will be presented a form with a set of questions for grading each

of the terrain generation methods based on the results created during the scenario generation attempts. The form includes the following questions:

- **Question 1** How do different terrains look like on a first glance?
- **Question 2** What ranking would you put these terrains in?
- **Question 3** Which method and its images were of general interest to you? What about the least interesting?

On the first question, participants are tasked to rate each generated terrain's visuals from bad to very good with three other options being in the middle: mediocre, neutral and good. The second question requires the terrains to be ranked from the best to worst using numbers between one to twelve. One being the best and twelve being the worst. In the final question, participants are asked to mark down the method they found the most interesting and the least interesting. All the data will be collected into separate charts and processed in a numerical form to make the analyzing process and result presentation more comprehensible. The evaluation forms can be found in appendices A and B, where A is the Finnish version and B is the English version.

4 Generation Methods

This chapter covers the terrain generation methods and their respective implementations that were selected for the research task set in Chapter 3. It will also cover some of the generation methods that were not included in the research process, giving the reasoning why they were found to be unsuitable for the tasks.

4.1 Noise-based Method

The PTG method selected for the first implementation is Ken Perlin's noise function. Other noise functions were considered, but Perlin's was the most suited for our task at making a noise-based terrain generator. It is also a well-known function, so a certain set of expectations can be put on it such as low resource requirements and high speed for terrain generation.

In the implementation, the noise function is used to make a two-dimensional grid of gradient vectors to act as an overall height map for the entire terrain. For each vector, we assign an offset value, and we count between these respective pairs the dot product and we then interpolate this outcome. The grid is then used as a height map and its values are directly directed to a singular mesh, which then will be rendered as a three-dimensional map.

To make the terrain look more interesting than just be in greyscale, a visualizer will colourize the terrain with colours and textures, which are tied to specific height value ranges. The end result is a blend of selected colours and textures to make

the terrain look a lot more natural. To amplify this natural feeling, the generation process also spreads 3D trees with specific limitations around the terrain map. The natural feeling of environment is important during the visual evaluation process. These value ranges, textures and colours can be adjusted if need be, but for the purpose of gathering data for the second research question, these values will stay unmodified throughout the performance measurements where the pictures will be taken for the later evaluation.

In order to get performance data for the first research question, the PTG implementation contains an option for automatic terrain update and offset value changing, which technically allows us to browse the terrain on both X and Y axis for an unlimited amount.

4.2 Voxel Method

The title of the second PTG implementation is a bit misleading, as it still utilizes the same noise function to create a height map, but the terrain is rendered via the use of voxels (volumetric pixels). We considered Worley Noise function for this task, but we found out that the visual outcome of the function was not suitable for terrain generation, and instead we selected Ken Perlin's Simplex noise function to be the main stochastic function for the task. Due to a problem present in the development phase which could not be solved easily and other constraints, we had to resort to using the Perlin noise function instead. We still believed the voxel approach to visualizing the terrain would make enough of a difference to have some scientific value for this thesis.

The terrain generation has been split into multiple parts, and it revolves around the use of terrain biomes. Biomes are split into separate chunks, that have a fixed width and height size values to work with Unity's limitations when it comes to the amount of vertices used in a singular mesh. Each biome has specific set of

parameters and constraints related to its creation, and instead of using a singular height map for the entire terrain, each type of biome has its own height maps generated for it. These various biome height maps are then combined to produce the final outcome. Essentially this use of multiple layered noise maps makes this a fractal noise implementation.

Each biome type also has a fixed set of objects, like trees and rock formations, that can spawn within the specific biome. These objects are also generated from scratch, so they will have differences in their look as they are created and placed. The visualizer is also different, as it only takes care of the voxel cube placements and does not colour the terrain based on height values. Instead, each biome has specific texture values for the voxel types present in it. This allows larger visual differences in the terrain. And just like in the first implementation, this implementation too has the possibility for endless terrain viewing during program's run-time.

4.3 2D Tiling Method

The final method implementation is a two-dimensional parameterized tiling procedural terrain generator utilizing fractal noise. This implementation also utilizes the idea of using terrain biomes, but instead of assigning noise maps for height to each biome type, the implementation uses three other noise maps to create constraints for the biomes that also have their separate sets of favourable parameters. For creating these limitations, the generator creates separate noise maps with different seed values to represent general height map, a heat map and a humidity map. All these maps have user-assigned limiting parameters, which limits the biome creation for the terrain.

To visualize the outcome, each biome has an array of textures which are selected and placed randomly in the biome. The terrain map has a fixed width and height size, which can be adjusted outside the program's run-time. For the testing purposes,

the size has been set to 50 by 50 tiles. Like both other implementations, this too has the possibility for endless terrain viewing during its run-time.

To make it easier to understand what and how each PTG implementation does their respective generation of synthetic terrain, a list of functions and features can be seen in Figure 4.1.

Implementation:	Noise-based	Voxel	2D Tiling
Visuals	Gradient colouring	Volumetric pixels	Texture arrays
Noise maps	Single	One per biome type	Three
Noise function	Perlin noise	Perlin noise	Fractal noise
Parameterized	Yes	Yes	Yes
Other features	Automatic terrain update, 3D terrain elements, multiple interconnected terrain meshes	Automatic terrain update, voxel terrain elements, terrain chunk library for load management, biome library	Half-automatic terrain update, fixed map size, biome library

Figure 4.1: An arranged feature list for the PTG implementations

4.4 Non-included Methods

Multiple simulation type PTG methods were considered to be implemented for testing, but due to their respective limitations and implementation difficulties, they were ommited.

Hydraulic erosion simulation

Erosion simulation type of PTG was considered to be implemented, because they can be used to create very realistic looking terrain due to being a simulation of actual erosion taking place on a generated terrain. It would have also been interesting from the perspective of performance to see the differences between simulation and other

types of PTG methods. The problems arose when attempting to perform simulation calculations for the water droplets that would have been spread randomly around the terrain. The required programming skill and understanding of mathematics for required calculations turned out to be too difficult to be solved in a short time.

Another problem with erosion simulation PTGs is that they are tied to their respective use-cases. They are not in most cases suitable for multiple types of terrains, which makes it pretty difficult to test a single simulation engine with different scenarios.

Agent-based simulation

Agent-based simulation was considered for testing, because it did not have the exact same limitations as the previously mentioned erosion simulation. It would have been based on the work of Doran and Parberry [6], but due to time constraints and our limited technical understanding regarding the creation of a simulation engine, the development had to be put on hold in favor of a simpler terrain generator.

5 Results

This chapter covers the testing and evaluation results for the research questions presented in Chapter 3. First starting by showing what kind of terrains were produced with the criteria for each scenario and then proceeding to the results for the hardware performance testing in Section 5.2. This gives us understanding on how much random access memory the generation process requires during the one minute test run for each scenario. In Section 5.3, we cover the results of the survey for the scenarios visuals by presenting scores for each scenario and then arranging them in the order from the most favourable to the least favourable. The result for the most visually interesting technique is presented.

5.1 Generated Terrains

For the first test scenario using the first implementation, the generator was able to produce a fairly flat terrain surface for the entire visible area in Figure 5.1. Trees have been spread around quite randomly. There seems to have been some type of error while placing the trees around the terrain, as some of the them are clipping into the ground. With the requirements presented in Section 3.1 and comparing them to the outcome, this test can be considered as a success.

For the second test scenario with the first implementation, the generator was able to produce a terrain surface with greater height variance with some lakes and ponds spread around the terrain, as can be seen in Figure 5.2. The end results

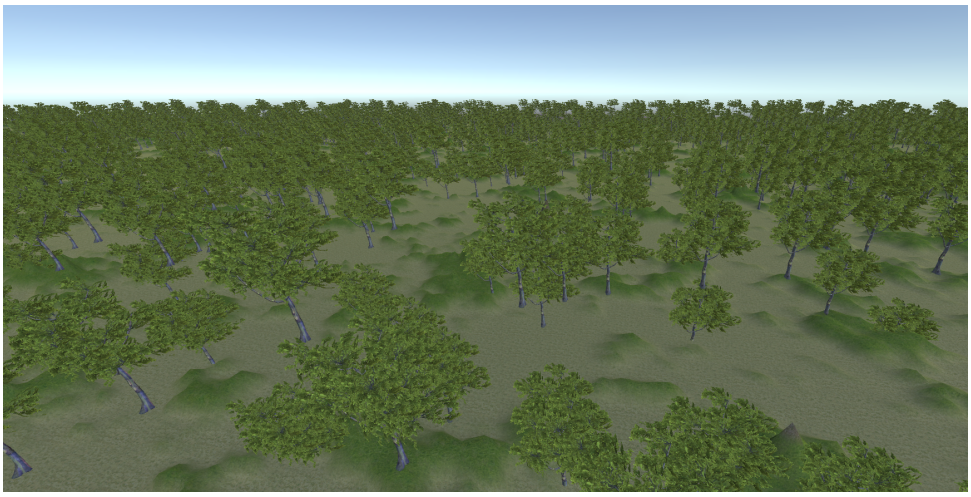

Figure 5.1: Method 1 Scenario 1 - Flat

for the test meets all the requirements presented in Section 3.1, hence it can be considered as a successful test. Although it cannot be directly explained why some of the hill tops have the snow texture applied to them, as they are not tall enough to meet the height requirements to be mountains either, it can be assumed that the might have been faulty parameters set before the generation process began.

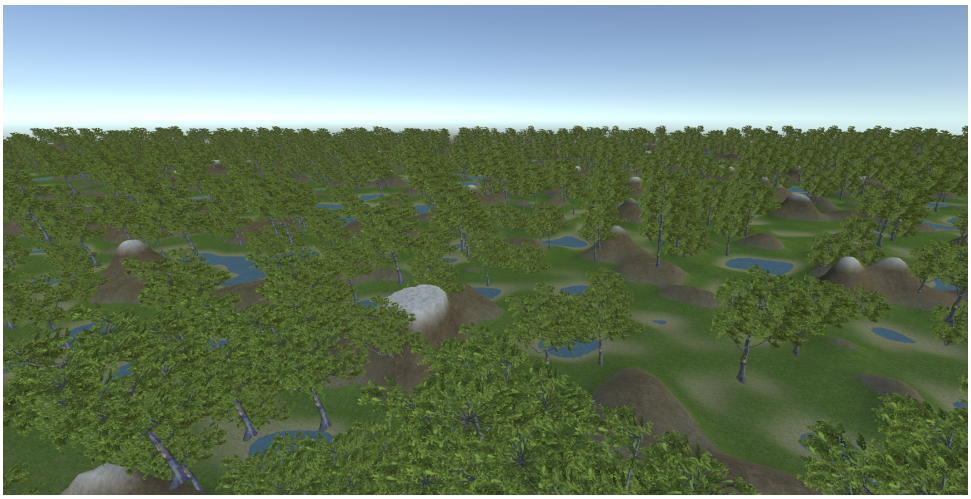

Figure 5.2: Method 1 Scenario 2 - Woodland

For the third test scenario, the generator was able to produce a clear set of mountains as can be seen in Figure 5.3. As it meets the set requirements presented

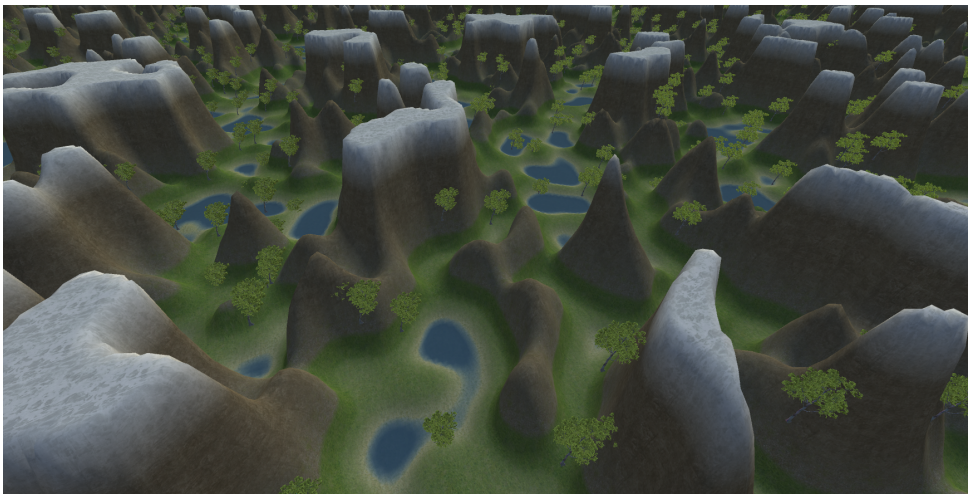

Figure 5.3: Method 1 Scenario 3 - Mountains

in Section 3.1, it can be considered as a successful test for generating mountainous terrain. The flat tops of the mountains can be explained by the generator meeting the maximum height value and then cutting it off as flat, since the maximum height value is 1, starting off from 0. While the overall visuals for the terrain look decent, the scale difference between the terrain and randomly placed tree objects make the mountains look fairly small in comparison. The scale for trees should have been set smaller for a better looking outcome.

For the fourth and final test scenario on the first implementation, the generator was able to produce a multiple groupings of islands in the ocean as can be seen in Figure 5.4. The reason why there are multiple separate island groups seen in the picture is related to how the program handles continuous terrain generation process. This does not impact on its capabilities to produce the requirements set in Section 3.1, which are clearly met. This makes the fourth test a success, and with it every scenario is now completed without any failings. Although some issues and errors were present and notified off, the overall used method is still valid for all these tasks.

Following the first implementation and its scenario tests, comes the second implementation with its own results.

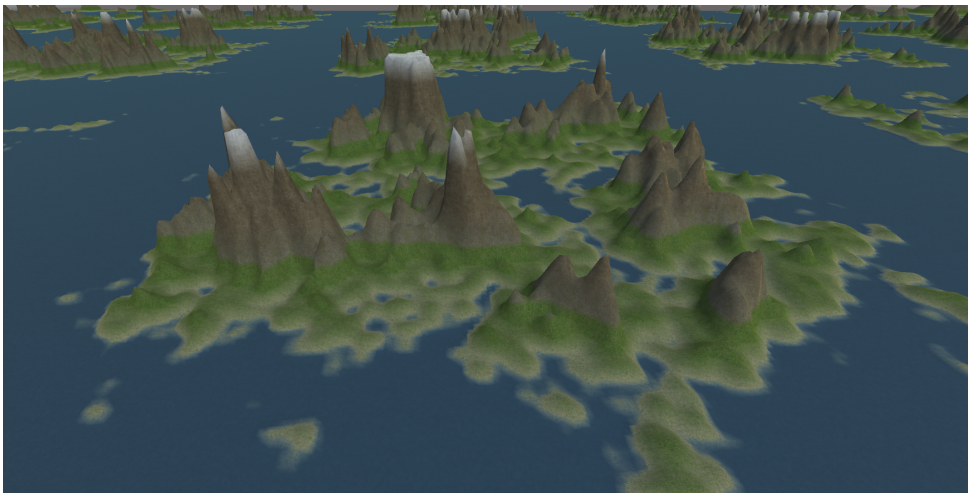

Figure 5.4: Method 1 Scenario 4 - Island & Ocean

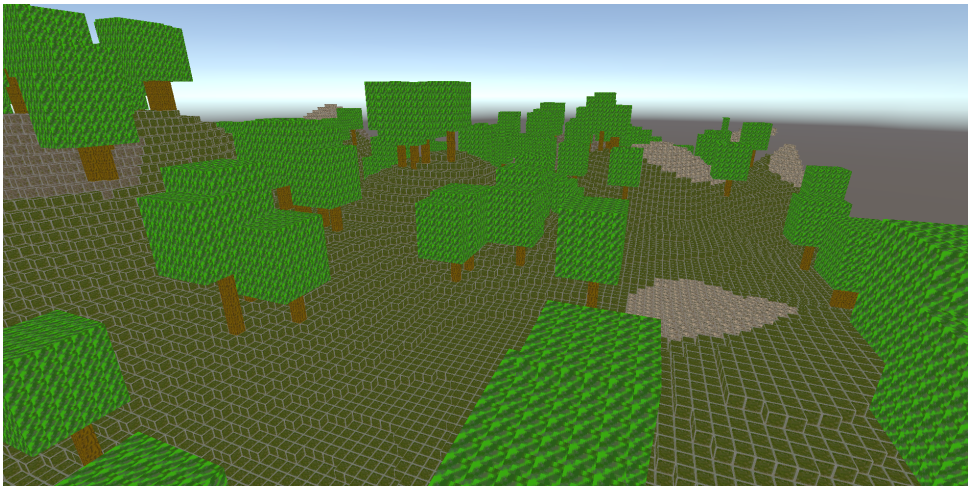

Figure 5.5: Method 2 Scenario 1 - Flat

For the first test scenario on the second implementation, Figure 5.5 shows that the voxel generator implementation was not able to produce a flat terrain as it did not meet all the requirements set for the test scenario. The generator was able to create random positions and groupings for tree objects around the terrain, but that is only one of the requirements. Based on the amount of voxels present in the picture, it is clear there is significant increase and decrease on the general height level of the terrain. The terrain shows curves going up and down on separate points multiple times. This marks down the scenario test as a failure. It is a possibility,

that if the parameters had been different for each of the generated biomes, maybe it would have been able to generate a flat terrain, but for this test run it was not able to do so.

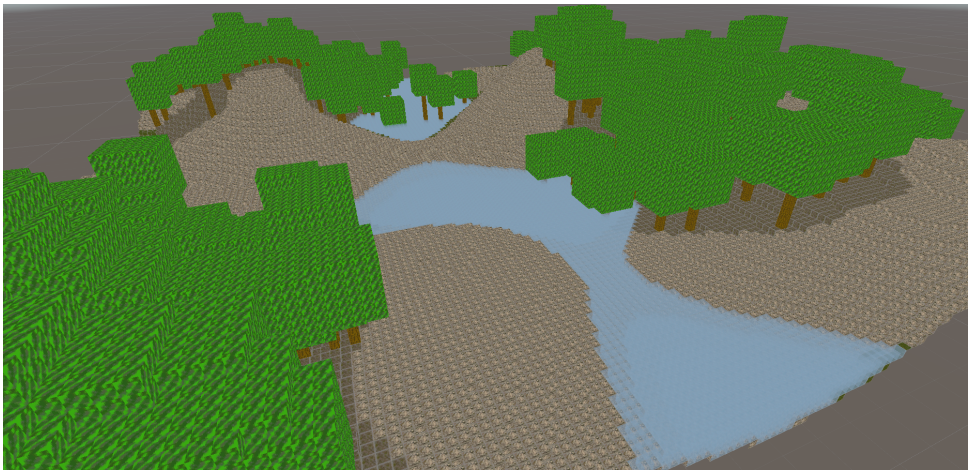

Figure 5.6: Method 2 Scenario 2 - Woodland

For the second test scenario on the second implementation shown in Figure 5.6, the generator was able to produce a terrain which meets all the necessary requirements for the Woodland test scenario. This makes the scenario test a success.

The terrain is split into multiple different sections by the two lakes in the picture, although the splitting is not completely clean as there are trees present in both aquatic environments. The reason for this happening has most likely to do with the order of biome placements during the generation process, as lakes and other similar environments are placed first into the terrain and other biomes are then added on top of it. It is a possibility this cosmetic fault could be avoided by changing the generation order, or presenting some limiting factor for tree placement near biome's edges. This only shows that sometimes it is difficult to control object placement during PTG.

Gathering test results turned out to be a bit more difficult for the third test scenario using the second implementation's generator. During the first few test runs, it was discovered that if the generator produced mountains that were too tall,

the generator would somehow not be able to provide textures required for voxels surrounding the mountain biome. The voxels existed and could be found in Unity's editor when put on pause. No error was shown in the log files, but we believe this problem to be an adverse effect of Unity engine's own limitations when it comes the amount of vertices in a singular mesh as mentioned in Section 4.2. We had been aware of this problem when developing the generator, but had not expected this type of outcome and effect. After setting a hard limit for how tall a mountain can be, the problem got solved and we were able to make tests and take measurements on the third scenario.

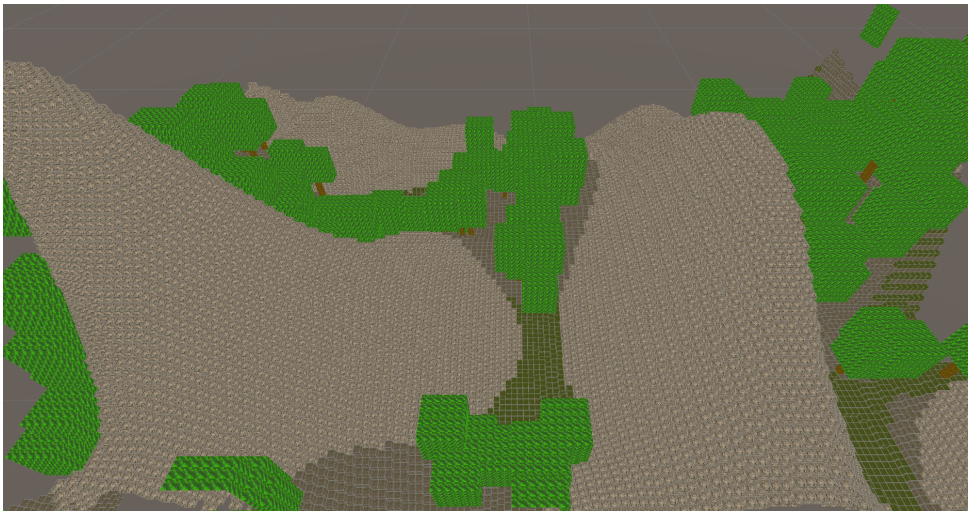

Figure 5.7: Method 2 Scenario 3 - Mountains

As illustrated in Figure 5.7, the generator was able to produce tall, mountainous terrain. Regardless of its capability to produce tall terrain, it will be considered as a failure, for it was not able to generate any lakes into the terrain. Not being able to meet all the requirements for this scenario type marks it as a failed test.

For the fourth and final test scenario with the second implementation presented in Figure 5.8, the generator was able to produce a large ocean area with bald, rocky outcrops spread around the terrain. It did not generate any other objects around the terrain nor into the water. Since the requirement list is smaller for the fourth

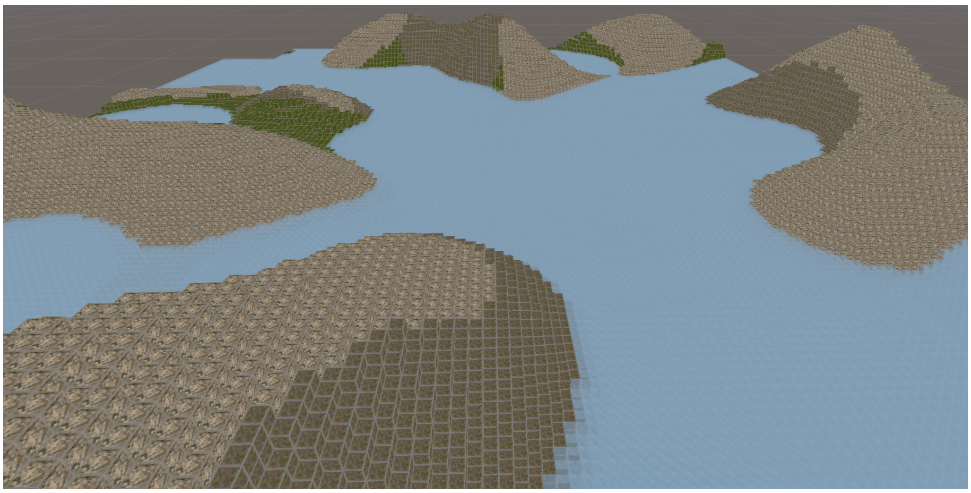

Figure 5.8: Method 2 Scenario 4 - Island & Ocean

scenario, needing only some islands to be generated and the ocean to be on equal level, and the generator's result is able to fulfill the requirements, the test can be considered as a success.

The third implementation's results got summarized into a single figure, seen in Figure 5.9. The top-left corner being the results for Scenario 1, top-right Scenario 2, bottom-left Scenario 3 and bottom-right being Scenario 4. The reason for summarizing the results into a single figure, is because it is a two-dimensional terrain generator and the lack of an additional dimension means there is not much to be seen in individual figures. In other words, it is just more convenient this way.

Results of the third implementation look overall similar to each other, because they all share the same set of textures. The textures are arranged into a tile set, which is then used to provide necessary visuals for each biome type present in the terrain. This gives them a unified look and they could easily be mistaken to be from the same generated terrain sample, but the purpose of this testing was to see if the method could create all four scenario types. Each terrain contains the necessary parts required for each scenario type (trees, plain terrain, water, larger height difference and eventually ocean), making the third implementation a success.

Figure 5.9: Method 3 Scenarios 1-4

Although it is debatable with the third implementation if all the scenarios were successful, because the hardest thing to differentiate in a two-dimensional terrain is the terrain height. In the third scenario sample, the height difference is clear with the white mountain tops, but how do we know that there is height difference in the second scenario sample since it never had a mountain biome set to be generated for it? The method implementation did utilize all three map types mentioned in Section 4.3 for each scenario. We could assume that there is a height difference in Scenario 2, but because there was no specific biome to represent a middle level height difference it cannot be proven directly, unless the tile set is updated and another test is made.

5.2 Technique Comparison Results

In Figure 5.10 is listed the use of system memory for each scenario with their respective methods. As explained in Section 3.2, the memory usage was gathered during a sixty second period for each scenario to see how resource intensive the implementations would be.

As can be seen from the bottom quadruple, the first method required around 1.6 gigabytes in order to function and do basic terrain generation for the first scenario. After adding features into the generator in later scenarios, the use of system memory steady keeps on rising to 1.75 gigabytes in scenario three. For some reason, the fourth scenario uses 2.05 gigabytes of system memory to generate the terrain. This sudden rise in resource demand should not be a thing, as the fourth scenario has the least amount of features required to be generated for the scenario test. One possible way to explain this resource anomaly is that the system was able to generate terrain even faster, because it was easier for the system to generate, thus using more resources temporarily. There is nothing in the gathered data to outright confirm this claim, so it will remain as a guess.

Each method and scenario pairing 'Method number Scenario number' will be referred to in shortened format 'M number S number', for example M1S1. The middle quadruple shows that the second method required slightly less or about an equal amount of system memory to be able to function for each test scenario, being able to function with 1.5 gigabytes of memory. Unlike with M1S3, M2S3 used slightly less system memory during the test run despite both having the same terrain requirements. We found that the second method's fourth scenario also had a similar resource anomaly as with the first method, although this required closer to 2.5 gigabytes of system memory. This is 450 megabytes more than with the first method. And again, we found nothing in the gathered data that would support the idea that a simpler scenario should use more system memory compared to other

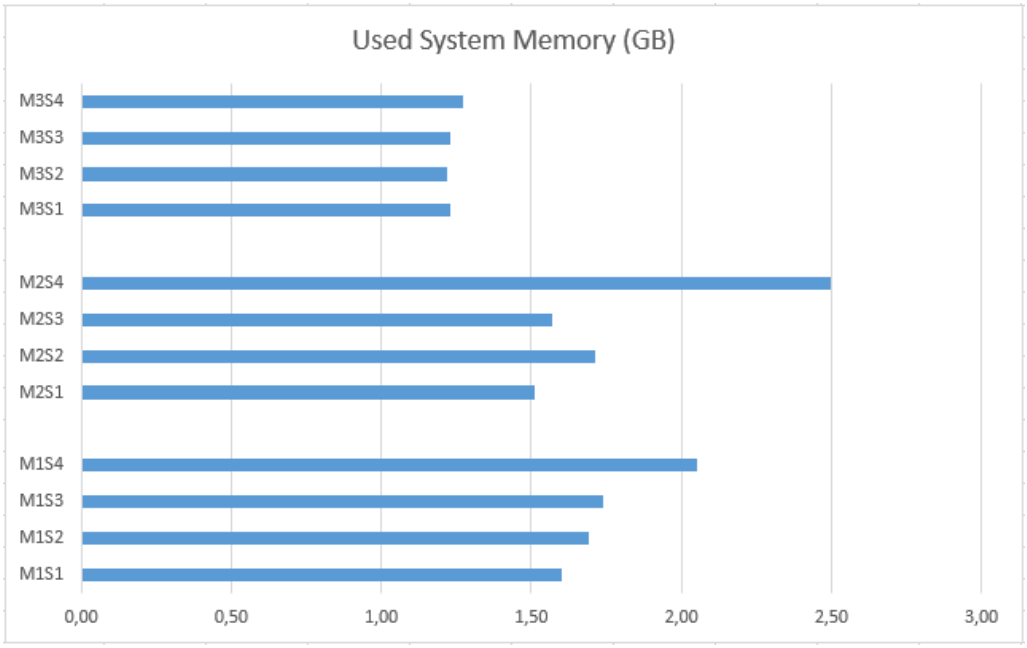

Figure 5.10: Total use of Memory in Gigabytes for each method and scenario.

scenarios with the same method. Therefore, we still suspect it has something to do with the system being able to process the simpler terrain faster, thus being more resource intensive in that regard. It is also a possibility, that this anomaly is being caused by Unity engine that the terrain is being rendered with.

The top quadruple representing the results for third and final method, shows that the method had overall the least amount of performance variances between each testing scenarios. Unlike the two other methods, this one did not experience the resource anomaly. With the gathered data we are also able to see that it used the least amount of resources in general. On average, the third method implementation used 1.24 gigabytes of memory for all testing scenarios, where as the first and second implementations used 1.77 and 1.82 gigabytes respectively for their testing scenarios.

Q1	bad	mediocre	neutral	good	excellent
M1S1		4	2	6	
M1S2		1	5	5	1
M1S3		1	4	7	
M1S4	1		3	4	4
M2S1	1	2	6	1	2
M2S2	2	1	3	5	1
M2S3	4	4	3	1	
M2S4		5	4	3	
M3S1		4	6	2	
M3S2		2	6	4	
M3S3			3	5	4
M3S4			5	7	

Figure 5.11: Results for Evaluation 3.3 Question 1

1	M1S4
2	M1S3
3	M3S3
4	M1S2
5	M1S1
6	M2S2
7	M3S2
8	M3S4
9	M2S1
10	M3S1
11	M2S4
12	M2S3

Figure 5.12: Results for Evaluation 3.3 Question 2

5.3 Subjective Evaluation Results

In Section 3.3 we introduced a set of questions that would be presented to a group of evaluators. This group would then evaluate the generated terrains presented in Section 5.2.

For the first question the evaluators were asked their initial reactions towards the generated terrains, which can be seen in Figure 5.11. The terrains were presented in the same order as they were presented in Section 5.2. For the first implementation,

the evaluators had primarily positive or neutral response to the generated terrains, where as they had more neutral or negative first reactions on the second implementation's terrains. With the third implementation, the reactions on the first terrain were clearly neutral or negative, but the other terrains had more positive reactions to them.

For the second question the evaluators were tasked to rank the terrains from one to twelve. The arranged results can be seen in Figure 5.12. We can see that most evaluators found the first implementation's terrains to be the most interesting for them, with the only exception in this top four being the third implementation's third scenario that takes the third place. In the ranking list, the evaluators preferred the second implementation the least. Third implementation has rankings all over the list, showing that generated terrains split evaluators opinions.

Preferred Implementation		
	Positive	Negative
1st Imp.	67 %	25 %
2nd Imp.	8 %	33 %
3rd Imp.	25 %	42 %

Figure 5.13: Results for Evaluation 3.3 Question 3

For the third question, the evaluators were asked which method and their respective images they found to be interesting in general, and which method they found to be the least interesting. From the gathered results, 67% of the evaluators preferred the first implementation over the others, where as 25% favoured the third implementation. A small margin of 8% of the evaluators preferred the second implementation.

When asked which method and their respective images the evaluators preferred the least, the first implementations gathered 25% of all votes. The second im-

plementation gathered 33% of all votes, and 42% of votes were put on the third implementation.

The arranged results can be seen in Figure 5.13, from which we can see that the first implementation had the most positive and the least negative response to. Despite the second implementation having the least amount of positive responses, it was not the one with the most negative responses. When comparing the results seen Figures 5.12 and 5.13, we are able to see that the results are fairly similar. The least amount of positive preference with a third of all negative votes for the second implementation are noticeable in Figure 5.12.

6 Analysis

This chapter covers the analysis of the results presented in Chapter 5. Starting with the gathered method implementations, and then proceeding over to gathered performance data and finishing the analysis with the evaluation data.

As we are able to see in Chapter 5, most method implementations were able to meet the set requirements for each testing scenario. There were a few generated terrains that did not meet all the respective requirements, usually missing a single needed feature to be considered successful, but otherwise majority of all tests were successful. There were a few instances of something unexpected happening, for example, the terrain in Figure 5.2 having snow textures on hill tops, despite them not being tall enough for the texture to show up. All of these problems most likely could be explained with faulty program parameters or other errors in code, but trying to fix everything would have meant doing all the performance tests over again. This would have meant also postponing the evaluation phase. We were still able to obtain PTG performance data and set up the survey for the evaluators and gather data from the provided answers.

From all the performance and survey data collected, we can observe that for the first implementation, the first impressions of the evaluators were better the more resources the terrain generation had required. This observation, in turn, does not match when comparing performance data to the second implementation and evaluator reactions. This can be seen even better with the third implementation,

as its steady performance does not seem to affect the opinions of the evaluators, with the rankings being spread across the leader board. In general, we can conclude from this that there is no direct connection between the performance data of the implementations and the first impressions of the evaluators.

When comparing the results of the first and second evaluation questions, it can be observed that there are small but insignificant differences between these results. From the first reactions of the evaluators, it is possible to make en estimate of which generated terrains would end up at some point in order of popularity. But when looking at the order of popularity of the second evaluation question, which is based on the scores of the critics, the order of the terrains is actually a little different. This finding, on the other hand, is of little relevance to the bigger outcome, and yet we are able to draw conclusions about which method evaluators prefer most and least.

From the collected results, however, we can observe that the evaluators mainly prefer more realistic or natural looking terrains rather than two-dimensional or implemented with voxels type of terrains. With this observation in mind, we can make the conclusion that developers should prefer PTG methods that are meant to create realistic terrains. This only reinforces the list of most essential requirements set by Fischer et al. [5] in Section 2.2, where they list realism as one of the most essential requirements for a PTG system. Although Fischer et al. also list performance as one of the key four requirements, in this test case performance did not seem to have any larger impact on the results. It is a possibility that performance would have mattered if the generated terrains had been larger in scale and had used more system memory during PTG process, since right now most implemented methods only required on average between 1.2 and 1.7 gigabytes of system memory to operate.

Going back to the research questions presented in Sections 3.2 and 3.3 :

RQ1: How well do these selected terrain generators perform when under duress for sixty seconds?

Figure 5.10 gives us the clear view on the overall performance for each PTG method. In this case, the less an implementation uses system memory the better it is. For these test results, the amount of successful test scenarios is a secondary priority.

The 2D tiling method 'M3' performed the best as it only used 1.24 gigabytes of memory on average during the PTG processes, and it was also successful in creation all four test scenarios. The noise-based method, 'M1', used 1.77 gigabytes of memory on average and the voxel method, 'M2', used 1.82 gigabytes on average. When looking at the amount of successful test scenarios between M1 and M2, the latter had only two successful scenarios out of four scenarios. M1 had all terrain test scenarios generated successfully. From all of this, we can make the conclusion that method M3 performed the best and M2 the worst, leaving M1 in the middle only due to significantly higher demand on memory.

RQ2: How good do the generated terrains look like under the grading of a group of evaluators?

Figure 5.12 illustrates the ranked list for all the generated test scenario terrains. As mentioned in Section 5.3, most evaluators prefer method M1's generated terrains. After that, methods M2 and M3 are spread around the ranking list. M3 has rankings all over the list, but all its terrains are still within the best of 10. M2's terrains are primarily on the bottom of the list, clearly showing that it had the least amount of favour gathered from the evaluators. Figure 5.11 also confirms this, as the first reactions towards M1 and M3 were mainly positive, but M2 was seen either as neutral or outright bad. This puts the methods in the following order of evaluator preference: M1, M3 and M2.

7 Discussion

As a topic, procedural generation and PTG methods are interesting. The more PTG is being researched, the more realistic the synthetic terrains are going to get. Currently it is fairly easy for anybody to start learning about PTG and make their own implementations based on the information found online. The work carried out in this thesis is a proof of that, as I had no prior background in procedural generation and yet I was able to learn and make my own implementations during my research process. Based on that experience, those PTG methods that took the stochastic approach we talked about in Section 2.2.1 were the easiest to approach with my limited programming skills. We were interested in learning doing synthetic terrains through simulation, but with such constraints as time, difficulty to find learning material and programming skill, attempting at creating a simulation engine felt nearly impossible.

When it comes to creating these PTG software and implementations, from what we found and understood as we did our research, is that a large portion of people involved in the PTG research field want to make the use and the PTG process overall easier for more casual people to utilize the tools properly. It was often mentioned in papers as a reason, that people should have an easier time at creating synthetic terrains despite lacking otherwise necessary skills and in general make the terrain creation process quicker to perform; both adjusting and altering the parameters, and rendering the terrain.

For those who would conduct this type of hands-on research we did about measuring PTG implementation performance should not do it like we did. Instead we would recommend learning from our mistakes and do the following changes to the research process:

- *Different engine:* There is not inherently anything wrong with Unity as a game-engine as it offers a lot of tools to make the visualization of synthetic terrains easier, but because with it also comes a lot of unnecessary features and components that might bloat your PTG solution. We would recommend trying Unreal Engine 4 or 5 instead or creating own visualizer.

- *Different performance measurement tool:* As we mentioned in Section 3.2, we used Unity's own Profiler Analyzer tool to take the performance measurements. Although it does what it is meant for, it also suffers from the same problem that it contains a lot of non-essential parts for measuring performance that need to be removed in order to increase the accuracy. Our test results in Figure 5.10 showed unexpected anomalies, some of which we are assuming happened partially because of the Analyzer tool.

- *Larger test terrains:* We feel we should have tried generating larger terrains at once to really push the engine's limits and see what kind of performance data we could get from it. But because Unity still has the problem of how many vertices can be found in a singular terrain mesh, we do not think we could have done this true large-scale terrain test without there being visual faults in the rendered terrain. Thus we do recommend using some other visualizer solution.

- *Better evaluation form:* After gathering the evaluation forms from our volunteers, we got feedback about some obscurities in some of the questions. This made us realize that we should have paid more attention to the overall clarity of

ours questions. We are not sure if the results would have been different if our questions had been better, but reading comprehension is something to pay a lot of attention to when doing this type of evaluation again in the future.

We are still satisfied with our findings and results we got during our research process, as they only confirm that stochastic methods are fast and low-end on resource intensity when it comes to PTG methods.

8 Conclusion

As we have learned, procedural generation refers to the automated process of creating data. This can then be used for many purposes, such as 3D models and textures as assets, both of which can be utilized in entertainment industry and academic research tasks. As our understanding regarding the subject and technology improve, we are able to create more realistic synthetic terrains.

From Chapter 2, we learned that PTG process utilizes generated randomized data, which is then used for creating sets of desired attributes and terrain features for the synthetic terrain scenes. Often using a noise method for various purposes, such as creating height maps. The overall approach to and selecting suitable methods for PTG also varies depending if the user wants their terrain 2D or 3D, as some methods are more suitable for certain types of requirements than others. But each PTG system should strive for realism, good performance and the right balance between usability and flexibility. After this, we gave an overview on four large groups of different PTG techniques and on each approach to PTG and method in every group.

In Chapter 3, we introduced both our research questions, of which the first one was about how well our PTG implementations would perform. We listed our selected hardwares and softwares we were going to employ for this task, along with other requirements related to the synthetic terrains, such as testing scenarios and what the failing conditions would be if our implementations would not be able to generate

the terrain with all the required terrain features. The second research question was about a group of volunteers evaluating the generated terrains, and in return give numerical data for analysis so we could put the terrain in ranking order.

We presented our selected PTG methods in Chapter 4. We gave the introductions to each of our approach and why we had selected them for testing. We also gave our reasons why we had to leave some methods out, or do other changes due to constraints or other problems we did not have the resources to solve properly.

Chapter 5 covered the results of the research. We almost were able to generate each terrain type successfully to meet the scenario requirements, but some synthetic terrains were missing required features and thus had to be considered as failures. From our gathered performance data we had learned of a resource anomaly that was troubling a certain testing scenario when running our 3D terrain methods. We made the assumption that this anomaly was related to Unity itself and not necessarily to our implementation when it comes to rendering terrain.

We also learned that our evaluators clearly preferred our 2D tiling and first noise-based implementation over the voxelized visualization of the terrain. We elaborated this in Chapter 6, where we analyzed our gathered results and what might had gone wrong during our research process. In Chapter 7, I lined out our suggestions on what we would do differently in the future is given the chance to do this type of research again.

From all of this, we can see that procedural terrain generation is a complex subject with still a lot to offer for researchers. It is reasonable to assume we will be seeing even more complex approaches to creation of synthetic terrains in the next decade or so as our understanding on the various PTG approaches increases. Introducing quantum computing to calculate PTG simulations would probably give us unimaginable results.

References

[1] L. O. Valencia-Rosado and O. Starostenko, "Methods for procedural terrain generation: A review", in *Pattern Recognition*, J. A. Carrasco-Ochoa, J. F. Martínez-Trinidad, J. A. Olvera-López, and J. Salas, Eds., Cham: Springer International Publishing, 2019, pp. 58–67, ISBN: 978-3-030-21077-9.

[2] G. Smith, "An analog history of procedural content generation.", in *Foundations of Digital Games (FDG) conference*, 2015.

[3] K. Phillip, *Macromedia Flash 8@ work, Projects and Techniques to Get the Job Done*. Sams Publishing, Feb. 2006, 1st Edition, ISBN: 0672328283.

[4] J. Smed and H. Hakonen, *Algorithms and Networking for Computer Games*. New York: John Wiley & Sons, 2017, ISBN: 1119259762.

[5] R. Fischer, P. Dittmann, R. Weller, and G. Zachmann, "Autobiomes: Procedural generation of multi-biome landscapes", *The Visual Computer*, vol. 36, no. 10, pp. 2263–2272, Oct. 2020, ISSN: 1432-2315. DOI: 10.1007/s00371-020-01920-7. [Online]. Available: https://doi.org/10.1007/s00371-020-01920-7.

[6] J. Doran and I. Parberry, "Controlled procedural terrain generation using software agents", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 2, pp. 111–119, 2010. DOI: 10.1109/TCIAIG.2010.2049020.

[7] M. Blatz and O. Korn, "A very short history of dynamic and procedural content generation", in *Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation*, O. Korn and N. Lee, Eds. Cham: Springer International Publishing, 2017, pp. 1–13, ISBN: 978-3-319-53088-8. DOI: 10.1007/978-3-319-53088-8_1. [Online]. Available: https://doi.org/10.1007/978-3-319-53088-8_1.

[8] I. Karatzas and S. E. Shreve, "Brownian motion", in *Brownian Motion and Stochastic Calculus*. New York, NY: Springer New York, 1998, pp. 47–127, ISBN: 978-1-4612-0949-2. DOI: 10.1007/978-1-4612-0949-2_2. [Online]. Available: https://doi.org/10.1007/978-1-4612-0949-2_2.

[9] J. Doob, *Stochastic Processes*. Wiley, 1962. [Online]. Available: https://books.google.fi/books?id=7Bu8jgEACAAJ.

[10] T. J. Rose and A. G. Bakaoukas, "Algorithms and approaches for procedural terrain generation - a brief review of current techniques", in *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, 2016, pp. 1–2. DOI: 10.1109/VS-GAMES.2016.7590336.

[11] J. Gallostra, *Landscape generation using midpoint displacement*, Date accessed: 19.10.2021, Dec. 2016. [Online]. Available: https://bitesofcode.wordpress.com/2016/12/23/landscape-generation-using-midpoint-displacement/.

[12] J.-D. Genevaux, E. Galin, E. Guérin, A. Peytavie, and B. Benes, "Terrain Generation Using Procedural Models Based on Hydrology", *ACM Transactions on Graphics*, 4th ser., vol. 32, 143:1–143:13, Jul. 2013. DOI: 10.1145/2461912.2461996. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01339224.

[13] K. Perlin, "Image synthesizer", *Computer Graphics*, vol. 19, no. 3, pp. 287–296, ISSN: 0097-8930.

[14] S. Gustavson, "Simplex noise demystified", *Linköping University, Linköping, Sweden, Research Report*, 2005.

[15] S. Lipschutz, *Schaum's outline of linear algebra*, New York, NY, 2012.

[16] K. Perlin, "Improving noise", *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 681–682, 2002, ISSN: 0730-0301.

[17] S. Worley, "A cellular texture basis function", in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 291–294.

[18] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann, 2003.

[19] G. Hettinga, R. Beckhoven, and J. Kosinka, "Noisy gradient meshes: Augmenting gradient meshes with procedural noise", *Graphical Models*, vol. 103, p. 101 024, Apr. 2019. DOI: 10.1016/j.gmod.2019.101024.

[20] K. Choros and J. Topolski, "Parameterized and dynamic generation of an infinite virtual terrain with various biomes using extended voronoi diagram.", *J. Univers. Comput. Sci.*, vol. 22, no. 6, pp. 836–855, 2016.

[21] H. F. Yin and C. W. Zheng, "A practical terrain generation method using sketch map and simple parameters", *IEICE Transactions on Information and Systems*, vol. 96, no. 8, pp. 1836–1844, 2013.

[22] J. Olsen, *Realtime procedural terrain generation*, 2004. [Online]. Available: https://web.mit.edu/cesium/Public/terrain.pdf.

[23] I. Marák, B. Benes, and P. Slavík, *Terrain erosion model based on rewriting of matrices*, 1997. [Online]. Available: https://otik.uk.zcu.cz/handle/11025/15907.

[24] A. Puig-Centelles, P. A. Varley, O. Ripolles, and M. Chover, "Automatic terrain generation with a sketching tool", *Multimedia Tools and Applications*, vol. 70, no. 3, pp. 1957–1986, 2014.

[25] A. P. Mangra, A. Sabou, and D. Gorgan, "TSCH algorithm-terrain synthesis from crude heightmaps.", *Romanian Journal of Human-Computer Interaction*, vol. 9, no. 2, 2016.

[26] F. P. Tasse, A. Emilien, M.-P. Cani, S. Hahmann, and N. Dodgson, "Feature-based terrain editing from complex sketches", *Computers & Graphics*, vol. 45, pp. 101–115, 2014, ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2014.09.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849314000818.

[27] M. B. B. Andersson, F. Gelotte, J. Bjarne Graul Sagdahl, K. Berger, and S. Kvarnström, "Procedural generation of a 3d terrain model based on a predefined road mesh", Ph.D. dissertation, University of Gothenburg, 2017, pp. 1–52.

[28] M. Becher, M. Krone, G. Reina, and T. Ertl, "Feature-based volumetric terrain generation", in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '17, San Francisco, California: Association for Computing Machinery, 2017, ISBN: 9781450348867. DOI: 10.1145/3023368.3023383. [Online]. Available: https://doi-org.ezproxy.utu.fi/10.1145/3023368.3023383.

[29] A. Emilien, U. Vimont, M.-P. Cani, P. Poulin, and B. Benes, "Worldbrush: Interactive example-based synthesis of procedural virtual worlds", *ACM Transactions on Graphics (TOG)*, vol. 34, no. 4, pp. 1–11, 2015.

[30] Y.-T. Yeh, L. Yang, M. Watson, N. D. Goodman, and P. Hanrahan, "Synthesizing open worlds with constraints using locally annealed reversible jump mcmc", *ACM Transactions on Graphics*, vol. 31, no. 4, pp. 1–11, 2012.

[31] H. Zhou, J. Sun, G. Turk, and J. M. Rehg, "Terrain synthesis from digital elevation models", *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 4, pp. 834–848, 2007.

[32] J.-D. Génevaux, E. Galin, A. Peytavie, *et al.*, "Terrain modelling from feature primitives", in *Computer Graphics Forum*, Wiley Online Library, vol. 34, 2015, pp. 198–210.

[33] M. Frade, F. F. de Vega, and C. Cotta, "Automatic evolution of programs for procedural generation of terrains for video games", *Soft Computing*, vol. 16, no. 11, pp. 1893–1914, 2012.

[34] P. Walsh and P. Gade, "Terrain generation using an interactive genetic algorithm", in *IEEE Congress on Evolutionary Computation*, IEEE, 2010, pp. 1–7.

[35] B. Onrust, R. Bidarra, R. Rooseboom, and J. Van De Koppel, "Ecologically sound procedural generation of natural environments", *International Journal of Computer Games Technology*, vol. 2017, 2017.

[36] O. Šťava, B. Beneš, M. Brisbin, and J. Křivánek, "Interactive terrain modeling using hydraulic erosion", in *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '08, Dublin, Ireland: Eurographics Association, 2008, pp. 201–210, ISBN: 9783905674101.

[37] P. Roudier, B. Peroche, and M. Perrin, "Landscapes synthesis achieved through erosion and deposition process simulation", in *Computer Graphics Forum*, Wiley Online Library, vol. 12, 1993, pp. 375–383.

[38] G. Cordonnier, J. Braun, M.-P. Cani, *et al.*, "Large scale terrain generation from tectonic uplift and fluvial erosion", in *Computer Graphics Forum*, Wiley Online Library, vol. 35, 2016, pp. 165–175.

[39] G. Cordonnier, E. Galin, J. Gain, *et al.*, "Authoring landscapes by combining ecosystem and terrain erosion simulation", *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–12, 2017.

[40] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Prentice Hall, Dec. 2002, pp. 1–1080, 2nd Edition, ISBN: 0137903952.

Appendix A Arviointilomake - Suomi

Kysymys 1: Miltä erilaiset maastot näyttävät ensi katsauksella? Merkkaa yksi rasti ruutuun per rivi.

Metodi X Skenaario Y *Method X Scenario Y*	Huono/ Bad	Keskinkertainen/ Mediocre	Neutraali/ Neutral	Hyvä/ Good	Erinomainen / Very Good
M1S1					
M1S2					
M1S3					
M1S4					
M2S1					
M2S2					
M2S3					
M2S4					
M3S1					
M3S2					
M3S3					
M3S4					

Kysymys 2: Minkälaiseen paremmuusjärjestykseen laittaisit nämä maastot? Merkkaa vastaukset välillä 1-12, 1 ollen parhain ja 12 huonoin.

METHOD X SCENARIO Y	VASTAUSKENTTÄ/ANSWERFIELD
M1S1	
M1S2	
M1S3	
M1S4	
M2S1	
M2S2	
M2S3	
M2S4	
M3S1	
M3S2	
M3S3	
M3S4	

Kysymys 3: Minkä menetelmän ja sen kuvat koit yleisesti kiinnostavaksi? Entä vähiten kiinnostavaksi? Merkkaa '+' eniten kiinnostavan kohdalle ja vähiten kiinnostavan kohdalle '-'.

MENETELMÄ / METHOD	VASTAUSKENTTÄ/ANSWERFIELD
M1	
M2	
M3	

Appendix B Evaluation Form - English

Question 1: How do different terrains look like on a first glance? Insert an X to mark your answer, one per line.

Metodi X Skenaario Y *Method X Scenario Y*	Huono/ Bad	Keskinkertainen/ Mediocre	Neutraali/ Neutral	Hyvä/ Good	Erinomainen / Very Good
M1S1					
M1S2					
M1S3					
M1S4					
M2S1					
M2S2					
M2S3					
M2S4					
M3S1					
M3S2					
M3S3					
M3S4					

Question 2: What ranking would you put these terrains in? Mark the answers between 1 to 12, 1 being the best and 12 being the worst.

METHOD X SCENARIO Y	VASTAUSKENTTÄ/ANSWERFIELD
M1S1	
M1S2	
M1S3	
M1S4	
M2S1	
M2S2	
M2S3	
M2S4	
M3S1	
M3S2	
M3S3	
M3S4	

Question 3: Which method and its images were of general interest to you? What about the least interesting? Mark '+' for most interesting method and '-' for the least interesting method

MENETELMÄ / METHOD	VASTAUSKENTTÄ/ANSWERFIELD
M1	
M2	
M3	