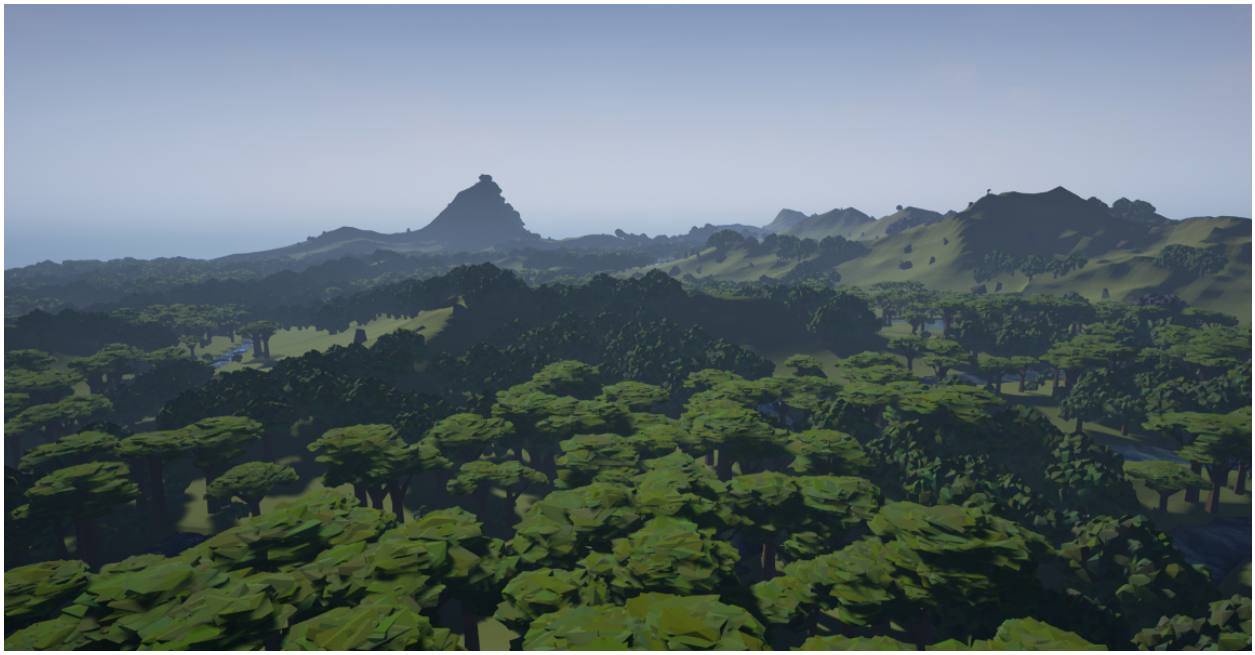
CHALMERS | GÖTEBORGS UNIVERSITET

SeamScape

A procedural generation system for accelerated creation of 3D landscapes

Bachelor Thesis in Computer Science and Engineering



Link to video demonstration: youtu.be/K5yaTmksIOM

Sebastian Ekman
Anders Hansson
Thomas Högberg
Anna Nylander
Alma Ottedag
Joakim Thorén

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2016

SeamScape
A procedural generation system
for accelerated creation of 3D landscapes

Sebastian Ekman
Anders Hansson
Thomas Högberg
Anna Nylander
Alma Ottedag
Joakim Thorén

Supervisor: Marco Fratarcangeli
Examiner: Arne Linde, Department of Computer Science and Engineering

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden 2016

Acknowledgements

This bachelor thesis was done during the spring of 2016 and was a collaboration between students from The University of Gothenburg and Chamers University of Technology. We would like to thank our supervisor, Marco Fratarcangeli, who was a source of inspiration and guidance during the project. Furthermore, we would like to thank *Fackspråk (FKI)* for providing lectures and feedback regarding referencing, report structure and language. Also, we thank the opposition-group for valuable feedback on the report. Finally, we thank our examiner, Arne Linde, for together with his colleagues holding the course DATx02 year 2016.

SeamScape
A procedural generation system for accelerated creation of 3D landscapes

Sebastian Ekman
Anders Hansson
Thomas Högberg
Anna Nylander
Alma Ottedag
Joakim Thorén

Department of Computer Science and Engineering,
Chalmers University of Technology
University of Gothenburg

Bachelor of Science Thesis

Abstract

This bachelor thesis rose from the idea of speeding up game production by using procedural generation techniques. Procedural generation in the project's scope involves the algorithmic production of data which represents a 3D environment. This bachelor thesis describes Seamscape, an application which procedurally generates landscapes in real-time consisting of terrain, water, vegetation, and rocks.

The purpose of the thesis was to use procedural generation as a means to create 3D environments in real-time. Generating landscapes in real-time as opposed to beforehand allows the user to visualize a design quickly, especially for an interface with many design options. However, this project focused more on generating environments than providing freedom of design. Simultaneously, the software design takes user design possibilities into account for future development.

The methods employed are heterogeneous, and apply to specific parts of the landscape. Vegetation was created using L-systems and the terrain was built with noise functions. Distribution of vegetation on the heightmap was done with a technique using ecosystems. The project invented a method to create rivers, based on a selection of studies. Additionally, the rock generation process was tailored to the project, using a low-polygonal style.

The results show that procedural generation of low-polygonal environments consisting of terrain, water, vegetation, and rocks is possible in real-time. Seamscape generates environments with all these features. Future development ideas lay mainly within the domain of diversification and extending the design possibilities for the user. In conclusion, the chosen methods were suitable for procedural generation of landscapes in real-time.

Keywords: Procedural generation, Computer Graphics, Unreal Engine, L-systems, Noise functions

Sammandrag

Idén som ledde detta kandidatprojekt var möjligheten att snabba upp spelproduktion med hjälp av procedurell generering. Procedurell generering i denna rapports avgränsning innebär algoritmiskt skapande av data som representerar en 3D-miljö. Seamscape kallas den produkt som utvecklades. Det är en applikation som använder sig av procedurell generering för att i realtid skapa ett landskap bestående av terräng, vatten, vegetation och stenar.

Syftet med kandidatuppsatsen var att använda procedurell generering som ett medel för skapa 3D miljöer i realtid. Generering av miljöer i realtid istället för i förhand ger upphov för att snabbt kunna visualisera designer. Givet att användaren kan välja attribut i miljön ger det även större designmöjligheter. Detta projekt fokuserade däremot mer på generering av miljöer, och mindre på att ge användaren designfrihet. Emellertid har mjukvarudesignen tagit hänsyn till designmöjligheter för framtida utveckling.

Metoderna som användes är heterogena, och genererar specifika delar av miljön. Vegetation genererades med L-system och terräng genererades med brusfunktioner. Utplacering av vegetationen gjordes med en teknik som använder sig av ekosystem. En metod uppfanns för att generera vattendrag baserat på tidigare studier inom området. Vidare utvecklades en skräddarsydd metod för att generera stenar som följde stilen angående lågt polygonantal.

Resultatet visar att procedurell generering av låg-polygon miljöer bestående av terräng, vatten, vegetation och stenar är möjligt i realtid. Seamscape skapar sådana miljöer. Framtida utvecklingsideer ligger främst inom domänen att skapa variation och utöka designmöjligheterna för användaren. Sammanfattningsvis var de utvalda metoderna lämpade för procedurell genering av landskap i realtid.

Nyckelord: Procedurell generering, Datorgrafik, Unreal Engine, L-system, Brusfunktioner

Contents

Glossary

Convex hull The convex hull C is a convex set that encloses points in set S as tightly as possible. 24, 25

L-system A Lindenmayer system (L-system) consists of a set of production rules and an alphabet of symbols. From a given initial string of symbols, the production rules describe how to expand the current word. These rules take one combination of symbols and turn them into another combination of symbols. This process in turn describes a new word onto which the process can be iterated. 10

Perlin noise A type of pseudo random noise. The noise output for each point is evaluated by interpolating between influence values. Influence values are generated through the use of randomized gradient vectors. 6

Russian Roulette An algorithm for selecting an element e from a set of elements S, where e_i has a probability $P(e_i)$ of being selected such that $\sum i = 1^N P(e_i) = 1$ sum up to one. 21

Spherical Gaussian A multivariate normal distribution where $\Sigma = I * \sigma$. 25

Worley noise A type of pseudo random noise. The noise value for each point is evaluated by combining distances to nearby Poisson distributed feature points. 7

1 Introduction

1.1 Background

Some may argue that nature on Earth is wild and unpredictable. Describing it as unpredictable indicates that it is to some degree random, which is not necessarily the case. Much, if not all, of nature has a cause and effect relation, for example, the presence of the sun and atmosphere allow flora to grow on Earth. However, nature can still be simulated in some ways using randomness and probability.

Laws governing nature exist, and scientists have discovered many phenomena of nature, such as Fibonacci's phyllotaxis, which explains the symmetry displayed in a vast amount of both flora and fauna.[1] Software developers, especially within graphics, attempt to simulate such laws with modern technology. Within the applications of computer graphics, software engineers and designers portray nature through 3D models. One method of describing it in computer graphics is by using *procedural generation*. Procedural generation in this bachelor thesis refers to the algorithmic generation of data which represents elements of a 3D environment. It can accurately model the symmetry and complexity of nature. For example, a botanist by the name of Lindenmayer described plant species in all their growth phases by developing a way to generate flora [2] procedurally.

Procedural generation sprung out of the game industry in the early 80's. One primary cause for this was that memory constraints limited the physical size of in-game worlds. The IBM PC XT (1983) had a 10 MB hard disk drive and as such it was not possible to store vast virtual worlds on the hard drive disk. The video-game Elite was the first game to feature a procedurally generated world in 1984, with eight galaxies and 256 planets in each galaxy[3]. If each planet in such a setup used 5 kB the computer would run out of disk space $8 \cdot 256 \cdot 5$ kB ≈ 10 MB.

Hard drive space can be saved by procedurally generating galaxies and planets as spaceships approach it, storing the data on RAM and discarding the data when it's no longer needed. Procedural generation can diminish the problem of a finite hard disk drive.

While the memory size of modern computers has substantially increased compared to the machines of the 80's, the issue of limited hard drive capacity persists due to demand in larger and more detailed game worlds. Games such as Minecraft or No Man's Sky boast seemingly infinite worlds which players can explore. Such magnitudes of content in games is currently only possible by using procedural algorithms.

Another more recent use of procedural algorithms has been to remove uncreative tasks from the designer. Expansive game worlds quickly become a resource- and time-consuming to create by hand. Procedural generation can accelerate the workflow of the artist by automating tasks. For example realistically distributing vegetation in a 3D environment can be a tedious task. This can be automated using methods presented by Hammes in *Modelling of ecosystems as a data source for realtime terrain rendering* [4].

While procedural models can generate enormous quantities of data, they are limited by their underlying mathematical models. For example, the environments can often exhibit noticeable patterns. Therefore, within this industry, artists are indispensable for tasks such as post-processing. A central challenge for the future is to generate worlds with varied content, which at the same time follow the intentions of the designer.

1.2 Purpose

Today it is costly both in time and currency to produce games. To further illustrate this, Ph.D. candidate D. Williams explains that the average cost of a designer per year is 60 000 dollars in the US industry[5]. Also, the project assumes that especially big-name games rarely have one designer. To alleviate game development processes, the concern which drove the bachelor project associated with this report was how to aid game developers in creating 3D landscapes for games. More specifically, the purpose of this project is to create a computer program which procedurally generates entire, coherent, biomes in realtime. In this case, *'coherent'* refers to the landscape not being disjointed or elements being out of place. If computer software could produce biomes in real time, then that may speed up game development processes.

However, there are already several products which allow developers to create seemingly realistic environments procedurally, such as *TerraGen* [6]. Because of this, the project narrowed the scope into the creation of *stylized* game environments, meaning they are not realistic, but artistic. Procedural generation of stylized environments is something which has not been explored to a great extent before.

This report discusses the developmental process and the theory behind the methods employed in the project. It describes how the project combined the procedurally generated landscape parts, recounting the steps taken to reach the result. Additionally, it examines the methods used regarding effectiveness and efficiency. Thus, it points out shortcomings and strengths of the methods. The report also evaluates the application from a sustainable development approach. That includes ecological, economic and ethical concerns. Finally, it discusses the future of this project.

1.3 Project goals

There are several layers of goals for this project. Mainly, there are goals concerning what elements the environment should exhibit. However, there are also some secondary goals regarding how to display the landscape, yet that is not the main concern of this project. Finally, there are goals regarding the graphical style, namely, how the landscape looks.

1.3.1 Environment

Most of the project goals regard the creation of the environment. For example, it should feature both water bodies, sky, flora, rocks, and more than one elevation range in the terrain. The elevation range indicates the difference between the highest and lowest points on the terrain, meaning whether it forms a mountainous terrain or a flat, for example, marshland. There should be at least one type of tree and one type of bush. The trees and bushes should be procedurally generated such that individual specimen are unique. The environment should also feature grass. There are no set goals regarding the amount of trees, mountains or water streams rendered.

Another goal within the environment domain is to make the visualized world coherent. The coherence involves logical placement of environmental objects and water bodies. Rendered elements shall together form a consistent world within its realm of logic. For example, there should be less vegetation on high elevations or in rocky areas.

The goal is to use procedural generation for *heightmaps, river networks, wind maps, models for*

trees, plants and stones, and also the *distribution of objects*. Heightmaps are 2D maps which can be used to store data about the elevation of an environment. There will also be a few modules without procedurally generated content, for example the sky.

Regarding the water, the goal is to have river networks and bodies of water with low elevation. While water physics is a sub-goal of the project, it is regarded as low prioritized.

A final aim for the environment is to have animation in the form of wind. The wind should affect procedurally generated plants foremost. A subsequent plan could be to add the wind to other terrain elements such as the water. This objective is not as fundamental as the others, however in a game environment, wind and animation can add significantly to the atmosphere. Wind physics on water surfaces will, however, entail that the water uses physics, which as mentioned is a secondary goal of the project.

1.3.2 Interactivity

Besides the landscape having certain traits, there is the matter of how the user interacts with Seamscape to generate the environment in the first place. The project decided to have a simplistic interface for generating the landscape. Additionally, the target group for the landscape generating software ranges from graphics designers to programmers. Therefore, the interface must limit terminology and technical prerequisites to fit those without a technical background. Thus, no knowledge of procedural generation techniques behind the design should be required.

Note that designing interactions can dabble into psychology, which is out of this scope. Therefore, the focus will be on simple, already tested, interaction design patterns. Interaction design patterns make use of human behavior to fit an interface to the user, which Tidwell describes in more detail in her book [7].

Furthermore, the landscape generating software will strive for a minimal delay when generating landscapes. The focus is to avoid waiting time caused by the generation and to some degree the rendering. However, the project will not optimize every aspect of the content generation and rendering. The optimization will be heavily limited to the choice of method for content generation and rendering.

Finally, the project considers the presentation of the environment. The user must be able to traverse the environment freely, using a camera lens with no perspective distortions.

1.3.3 Art style

On the topic of style, the project aimed for a simplistic, to some extent *low polygonal* terrain. The style is aimed to be similar to that of the game *The Witness (2016)* [8], which uses vibrant colors, partially low polygon structures, and subtle use of textures.

Secondly, the project aimed to have a moderate amount of details in the environment. For instance, plants should branch and display leaf structures. The project seeks to have blocky, low-polygon rocks similar to those in *The Witness*. Many objects in the environment will have a single color, yet be shaded depending on the incoming light.

1.4 Project limitations

Due to time scarcity, prioritization had to be made regarding the contents and premise of the project. For example, the project completely disregards structures which are not naturally occurring. There are no cities, villages or roads. The application generates a limited size terrain. Furthermore, no sound is considered as the project focuses entirely on the visuals. Finally, no fauna will be present in the environment.

A premise which to some degree defines the outcome of the project is the use of a graphics engine. Without such an engine, features such as lighting and rendering would have to be resolved manually. In that case, the project would use a raw graphics library, such as OpenGL, to define the pipeline. The constraint of using an engine has some implications on the project. There will be less room for optimization as the underlying framework of the engine handles the rendering pipeline. However, it also allows the project to disregard problems surrounding rendering. Also, for the project, it is unrealistic to build a pipeline which compares to a dedicated graphics engine in quality.

There is aside from limitation regarding design, the project tools, and sound, also a restriction regarding art style. It will not be modifiable; all generated environments will use the same art style. Furthermore, the art style is limited to not use textures. Finally, the procedural generation techniques will not apply textures onto meshes.

A final limitation of the project is that it will not consider exporting or in any way save the generated worlds. Since this will call for parsing and build a file system that can keep all he necessary data, in such a way that the editor can later interpret this when re-opening the world. When the editor closes, the world, with all the progress and work, will be lost. Developing a file system is a possible extension to the bachelor project.

2 Theoretical background

This chapter aims to explain the theoretical basis for the project. It illustrates the procedural generation methods used for the terrain, plants and rocks. It also explains how water networks are combined with the terrain. Finally, it explains how to combine these elements into one environment. Creating a consistent environment involves the placement of plants and rocks on the terrain.

2.1 Procedural generation techniques for terrain

The problem at hand is how to generate a terrain procedurally. One method is to use *fractal noises* which, amongst others, Smelik et al. explains in a survey of procedural generation methods [9]. It can be used to yield 2D heightmaps of terrains. Fractal noise is a composite word for layered noise with expanding amplitude and wavelength. In a conference proceeding of computer science in Canada, Professor Qin showed that fractal noises can be used to generate textures and terrain [10]. These can be adapted for stylized results.

2.1.1 Perlin and Worley Noise

One needs to choose a type of noise to weave it into a layered, fractal noise. Amongst others, Perlin noise can be used. Perlin noise was developed by Ken Perlin for the purpose of generating natural looking computer graphics [11]. It is self-similar, and displays repeating patterns as *spaced pulses*. The distance between the pulses is called the *wavelength* [12]. The self-similarity allows the construction of fractal noise by combining Perlin noises with different wavelengths. See an example of Perlin noise at Figure 1.

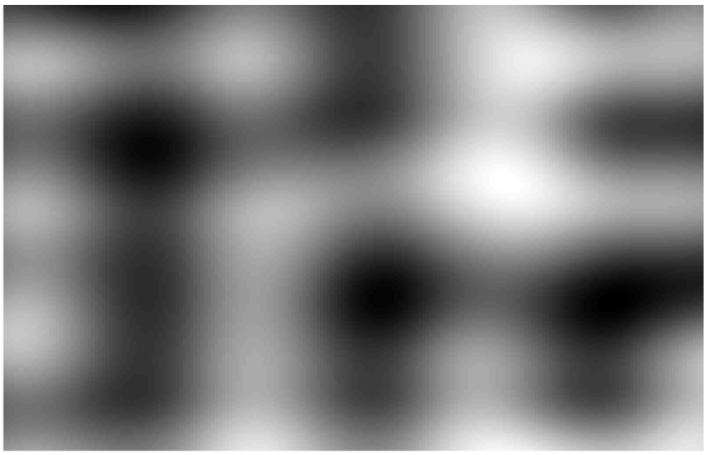


Figure 1: Perlin noise

The output of the Perlin noise function is a scalar value for a given input $P \in \mathbb{R}^N$ and an arbitrary number of dimensions N, as displayed here:

$$\text{Perlin} : \mathbb{R}^N \to \mathbb{R}$$

The domain, namely the set of points, is divided into a grid of noise cells forming N-dimensional cubes. The corner of each cell is associated with a random normalized N dimensional *gradient vector*. The evaluation of the output starts with locating the cell enclosing the input P. Distance vectors between the position P and every corner of the cell are evaluated in the domain set. *Influence values* are calculated by taking the dot product of every distance vector and the corresponding corner gradient. The actual output value is an interpolation of the influence values with a smooth fade function as weight for the interpolation [13]. A N -dimensional cube has 2^N corners. The expression implies an exponential growth of calculations per dimension.

Worley noise, also known as Cell noise, is another type of noise which can be layered into fractal noise. Worley noise complements Perlin noise in that it produces sharp peaks and other features not obtainable through Perlin noise.

Feature points are randomly distributed in the domain set of the noise. Given an input point inside the set, distances between the point of entry and the N nearest feature points are evaluated. These N distances can be combined in many ways and N can vary to generate different output. Also different metrics can be used to assess the distances and also lead to varying results [14].

2.1.2 Distance metrics

Distances can be calculated by using different metrics. The distance metric can be changed to vary the noises. This is a short description on how to calculate the distance between two points in $\mathbb{R}^N$. Firstly, the distance vector in $\mathbb{R}^N$ is written as follows:

$$\Delta\vec{r} = (\Delta x_1, \Delta x_2, \ldots, \Delta x_N).$$

One method to determine the distance vector is by using the *Euclidean distance*:

$$d_{euclidean} = \sqrt{\sum_{i=1}^{N} \Delta x_i^2}$$

Another distance metric is called the *Chebyshev metric*, which defines distances in the following way:

$$d_{chebyshev} = \max\left(|\Delta x_1|, |\Delta x_2|, \ldots, |\Delta x_N|\right)$$

A final method is called the *Manhattan metric*:

$$d_{manhattan} = \sum_{i=1}^{N} |\Delta x_i|$$

All mentioned distance metrics are defined and explained in further detail by Zadora et al. in their book called *Statistical Analysis in Forensic Science* [15].

2.2 Procedural generation of water networks

Various methods exist to generate water bodies such as rivers and streams procedurally. One such method, invented by Benes and Forsbach, is through simulation of erosion in an already generated terrain [16]. Another technique is to produce an actual network, or graph with geographical information, which is placed in the terrain, presented by Génevaux [17]. What these two approaches have in common is that they require some, and in the case of the Benes et al.'s method the complete, knowledge of elevation data. There is at least one method which does not have this requirement presented by Kelley et al., but the method is limited in details such as winding rivers and smoothness [18].

There exists another method inspired by Génevaux [17], where the network is created before the terrain. In this approach, the network is not based on underlying terrain. However, as all the mentioned methods, it produces networks covering a whole map in one run. The generation is done in several steps.

In summary, the project uses a custom made method which is partly inspired by existing ones. In the project's approach, points are distributed randomly in a two-dimensional area, including root points. Then the root points connect to others using a curve approximation. The curve is produced iteratively until a stopping condition is met. After this, a heightmap and the data needed for the visualization of the water surfaces are created. The following section describes the process in more detail.

2.2.1 Distribution of points

The first stage in the generation is the initialization step. Here potential coordinates for the water to flow through is distributed. Firstly, starting coordinates are generated. These are placed randomly, and define the outlets of the network. The outlets are for example the border of a generated area, a coastline. From these *root points* the network is grown. Around the roots, something referred to as *inner points* is then distributed. They define the flow of the water. The points are randomly distributed coordinates and are generated with a given density. The larger the area, the more points are created.

2.2.2 Network generation

In this stage, the water network is constructed. The root points are first placed in a pool. Afterward, the following procedure begins:

1. Chose a random point from the pool.
2. Find the closest inner point that has not yet been used, i.e. is not or have not been in the pool.
3. Create a curve between these points.
4. Check whether the curve crosses already created curves. If it does, ignore the point found in 2 and go back to 2. Also, go back to step 1 if all no point is suitable, and remove the point from 1 from the pool.
5. Save the curve created in 3.

6. Monitor the stopping condition: there are no points left in the pool.

All inner points which are not part of the network at this stage are discarded.

2.2.3 Value association

After producing a river network, a value association is assigned for all points. The values specify the size of the water body, the flow, and the height of the terrain at each corresponding point.

The size value for any point is defined as the sum of the size value of points associated with this one. If there are no such points, the size value is set to 1. The size values must be associated first, as the height values depend on them.

The association of height values starts by setting the height of the root points, h, to zero and the slope value, d, to 1. All connected points are assigned values iteratively. For example, if p is a point with assigned values and c is not, then the slope value is given by

$$d_c = d_p - 4 \cdot \frac{s_c}{s_p}(\frac{s_c}{s_p} - 1) \cdot w,$$

where s represent the size value, and w is an adjustable weight introduced to create variation. The height value is then given as

$$h_c = h_p + |\mathbf{c} - \mathbf{p}| \cdot d_c.$$

2.2.4 Heightmap

After performing the value association it is possible to generate a heightmap. The height in one point x is calculated using all curves in a given radius of the point. Call the points on these curves that are closest to x $\{p_0, p_1, \ldots, p_n\}$, where p_0 is the point closest to x. The height values $\{h_0, h_1, \ldots, h_n\}$ are an interpolation between the height values of points connected by the curve. The height in x on the heightmap is defined as:

$$h_x = \frac{|\mathbf{p_0} - \mathbf{x}|}{|\mathbf{p_1} - \mathbf{x}|}h_0 + (1 - \frac{|\mathbf{p_0} - \mathbf{x}|}{|\mathbf{p_1} - \mathbf{x}|})\frac{\sum_{k=0}^{n} h_k \cdot (r - |\mathbf{p_k} - \mathbf{x}|)^2(3 - 2(r - |\mathbf{p_k} - \mathbf{x}|))}{\sum_{k=0}^{n} (r - |\mathbf{p_k} - \mathbf{x}|)^2(3 - 2(r - |\mathbf{p_k} - \mathbf{x}|))},$$

In the above equation, r is the radius of the area. A polynomial function is used to ensure the derivative of the heightmap is continuous. The function, $f(x) = 3x^2 - 2x^3$, has a derivative of zero in $x = 0$ and $x = 1$. However, an issue arises when $P = \emptyset$ as p_0 does not exist. Here h_x is the height of the closest point on the closest curve, which will cause discontinuities.

The polynomial function above is also used to combine the height values from the water network with terrain height values. When x is close to the curves, only the height values are used. However, when it is further away, other values are used to create variation. Therefore, a *weight value* is defined as

$$w_x = \begin{cases} (r - |\mathbf{p_0} - \mathbf{x}|)^2(3 - 2(r - |\mathbf{p_0} - \mathbf{x}|)) & \text{when } |\mathbf{x}| < r \\ 0 & \text{otherwise} \end{cases}.$$

Finally, the width of the curve is defined as the size associated with the endpoint. When x is within the width of the curve, the height value is lowered. The height drop creates the ditch in which the water flows.

2.3 Procedurally modeling flora

While there are several methods for procedural generation of flora, one of the most prominent methods used is something called L-systems. *L-systems* can model plants and natural structures realistically using a formal mathematical language.

2.3.1 An introduction to L-systems

To understand L-systems one must first understand formal languages. A formal language is a set of words constructed by a formal grammar. The words consist of symbols and letters which are pre-defined for it. In turn, a formal grammar is a set of productions rules which transform words into new words. However, the meaning of the depends entirely on the interpretation.

L-systems are a rewriting grammar which was developed by biologist Aristid Lindenmayer [2] for the purpose of representing different flora with a mathematical notation. In short, L-systems can generate a *word*, using formal grammars. Each symbol in the string will represent information about parts of the plant. A program translates the word into 2D or 3D shapes which finally become a whole plant.

Formally, a L-System is defined as a triplet $G = \{V, \omega, P\}$ where V is the alphabet (V^* the set of all words over V, and V^+ the set of nonempty words over V), ω is the axiom (a non-empty word $\omega \in V$) and P ($P \subset V \times V^+$) is a finite set of production rules on the form $(a \to \chi)$ where $(a, \chi) \in P$, and a is called the *predecessor* and χ is called the *successor*. In summary, the L-system is given an initial word (the axiom) and changes the axiom into a new word by following the rules.

There exists a diversity of L-systems, each which can create plants of varying complexity. The following section summarizes Lindenmayer's *DOL-Systems*, *Stochastic OL-Systems*, *Bracketed L-Systems*, *Context-sensitive L-System*, and *Parametric L-Systems* and their usage.

2.3.2 Types of L-systems

The simplest kind of L-systems Lindenmayer defined are known as DOL-Systems, **D**eterministic context-free (**0**-context) **L**-Systems. A context-free L-system is deterministic iff there exists one production rule for every symbol, thus always providing the same result for the same axiom. DOL-systems are mostly used to produce fractals.

An example is the Algae L-System:

$$V = \{A, B, \varepsilon\}, \omega = A, P = \{A \to AB, B \to A\}$$

1. A
2. AB
3. ABA
4. ABAAB
5. ABAABABA

However, deterministic L-systems can only produce one specific output with no variation. To overcome this, Lindenmayer defined a *stochastic L-System*. It solves the problem of determinism by introducing more than one production rule per symbol, each rule with a probability of being selected for each step.

Due to the linear structure of any OL-system, branching structures of various flora, such as twigs or branches were absent. Lindenmayer solved this by giving a tree topology to the word; something called *bracketed OL-systems*. Lindenmayer introduces the two symbols '[' and ']' in the string to represent the axial structure of the word. The brackets allow the L-system to branch.

Additionally, a production rule can require a context, that is, a production is only legal for some predefined set of left neighbors and some other predefined set of right neighbors. For example, a context-sensitive system could perform the rule for B iff B comes after an A and is followed by three Bs. The neighbors are formally called the *context*. Lindenmayer states that this is useful for simulating the flow signals or hormones in a plant, and can more accurately describe its natural growth.

A final type of L-system is the *Parametric L-systems*, where parameters accompany some of the letters. For example, the parameter could represent the percentage of nutrients in the current branch, which makes the branch slowly narrow. A *condition* determines if a parametric production rule is legal, which for example enables L-systems to grow leaves on a branch only if the nutrient on that branch exceeds some threshold. A parametric symbol is called *module*. If no legal production rule on a module exists the *identity rule*, $X \rightarrow X$, is assumed.

2.4 Distributing vegetation on a given heightmap

Now that vegetation theory has been presented it is time to illustrate the theory regarding how to distribute it in the terrain. Johan Hammes defined a way to calculate ecosystems and placement of plants geometrically for a given heightmap [4].

To accomplish a varied and seemingly natural distribution of vegetation, a technique based on ecosystems is provided. For example, one ecosystem can flourish in valleys where water is gathered after rain while others thrive on steep hills. In Hammes' theory, all ecosystems consist of one or more *ecosystem layers* [4]. The layers are mixed to increase variation of the ecosystems.

Each ecosystem layer is limited to a level of detail. The lowest level does not show any objects while the highest renders even small rocks and grass. The ecosystem layers are computed recursively until the predetermined level of detail is reached. All ecosystem layers control a set of species. Each species are distributed randomly within the affected ecosystem [4].

Hammes defines ecosystems using several parameters. All variables have an associated maximum, minimum and sharpness value. The maximum and minimum values define the extent to

which the ecosystem layer prefers the variable. The sharpness represents how much the layers overlap each other. The proposed parameters to manipulate the ecosystems are as follows [4]:

1. Elevation: the altitude an ecosystem thrives in. The conditions may become harsher at a higher altitude, but the competition for resources decreases.
2. Relative elevation: used for calculating local valleys and ridges. Valleys are sheltered and usually more water gathers there.
3. Slope: on steep slopes water quickly flows off the surface. Also soil depth is heavily dependent on the degree of the slope.
4. Skew (slope direction): Affects sunlight hours and wind exposure.
5. Noise Distribution: used to enhance the effect of grouping within vegetation. Grouping occurs mainly because of plant reproduction benefits.

3 Method

This chapter describes what methods were employed in the project and why. It also provides a detailed step-by-step guide to the actualization of the project goals. Firstly it explains the tools used to realize the project. Secondly, it describes the general software structure, later going into the details of each software section.

3.1 Development tools

This section describes the tools used to realize the project goals, which foremost includes the chosen programming languages and the graphics engine. For a software project, the adopted programming language, as well as the graphics engine, have a striking effect on the resulting application. While a procedural generation of environments in 3D is not locked to use a specific programming language, there are languages which are more suitable than others. Therefore, the chosen tools are punctuated here.

The project chose the following instruments:

- *Unreal Engine*: This is the graphics engine selected by the project. Unreal Engine is both a graphics and a game engine and has recently become free to use. It is used to abstract the work outside the project, such as lighting and other issues which would slow the project down. The report provided a detailed reasoning about graphics engines in Section 1.4.
- *C++*: This is the programming language selected by the project, as Unreal Engine provides a programming environment for it. C++ is commonly used for graphics simulations as well as game development and also works close to the machine, translating directly into machine code. It is, therefore, practiced in this project.

3.2 Software structure

The software structure of the project describes how the modules relate to each other. This section describes what modules the project includes, and the relation between them. Note that the report will display details about the software model in the next chapter disclosing the result.

The full software lies within an Unreal project, wherein the modules for the software model are separated. The modules correspond to the different aspects of the environment; the vegetation, placement, rocks, terrain, and water. The report will unveil more about the methods applied in each regard in the following sections. The call structure shown in Figure 2 resembles in ordering of generation processes and dependencies in between these processes.

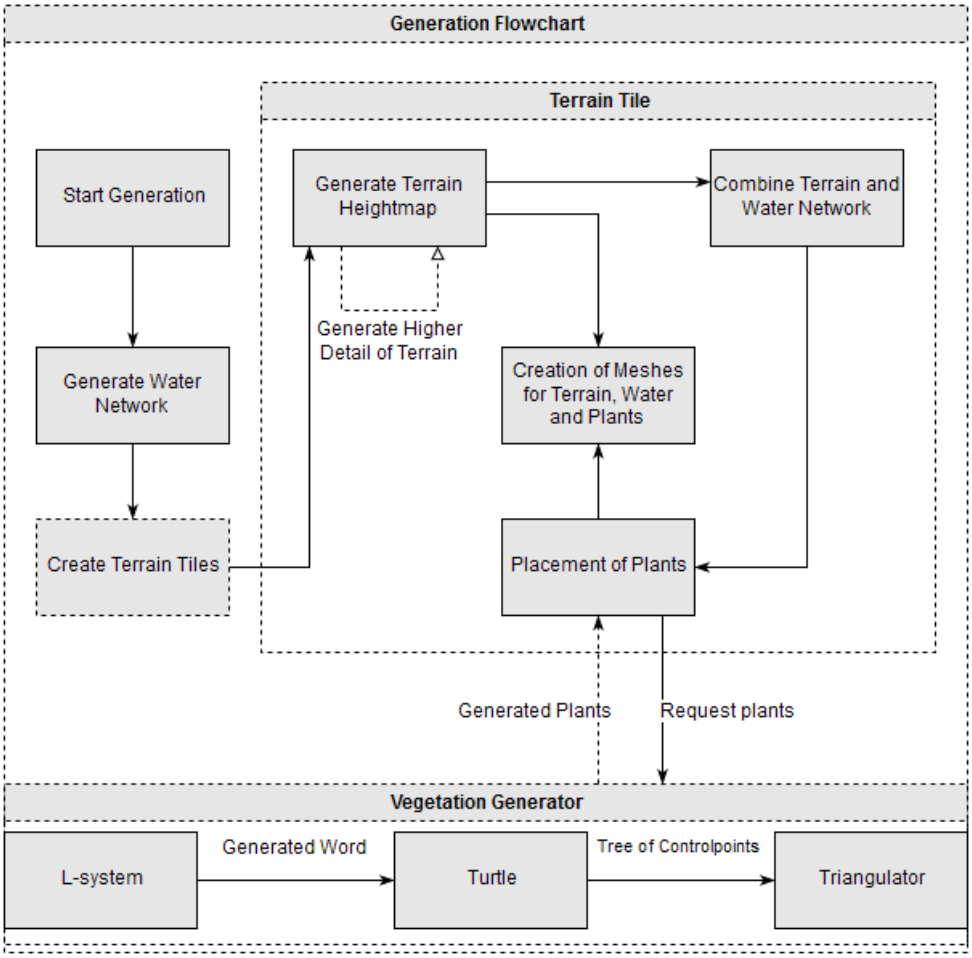


Figure 2: Flowchart showing the general call structure

3.3 Parallel computing

In an attempt to maintain high performance in Seamscape, threading was implemented at an early stage of the project. Separating the generation process allows the rendering of frames to continue while the generation is still ongoing. This section illustrates how Seamscape was parallelised.

When rendering a tile in a higher level of detail, a separate thread is started to handle the workload. The new thread, in its turn, starts generating different noise octaves in separate threads. The noise threads work together with another thread for producing a water network heightmap. The initial process performs procreational work in parallel then waits until results arrive. Whenever two noise octaves are complete, they are summed. The other octaves are then summed onto them to produce the final heightmap. Finally, the water network heightmap is summed onto the full noise using masking values as mentioned at the end of Section 3.6. The heightmap is then ready for the graphics engine.

The generation continues with the placement of plants but here is no further parallelization. The gathered data is loaded in when done and the generation process is finished for that level of detail.

3.4 Hash function and seed

In Seamscape, which relies on random values, there is a necessity for using a hash function. The project chose a hash function featuring a low number of collisions and high randomness. It is called XXHash[19] and was developed by data compression expert Yann Collet. XXHash is used directly as a generator for random values. The input of the hash function includes coordinates or other parameters depending on the application. The hash function is used by the noise algorithms, river generation, and in the distribution of vegetation and rocks. Finally, note that it is possible to regenerate the same landscape after it is discarded. It is achieved by using the same hash function input *seed* another time.

3.5 Terrain

This section describes how the project implemented the terrain generation. It describes the techniques used for producing it and why they were used. The procedure description includes the modeling and generation methods, and secondly, the terrain visualization method.

3.5.1 Modeling the terrain

The terrain is modeled through heightmaps. The motivation for the usage of heightmaps is because of their simplicity. Also, because the project does not include caves and overhangs, a more advanced model is not required. Due to these limitations, only a 2D model of the terrain is needed. Furthermore, this means there is no need to model the terrain with voxels or more complex models than a heightmap. Voxels are three-dimensional pixels, which can be used for modeling for example overhangs.

To enhance performance a dynamic level of detail was used. *Level of detail* entails the varying of detail on objects depending on how far away meshes are from the camera. Because of the dynamic level of detail, regions had to be regenerated during runtime with different resolutions. To ensure that the same location is always generated at the same coordinates, a hash function was used. In this project, it produces seemingly random, uniformly distributed values.

The heightmaps in this project are generated by combing two methods; noises and a river generation algorithm. Another way to make heightmaps is by combining a heightmap replica of a real location with coherent noise, as illustrated by Parberry in the Journal of Computer Graphics Techniques[20]. However, processing images from Earth into heightmaps requires more steps than pre-defined noise functions. Also, the use of realistic heightmaps from the world may impede the graphics style desired for the project.

3.5.2 Generating the heightmap

This project uses Perlin- and Worley noise to generate heightmaps. By adjusting parameters for each layer in the noise function, a high level of design control is obtained such that the style of generated terrain can be fine-tuned.

Other methods for procedural generation of heightmaps are Fourier synthesis and the Diamond-square algorithm. The main reason Fourier synthesis was not chosen was that it generates the entire heightmap at once [13]. Seamscape namely generates subregions independently. Also,

Fourier synthesis uses heavy computations. For example, Fast Fourier transforms [13]. Therefore, Fourier synthesis is not suited for a run time generation. The diamond square algorithm is faster than the Perlin noise algorithm but has more visual artifacts [21].

3.5.3 Choice of noise functions

The two types of noises implemented were Perlin and Worley noise. Perlin noise works well for generating smooth features but lacks the ability to generate sharp and jagged edges which are complemented by Worley noise. Worley noise is capable of generating a broad range of different patterns by changing distance metrics and combining the distance to the feature points.

The various distance metrics that are employed by the Worley noise implementation are Euclidean, Manhattan and Chebyshev. Steven Worley stated that Manhattan and Euclidean metrics produce visually different results [14]. A blog [22] mentioned the Chebyshev distance and by experimentation, it was confirmed that the Chebyshev distance, in comparison to Euclidian- and Manhattan distances, produced visually different heightmaps (see Figure 14 in Section 4.1.5).

Four procedures of combining distances to the feature point to compute the Worley noise was implemented. One uses the smallest distance. Another uses the difference between the two shortest distances. The third uses multiplication of the two shortest distances. The final procedure uses the mean value of the four shortest distances. These techniques were inspired by a paper from Linköping's University [23] and a blog [24]. The methods were further tested within the project. See Figure 13 in Section 4.1.5 for examples of heightmaps showing the various ways of combining distances.

The Worley noise was implemented by dividing the domain into a grid of cells. The cell containing the input point will be referred to as the center cell in this section. The implementation imposes a minimal amount of feature points in each cell. It ensures the feature points needed for calculating the output are located in the center cell or neighboring cells.

While Perlin noise was chosen for this project as one of the two types of noises, there are many alternatives. An alternative to Perlin noise is Simplex noise. The complexity of Perlin noise is $\mathcal{O}(2^N)$ whereas that of Simplex Noise is $\mathcal{O}(N^2)$ according to an article from Linköping University [25], where N is the number of dimensions. Therefore, Simplex noise has better complexity when dimensionality exceeds two. Since this project only requires noises in 2D, this benefit of Simplex Noise can be disregarded.

Another quality of Simplex noise is the lack of anisotropic artifacts [25], which Perlin suffers due to the underlying grid of noise cells. While Simplex noise could have been an addition to the noises used in the project, the other two were of higher priority. Due to time constraint, Simplex noise was not implemented.

3.5.4 Noise maps

The domain of the noise functions is divided into a grid of tiles. Every tile encloses a rectangular lattice where the elements correspond to noise outputs, see Figure 3. Every lattice can have an individual number of elements and spacing. This way a heightmap with differing level of details at provided regions are modeled. A tile corresponds to a region with a given level of detail. Likewise, the lattice elements correspond to height levels at given positions. Every tile is independently generated which is why a hash function is needed to ensure continuity between

neighboring tiles.

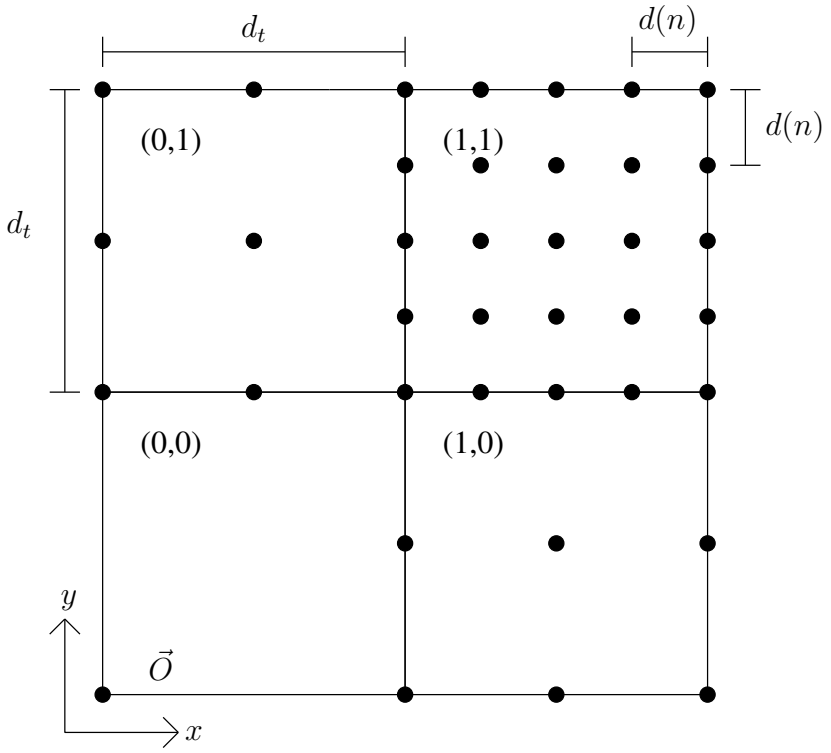


Figure 3: An example of a 2x2 grid of tiles. Point $\vec{O}$ denotes the origin of the global Cartesian coordinate system. d_t denotes the side length of the tiles. Note that different tiles have different number of lattice points n and different $d(n)$ depending on the level of detail. Tiles indices are marked in Figure 3. The dots mark the position of noise outputs T_{ij}.

Apart from dividing the domain set into a grid of tiles, it is also split into a grid of noise cells, see Figure 4. The Perlin noise function implementation uses the positions of the cell corners as inputs to the hash functions. Thus, the function generates the corner gradients vectors. The Worley implementation uses one corner position as hash input to produce the number of feature points and their positions within each cell. The method of finding the needed coordinates for the hash functions are explained in detail in Appendix B.

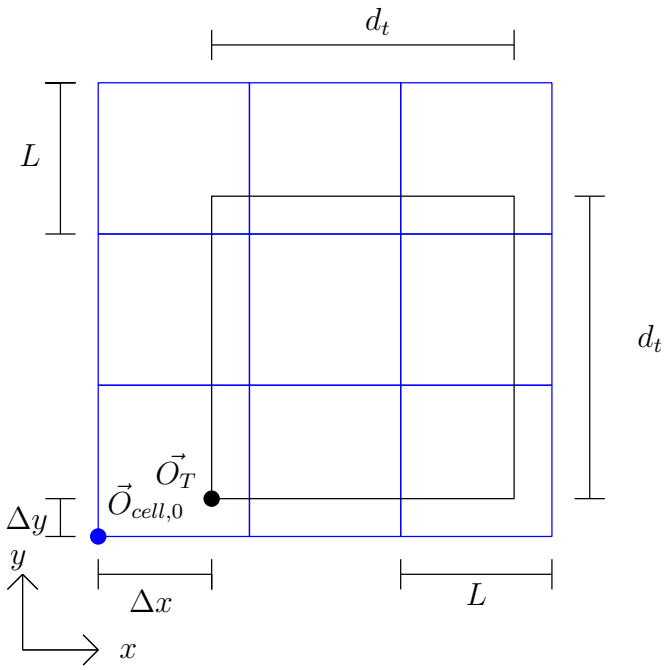


Figure 4: A tile with sidelength d_t and some noise cells. The noise cell length is denoted by L. Δx and Δy are the displacements between $\vec{O_T}$ and the origin of the noise grid $\vec{O}_{cell,0}$ in the x and y directions.

3.5.5 Visualizing the terrain

The visualization of the terrain lies mainly in supplying Unreal Engine with data it can render. What lay in the project's domain is the performance issues, and creating the data for Unreal Engine.

The interface toward Unreal controlled how the program loaded data. For example, the program supplied Unreal's interface with vertices, indices, normals and colors constituting 3D models. Furthermore, it is the project's understanding that data can not be loaded into Unreal's interface in parallel. This premise limits how much Unreal can operate simultaneously. Also, algorithms for fast rendering were used but were also restricted by Unreal engine's implementation. Additionally, there are limited options for loading procedural data in Unreal. In this project, a high-level approach has been selected. It used the interface called Procedural Mesh Component.

Finally, the level of detail gives the system a performance boost by performing modeling work on demand. When a tile needs higher detail, a separate process is created. However, the software holds already generated data in memory. Thus, if the terrain tile can once more use a lower level of detail, the data does not have to be regenerated.

3.6 Water

The project limited the presence of water to arbitrary rivers and stream networks. As it was already decided that the terrain noise was to be generated on demand, in a parallelized manner, the water network had to be produced before that, as mentioned in Section 2.2. The project could not locate an existing appropriate method fulfilling that criteria, where the most suitable was the work by Kelley et al.[18], which was also mentioned in the theory chapter. Therefore,

the tailored method from the theory section was implemented, inspired by Génevaux et al.[17] A critical requirement of the water generation method was its compatibility with the terrain generation, since it is dependent on the erosion introduced by rivers.

The method was based upon the idea in which points are uniformly scattered throughout the world. From these, a set of lines are created by iteratively randomly choosing two of these points, constituting the start point and end point of said line. The lines translate directly to rivers.

Our implementation uses quadratic Bézier curves to guarantee continuous tangents between lines. They are defined, by three points, as

$$B(t) = (1 - t^2)P_0 + 2(1 - t)tP_1 + t^2P_2,$$

where $t \in [0, 1]$. P_0 and P_2 are the points being connected by the line. P_1 is called the control point. $P_1 - P_0$ is a tangent in P_0 and $P_2 - P_1$ is a tangent in P_2, where the second is used in the connection with the next curve.

The use of Bézier curves allows winding waterways but makes finding the closest point in the heightmap more complex. The closest point calculation uses an equation of degree three, which is unnecessarily computationally heavy. Therefore, a simpler method was used where a sample of straight lines going in various directions from the point in the heightmap was created, and determined at what point they crossed curves close by. Then the closest point of crossing was used as distance to the curve. This type of test reduced the problem to a series of quadratic equations instead. Also, the fact that the Bézier curve is contained within the three points allows for smarter intersection testing. Instead of checking whether two curves intersect directly, it is examined whether line segments created between the points of two curves intersect.

A smoothing term is then used on the heightmap, since using only interpolated height values of the closest lines, caused visual discontinuities. It allows all lines in the vicinity of the height value being evaluated to contribute to the value. The difference between the initial and final method can be seen in Figure 5. After the water network generated its heightmap, it was split into tiles analogously to the terrain's noise map, and combined using the weight values mentioned in Section 2.2.4.

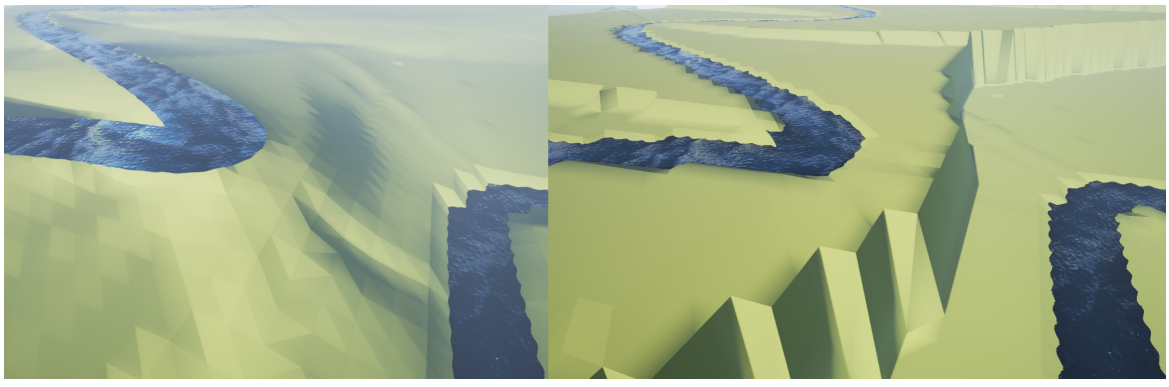


Figure 5: Terrain smoothing example. In the left picture a smoothing term has been added to the heightmap values.

Finally, the visualization of the water is a surface created along the Bézier curves, placed in the ditch the water map yielded. On this surface a reflective material created in material editor

of Unreal Engine was added to imitated small water waves. Another approach would be to use physics to simulate water. There is, for example, a library called Flex by Nvidia which contributes with such features. However, water physics was a secondary goal of the project.

3.7 Vegetation

The project's aimed vegetation consists of plants such as trees, shrubs, flowers and grass. This section describes how the project modeled the plants, through all stages until visualization. Firstly, however, it motivates the method used, comparing it to other popular methods.

There are numerous methods for generating vegetation. For example, SpeedTree is a toolkit for creating vegetation using procedural generation [26]. SpeedTree generates vegetation offline according to a forum post[27] by Greg Croft, the author of *6 ways SpeedTree "speeds" the creation process* [26]. However, offline generation is incompatible with this project. The trees in this project must be generated runtime and as such SpeedTree was not a feasible option.

Another method presented by Runions A. et al. utilizes a space colonization algorithm to create tree structures [28]. However, the expressiveness of this method is more limited than for example L-systems which was mentioned in Section 2.3. Structures generated by L-systems are given from user-defined production rules whereas structures produced by space colonization are defined by algorithms used with a set of user-defined parameters. Therefore, the project chose to use L-systems to generate vegetation in this project.

3.7.1 Creating the formal grammar: An L-system framework

The following sections describe the implementation of the L-system framework. It limits itself to presenting the most crucial details. The L-system framework source code is over 600 lines of code; hence minor details are omitted.

The central challenge was to combine the different variations of L-systems into one generalized L-system, as the source of theory by Lindenmayer et al.[2] regarding L-systems did not explain how to combine the separate systems. Through discussion and reasoning, most combination-related issues encountered were solved. However, right-context matching did not work in conjunction with bracketed- and parametric production rules (see discussion in Section 4.1.2) and was, therefore, omitted. The L-system method is illustrated using pseudocode in Algorithm 1.

Algorithm 1 L-System implementation summary (simplified)

```
1:  procedure STEP
2:      new_word ← ∅
3:      for all modules in word do
4:          i ← position of module in word
5:          S ← set of production rules on module
6:          L ← ∅
7:          for all p in S do
8:              if p.context matches at i and p.conditions evaluates to true at i then
9:                  Add p to L
10:         if L is empty then
11:             Append successor in the identity rule to new_word
12:         else
13:             p ← RussianRoulette(L)
14:             Append all successors in p to new_word and update parameters in new_word.
15:     word ← new_word
```

In addition to the pseudo code above, the implementation is capable of stepping L-systems with branching production rules. Here are two example production rules from the implementation:

$$P_1 : A(\alpha) < A(\beta) : \alpha^2 < 10 \xrightarrow{4} A(\alpha^\beta)A(5)$$

$$P_2 : B(\alpha, \beta) < A(\omega) : \alpha == 3 \xrightarrow{5} B(\alpha, 5)[A(3\omega)]$$

The probability is a real value, called priority, instead of a percentage. The use of prioritization removes summing-to-one issues of production rules with the same predecessor. Instead, any real number can be specified. The percentage is constructed by normalizing priorities into $(0, 1]$. The normalization happens in the Russian Roulette algorithm which randomly selects from the set of legal production rules. If, in the example above, both production rules apply, one is chosen as such: $P_{select}(P_1) = \frac{4}{4+5} = 0.\overline{4}$ and $P_{select}(P_2) = \frac{5}{4+5} = 0.\overline{5}$ If neither production rule applies, the identity rule is used.

Conditions and update rules were implemented with C++11 lambdas. An interesting effect of this is that conditions or update rules can depend on anything. For example, a production rule can be legal only if precipitation exceeds some threshold. See Code 1 in Appendix C for implementation details.

In Algorithm 1 the L-system computes the next word in a sequential manner as opposed to parallel (see a discussion on sequential stepping versus parallel stepping in Section 4.1.2).

3.7.2 Creating the L-system interpreter: a 3D turtle

After an L-system has generated a string representing a plant, it is something called a *turtle* which translates this into a 3D data structure. Lindenmayer introduces turtling as a way to create a sophisticated geometrical representation for L-systems [2]. There are no other prominent strategies to translate complex 3D L-system data to renderable data. There are several simpler geometric representations, which Lindenmayer explains in his work, including shorter or longer rectangles and methods which render fractals[2]. Foremost, this section describes the turtle

implemented in this project, including what type of data it handles. It also explains how the implementation differs from conventional turtling methods.

Firstly, the turtle has one dedicated type of input and one output. The input to the turtle is a string from an L-system containing information about the plant. For example, the input could be: *"F(5)&(24.0f)#(0,100,0)*". This text describes the plant as a whole and the above example translates to: "*Go forward five steps. Turn right by 24 degrees. Change color to dark green.*". The turtle exists in 3D space, and orientates itself in three axes; up, left and heading. Some commands turn the turtle in one of these directions by a specific angle. Here is the full list of commands which the turtle implements:

- +(δ): Turn left by angle δ, using the turtle's up-orientation RU(δ).
- -(δ): Turn right by angle δ, using the turtle's up-orientation RU(δ).
- &(δ): Pitch down by angle δ,using the turtle's left-orientation RL(δ).
- ^(δ): Pitch up by angle δ, using the turtle's left-orientation RL(δ).
- \(δ): Roll left by angle δ, using the turtle's heading-orientation RH(δ).
- /(δ): Roll right by angle -δ, using the turtle's heading-orientation RH(δ).
- #(r, g, b) Change the current status' color to the values of r,g,b in RGB format.
- <(δ): Thicken the breadth in the present state by value δ.
- >(δ): Narrow the breadth in the current state by value δ.
- L($resolution, style, r, g, b, displacement\_variance$): Save parameters on module L and turtle status in a leaf status. Insert that leaf status into array A.
- '[': Save the current state onto the stack
- ']': Pop the stack onto the current status

These commands correspond to those found in the Algorithmic Beauty of Plants; the book that first introduced L-systems[2], with the exception of L. See Section 3.7 for an explanation as to why L was added.

Furthermore, the project's implementation differs in that it does not draw the L-system at the same time as it generates the data. The turtle does not render the plants; it produces the data. Otherwise, each word would have to be traversed, interpreted and rendered in real-time.

Next, the output of the turtle is a set of control points in 3D space, containing information about its contour, color, width, and heading. These features are analogous to the general cylinder graphical data model Prusinkiewicz et al. present in their paper[29]. However, while they also use textures, that is out of this project's scope. More about the motivation for using generalized cylinders is described in the next section.

Finally, regarding the implementation, the turtle translates the word in order. It spans from the first to the last symbol. The project did not parallelize the translation. However, the process of generating plants was parallelized, since it is both easier to implement and likely yields higher performance gains. What differs in the project's turtle implementation is that while it traverses the word, it also saves a tree of nodes corresponding to the control points As such, the output

it yields is the root node of the tree. The node structure was chosen because it reflects the branching structure in plants. It is also convenient to traverse the structure using recursion, as it fits the generalized cylinder method.

3.7.3 Visualization of the plant species

The visualization of the plants concerns both the L-systems and the Turtle. This section describes the means by which the node, or control point tree from the Turtle becomes ready for Unreal Engine. While this does not result in a visible tree, the data is saved and can be rendered at a suitable time. This section describes and motivates the visualization method. It also shows the in- and output of the visualization module; namely the *triangulator*. The data is turned into a mesh and rendered in later stages, outside of this process.

The generalized cylinder method described by Prusinkiewicz et al.[29] is used to generate the tree-rendering data. The idea is to swipe a contour along the curve which can be interpolated between each consecutive control point. One reason this method was chosen was that it is based on parametric turtles, which the project uses. Additionally, the method produces a result with smooth curves, even between branches. It can also be extended to feature more variation in the form of contours. However, the project only had time to implement circular contour, as more complex shapes would, in turn, require a more complex polygon triangulation method. The principal difference from Prusinkiewicz studies is that this project separated the turtle and the triangulation into separate modules.

The input of the system comes from the turtle module while the output is modified to add leaves and later used in the placement module. The input to the triangulator is a tree of control points. The points describe, as mentioned in the previous Section 3.7.2, exactly how the corresponding plant should look. Moreover, the output of the triangulator is all the data which is needed to create meshes to be rendered in Unreal Engine. The data includes all the 3D points, the indices of the triangles the mesh should draw between all the points, and also the color at each point. The triangulator recursively traverses the tree and interpolates between each consecutive control point. Branch recursion allowed the triangulator to separate the branches. The separation was necessary for implementing the wind affecting the plants. It has to do with the wind calculation requiring knowledge of the branch hierarchy.

Another vital aspect of the triangulator is that it takes into account many aspects of the vegetation shape. For example, it takes into account when the width in two control points are not the same, therefore increasing or decreasing the size of each circle segment between them to normalize the generalized cylinder shape. It also takes a sample rate which determines the number of points which it should pick from each circular section. For example, the tree could be defined by a sample rate of four points per circle, resulting in a rectangular branch or trunk. The color is implemented analogously to the width, changing the color of each circle segment until it reaches the end-control-point.

3.7.4 Defining vegetation generators

This section describes the process for generating vegetation. Each plant species defined in the project produces unique sampled plants. A stochastic, parametric, right-context sensitive, bracketed L-system was chosen for generating the branching structure. L-systems are visualized by turtling (see Section 3.7.2), followed by polygon triangulation for use in Unreal Engine (see

Section 3.7.3). However, this chain of processes was encapsulated in a set of generator modules for each species. Every generator holds an L-system, a turtle and a triangulator.

Leaves were implemented by letting a module L exist within the alphabet of the L-system. Whenever the turtle encounters an L, it saves a leaf status in an array A. For each element in A, generate a rock using parameters stored in the current leaf status. These parameters were set to produce green rocks, or as the project puts it, *stylized leaves*. It was both convenient and practical to use the stone generator module for both rocks and leaves.

To boost the performance of the generation, the project used precomputation (see Section 4.2.1 for a discussion as to why this was necessary). A class *treePrecomputer* inserts all generated trees into itself. Before generating a new tree, the algorithm asks the *treePrecomputer* if all trees have been precomputed and if that is the case, return the next precomputed tree. Each precomputed tree is returned in a cyclical order $Tree_1, Tree_2, ..., Tree_{num\_precomputes}, Tree_1, Tree_2, ...$, which ensures that any generated tree may be reused. A brief pseudocode summary of the C++ implementation for a tree generator is presented in Algorithm 2.

Algorithm 2 Tree generation procedure summary

```
1:  procedure GENERATE
2:      if treePrecomputer.done() then return treePrecomputer.next()
3:      l-system ← ω, V, P
4:      for 1..steps do
5:          Step the L-system
6:      turtle ← Initial State
7:      turtle reads the word on l-system
8:      3D-model ← triangulation of turtle
9:      for all leaf_status ∈ A do
10:         leaf_3D_model ← stone_generator.generate(leaf_status)
11:         Offset leaf_3D_model in its modelspace by p
12:         Merge leaf_3D_model into 3D-model
13:     treePrecomputer.add(3D-model)
14:     return 3D-model
```

While Algorithm 2 is suited for vegetation with branching structures, it is not suited for grass due to superfluous complexity. Grass is not procedurally generated as opposed to trees and bushes, instead a class *GrassGenerator* simply returns a hard-coded 3D-model which is rotated when placed in the world.

3.8 Rocks

Rocks appear all over nature, in different forms or shapes. This section aims to illustrate the rock generation in Seamscape and why the used method was preferred. There are several methods for generating rocks, such as using SpeedRock where rocks are generated using an evolutionary algorithm and L-systems [30]. However, this project aimed for simpler shapes. Therefore, a simple algorithm for generating rocks was invented. The algorithm builds upon the idea of approximating rocks as the Convex hull of a set of uniformly sampled points from the hull of an ellipsoid.

The rock generation requires the following: a resolution N, a color, the displacement variance σ

which determines the sharpness, and finally the ellipsoid parameters a, b and c which determine the shape of the rock. Rocks are then generated as such:

1. Uniformly sample N points from an ellipsoid hull, parametrized by user provided a, b, c
2. For each sampled point, perturb the point by adding a sample point from the Spherical Gaussian distribution, parametrized by σ
3. Find the Convex hull of the sampled points. This Convex hull is the mesh of the rock, where each vertex in the mesh has the color of the user provided color.

Step 1 and 2 are linear in time complexity. The Convex hull algorithm in step 3 is has a worst case complexity of $\mathcal{O}(n^2)$ [31] where n is number of points. As such this algorithm has a worst case complexity of $\mathcal{O}(n^2)$ where n is the number of points (the resolution parameter). For achieving the low-polygon style in this project, a resolution of 20 sufficed.

The order of step 2 and 3 is crucial. If their order were reversed, the following unwanted effects would take place:

- The last rock would always have N corners
- The clockwise ordering of the vertices can be swapped on the camera such that the face becomes invisible due to backface-culling

The TetGen library is a library providing the Convex hull algorithm for a set of points in $\mathbb{R}^3$ [31], was used for the Convex hull problem in step 3. Other libraries for the Convex hull problem in $\mathbb{R}^3$ exists such as CGAL or *qhull*. CGAL was ruled out due to its numerous dependencies [32]. Likewise, qhull was disregarded because it had no clear advantages over TetGen in this project and risked requiring more time to implement. The Wolfram Language uses TetGen for various operations [33] which also favored TetGen.

Finally, the generated rocks were precomputed. The generator checks whether the input matches the parameters used during precomputation of rocks. If so, it returns a precomputed rocks in cyclic order.

3.9 Wind

Applying wind concerns the transforming vertices in, for example, plants. By transforming for example leaves, they appear to move in the wind. This section shows how the project implemented wind, and why the used method was chosen.

Several approaches to simulating wind exist. The project tried an Unreal four blueprint material which adds a wind effect to 3D models representing vegetation[34]. It stored trunks, branches, and leaves in the same 3D model. As such, the blueprint material was applied to the trunk, in addition to branches and leaves. Wind affecting the trunk produced undesirable results.

Another method presented in [35] utilizes modal analysis to achieve realtime wind effects. This method was ruled out due to its complexity. However, a method presented by Ota et al. [36] observed that 1/fb noise can approximate the wind arbitrarily well. This method was chosen in favor of others due to its seemingly straightforward implementation. Furthermore, the method

presented by Ota et al. was referenced in other papers, including *Wind projection basis for real-time animation of trees* [35].

Ota et al. linearly interpolate values from 1/fb noise to approximate wind force irregularities at time t. The force approximates phenomena such as turbulence, meaning short-term wind anomalies. However, long-term irregularities are not covered by this method. The wind strength is processed into a deflection force on each branch of a tree. For the deflection, a cantilever spring model is used. It approximates branches as beams to find a deflection vector.

Shin Ota et al.'s method can describe how branches, leaves and accentuated motions are simulated [36]. However, this project is only concerned with motions deflecting onto trunks and branches. The project limits itself to applying the wind to procedurally generated vegetation. For example, wind manipulation of water would require water physics as a prerequisite.

The formula for obtaining rotation per branch at time t for coordinate p, in one dimension, is:

$$\Theta(t,p) = arcsin\left(\frac{F(t,p) + P_B N(t)}{\frac{Ebt^3}{4l^3}}\right)/L$$

Note that in the original method presented by Shin Ota et al., the p parameter is omitted.

When the rotation has been calculated per branch, they are combined from child to parent. From the thin branches inwards toward the trunk. If the parent branch is rotated, all children should rotate first, according to Ota et al. [36]. Then the vertices must be transformed. The idea from this point was to transform each vertex in the ith tree as a function of T_i and T_i' using an Unreal blueprint material. T' is the tree of control points under the pressure of wind. However, this was never implemented, see Discussion 4.1.4.

Finally, in Ota et al.'s method, a 2D vector field modeled the wind field[36]. The components of the vectors were generated using a low-frequency Perlin noise. Then, to create the length of the vectors a second Perlin noise with higher frequency was used. Afterward, the vector field is displaced using a simulation of time progression. It can in theory progress in time because the wind function is periodic. However, this requires continuity between the edges. The continuity problem was not resolved.

Because the wind was cut from project no ambitious wind field generation was implemented. One way to generate a plausible wind field is by using curl noise, as in this paper from University of British Columbia [37].

3.10 Distribution of vegetation and stones

Distributing vegetation in a virtual world can be both simple and complicated depending on the method chosen. This section describes how vegetation is placed, and why the used method was preferred.

Firstly, the most basic way of positioning plants, apart from doing it manually, is to put them in an even grid and then using a random displacement algorithm. Simultaneously, this method does not consider species' preferences of inclination, water supply, or altitude.

However, there are two techniques for realistic placement of vegetation on an existing heightmap. One is a by comparison computationally heavy simulation, first described by Duessen et al.

[9]. The other used computational geometry to decide the vegetation distribution [9] and was designed by Hammes [4].

Because runtime generation was one of the first priorities of the project, it chose a slightly modified version of Hammes's method; *Modeling of ecosystems as a data source for realtime terrain rendering* [4]. Apart from considering species' preferences in different ecosystems, Hammes's technique is also tailored to consider the level of detail [4]. The consideration of the level of detail was suitable for the project and therefore chosen above Duessen et al.'s method. Also, rocks could be distributed analogously to plants using this method, with alterations to the placement preference.

The level of detail for each object is bound to its corresponding terrain tile. Each tile is split into a grid whose number of squares depend on the level of detail, as seen in Figure 6. An ecosystem is chosen depending on the heightmap for each area within the grid. While Hammes iterated the detail levels recursively for each area, until the correct level was reached [4], the project only iterates once for the requested level of detail. The optimization works by the assumption that all previous levels of detail have been requested in a previous state. Therefore, only the tiles which are switching level of detail are recalculated after moving the camera in the landscape.

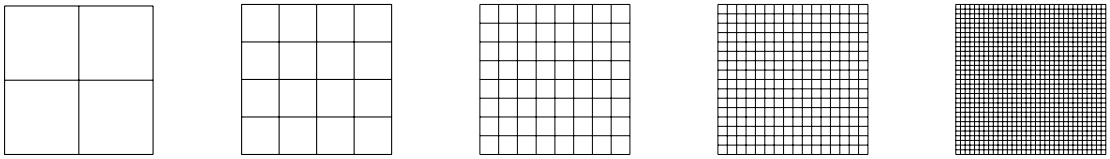


Figure 6: Grid division of world-tiles based on detail level.

Furthermore, what ecosystems thrive in different areas of the terrain is determined by their surroundings, as modeled by a set of parameters and probability distributions based on them. Four variables are used, which are defined by Hammes [4]: height, relative height, slope, and skew. The input for the ecosystems in Seamscape is a variation Hammes preset values. More coherent ecosystems were yielded by slightly modifying the parameters. In Table 1 (Appendix D) the different ecosystems and its variables used are shown.

Some complexity cutdowns and alterations to fit the project were done to the algorithm proposed by Hammes (the performance evaluation can be found in Section 4.2.1). The height is not from the center of the tile but in one of the corners. Also, because the tiles are independent, their edges can also be a seam or edge of the world. Some calculations require nearby tiles, and therefore instead re-use the current tile's height value instead of the nearby one. However, this slightly implicated the placement sometimes to appear off.

Unlike Hammes's implementation, this project did not use any ecosystem layers. The project goals state the use of at least one tree and at least one bush. Therefore, adding more ecosystem layers would be superfluous. Since the ecosystem variables must apply to all levels of detail, parts of the calculation had to be normalized. The calculation of relative elevation, slope and skew were normalized depending on the size of the tile. Furthermore, the ecosystem parameters were modified after normalizing the parameters, because Hammes's predefined values only applied to the lowest level of detail. The approach was taken to yield a vegetation placement which is tailored to the project's ecosystems.

Finally, Figure 7 visualizes the species preferences in varying elevation and moisture. For exam-

ple, light green trees stay close to the water while cherry trees are mostly exhibited in higher altitudes. Slopes are mainly occupied by stones in the example, but can be exchanged for bushes or other vegetation.

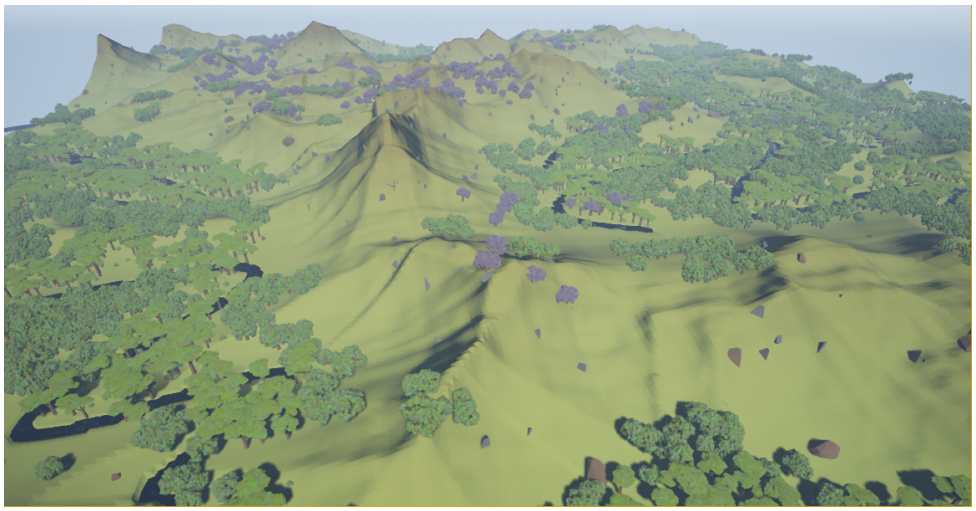


Figure 7: An example of an ecosystem in Seamscape.

Each ecosystem has a set of plants, and each plant is defined with the following variables:

1. Name - used for identification,
2. Level of detail - which level of detail it appears in (needed to compensate the missing ecosystem layers),
3. Density - representing how many specimen in each tile (and the tile size depends on the level of detail)
4. Minimum and maximum scale of the plant.
5. Generator - which will produce all the graphical information.

When an ecosystem is chosen for an area, the flora belonging to that ecosystem is distributed. Depending on the density, a noise function and a random offset the number of specimen are decided. They are then given a completely random position and rotation within the area. Their scale is also random but still held within the defined limits. All random values is determined using the hash function, having the seed, the level of detail, and the position as parameters to the function. This enables the values to be re-generated during a later stage.

3.11 Coloring the landscape

Before ending the Method chapter, some light is here shed on the use of colors in the project. This section motivates and summarizes the method used. Because coloring is a topic for designers, the project chose to use a color palette for the world from another game. The color palette is based on the forest ecosystems in the Witness [8], which feature vivid colors in a low-polygon world.

The leaves on trees have unique color as seen in Figure 8. See Section A in appendix for how the coloring per leaf was accomplished. Within the terrain, the palette is defined as an Unreal-specific *material*.

Figure 8: A tree from Seamscape illustrating coloring per leaf.

3.12 Water surfaces

An Unreal material are used to render the water surfaces. This is because Unreal has tools which simplifies the implementation of wave animations and light reflections. The reflection of the water depends on what angle the camera observes the water surface. Two textures from Unreal Engine's content library was used to displace the material. The textures are displaced by time dependent sinusoidal functions which results in wave animations. See Figure 15 i section 4.1.6 for the resulting water in Seamscape.

3.13 Day-night cycles

Another addition to Seamscape is its choice of using a day-and-night cycle system. This section describes the method. Firstly, it was implemented by using the tools in the Unreal engine using a blueprint class. Blueprints are a graphical programming language in Unreal Engine. This approach was chosen because it was an already existing feature of Unreal Engine, and therefore required minimal time.

By utilizing the features in Unreal Engine, the blueprint class updates the sun position, light intensity and the color of the sky depending on the sun angle. By changing the input variable t_{min}, the user can modify the duration of one-day night cycle. To calculate the speed of which the sun angle, θ_{sun} changes, The relation between time units in the day-night-system and realtime were determined. By taking an input variable t_{min} representing the period of a day and night

cycle, the following relation is derived:

$$\boxed{\theta_{sun} = \frac{24 \cdot t}{60 \cdot t_{min}} \cdot 15^{\circ}}$$

t are the number of elapsed seconds in realtime.

3.14 Seamscape's interface

The project aspired to have a simple interface toward the landscape generator. The interface was created using Unreal Editor's widget design editor. Unreal's interface designer was used because the game engine itself was chosen as an asset to the project.

In the Unreal Engine, events are created when for example buttons are clicked. These events can be bound to functions or keyboard interactions. Unreal's blueprint editor was used to construct the interaction sequence. It is a tool for visual coding. With drag and drop functionality and much more, one can drag in, connect functions, and create variables.

4 Result and discussion

Now that the method has been presented, the time has come to compare the results to the set goals. This chapter interleaves between the presentation of results and discussions of each element in the procedurally generated world. It discusses Seamscape's features, and how they correspond to the goals. Additionally, it analyzes the realtime performance of the software, which was also part of the goal specification. Figure 9 exemplifies Seamscape's features. Finally, it discusses the future of the product.

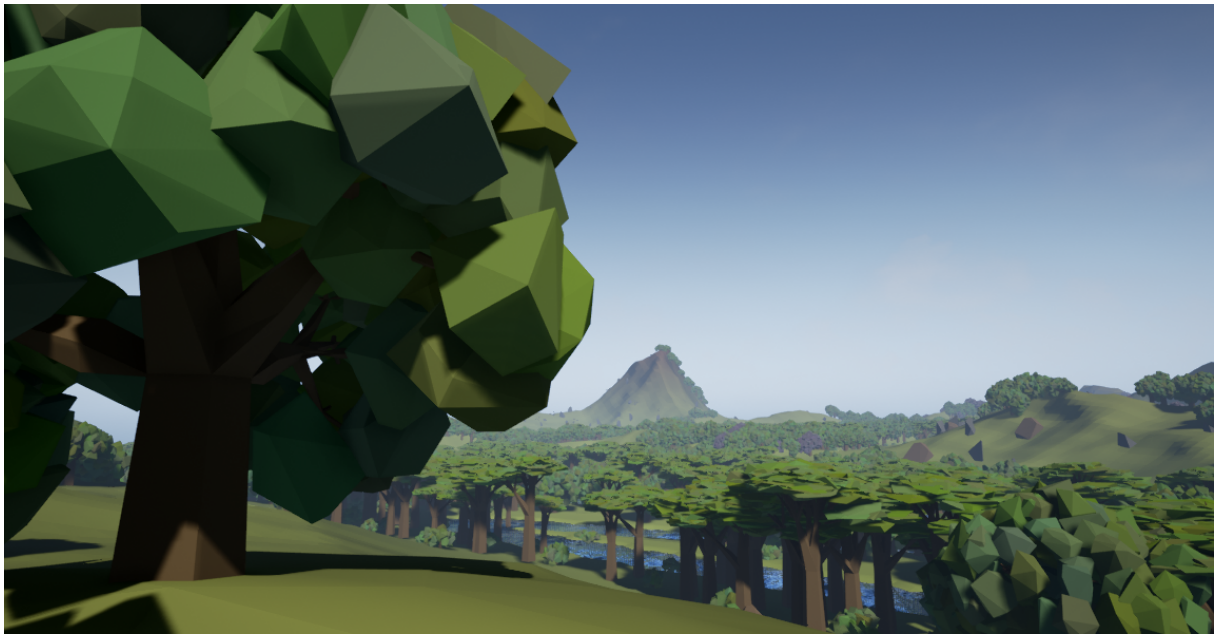


Figure 9: A procedurally generated stylized landscape.

Figure 9 illustrates the core features in the project. There are mountains produced by noise functions and trees from L-systems. The stones and river networks are generated by using our own techniques. Shaders were constructed using Unreal Engines material blueprint to achieve the lighting effects.

4.1 Features versus goals

Here the features of the project analyzed, to determine whether they fulfill the project goals. It includes goals surrounding; the graphical user interface, the vegetation, rocks, placement, water, and the wind.

4.1.1 The graphical user interface

At the beginning of the project, there was a primary focus on modifying the landscape through an editor. However, the mutability dropped to a secondary goal, because of the time it took to implement the landscape generator itself. The result is a graphical user interface (GUI) limited to changing the size, seed, and regenerating the terrain as shown in Figure 10 below:

Figure 10: Seamscape's graphical user interface.

The user of Seamscape was to be able to generate different ecosystems based on interactions with the GUI. This was not achieved. Simultaneously, the *editor* of the environment had a preliminary specified design which circulated the idea of the user combining and modifying sets of biomes. It considered how the user should be given a set of variables regarding mood; the weather conditions, time of day and sharpness of mountains. There were going to be more biomes, for example, desert, marsh, and hills. However, because the project did not have the time to add the final touches needed for different biomes, the design had to be scrapped. Thus, the goals regarding the GUI are considered failed. An editor which gives the user the freedom to design more versatile scenes would be the first addition to the project in future development.

4.1.2 Vegetation

In this section, the results regarding vegetation are shown and discussed. An excerpt of problems encountered with the generation of vegetation is addressed as well. Finally, points for future improvement are presented.

The result was made possible by an L-system framework combining bracketed-, stochastic-, parametric-, and left-context-sensitive production rules. However, no right-context matching was implemented for the L-systems. A major problem regarding this was how to order the branches. If the order of branches in a word does not matter, the production rules $A > B \rightarrow A$ matches $A[B][C]$ and $A[C][B]$. However, the latter does not match if order matters. No theoretical support for either alternative was found. Hence, support for right-context was omitted.

Through the L-systems, Seamscape is capable of generating distinct trees and bushes of different species. This goal was achieved as illustrated in Figure 9. It also meets the real-time generation requirements. Furthermore, the environment features grass, as shown in Figure 11.

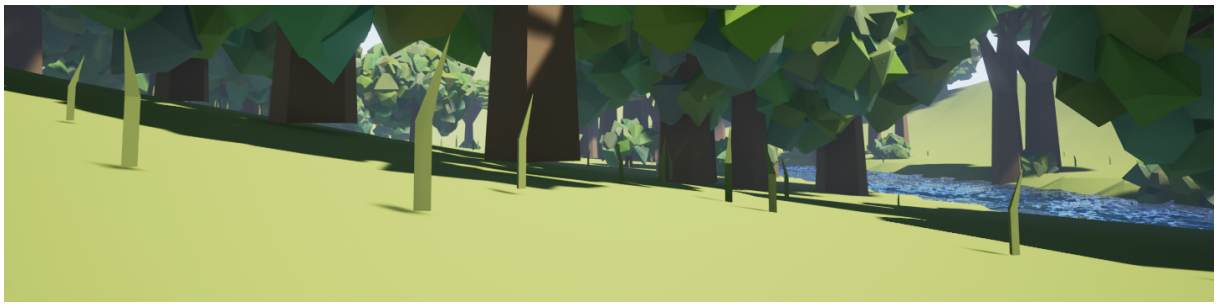


Figure 11: Grass in Seamscape.

The grass generation focused on speed of implementation. Therefore, the project pre-defined a 3D-model, as opposed to generating a 3D-model. Due to time-constraints, no other methods for making grass was considered. Also, only small amounts of grass could be distributed, as they were performance-heavy. Figure 11 also shows the amount of grass which can be displayed without implicating real-time performance. The lacking performance was partially due to the use of a volumetric model. Also, it was because no optimization was made. Examples of potentially more performance-effective methods are procedurally creating billboards or 3D models.

An optimization which could increase the performance of grass, and also the rest of the plants, regards the level of detail. Detail levels were implemented for terrain, but not for vegetation. Maximizing rendering performance of generated plants, grass, and also rocks was down-prioritized and ultimately not implemented.

Another optimization for Seamscape could be to parallelise the generation of L-system words. It is currently not possible because it would introduce race conditions in the production algorithm. This is illustrated in the appendix, in Algorithm 1 on lines 11 and 14. Parallelisation would speed up the generation of words and, consequently, the production of vegetation. A possible solution to the race conditions is to store words in a tree data structure. Rewriting nodes is a parallel operation. Hence, it eliminates any data races.

There is another benefit of storing words in a tree data structure. Namely, left-context checks can be done by traversing the direction of the root. There is, however, a complication to using a tree data structure. Particularly, a certain ambiguity in the interpretation of yielded words. Assume A is a root with children B and C. Then, the topology of the word $A[B][[C]]$ is undefined, since C has no parent. Therefore, the project could not implement parallel evaluation of modules. Nor could it achieve efficient left-context matching.

Another object of improvement concerns the leafage. There are performance implications on the trees which arise from the utilization of the rock generator. The rock generator was already implemented when it was chosen to be used as leaves. Also, low-polygon rock shapes are simplistic and follow the intended style of the project as foliage. Despite this, performance became an issue due to the amount of leaves generated (see Section 4.2.1). At the same time, the generation was optimized such that is did not pose a problem. For future development, however, it may be of interest to use other generators for leafage.

Apart from performance improvements, there is space for improvement regarding the diversity of species. Fundamentally, the project is more readily applicable to 3d environment development

if it can produce diverse landscapes. For example, by creating more species. One limiting factor to the project creating species was the L-system code size. Figure 9 were made by the L-system given in Appendix A, in 120 lines of code. At the same time, the project goal states that no more than one species is necessary. The project focused on performance regarding trees, rather than defining more species. There is, however, another problem regarding creating species. L-systems are made to capture phenomena in biological processes of real plants. Therefore, the programmer has to depict how each species grows.

A possible solution to include more species is to parse L-systems defined externally from a text file. It can also be done by introducing a sketch-based interface. For example, Anastacio et al. developed a method for sketching plants which control rule-based models [38]. Another way to produce variation is to use morphological shape design. Sanchez et al. provide a method to morph chairs genetically pairwise, creating new designs [39]. Similar methods could perhaps be applied to plants based on L-systems. None of these methods were implemented, as it was out of scope. However, being able to create more variation it is considered a central feature of future development.

4.1.3 Rocks

Here the rocks and the rock generator of Seamscape are presented and discussed. The most notable feature of the rocks lies in the rock generator's versatility. Namely, the generated rocks proved useful in several parts of the project, including tree generation. Therefore, it aided in reaching the goals discussed for the vegetation, apart from adding rocks to the environment.

The stones which are present in Seamscape can be generated using any color. They are generated from a point cloud with n points. Furthermore, the shape of the ellipsoid models of the rocks can be scaled in three ways; compact, tall or neither. A generated stone is shown in Figure 12.

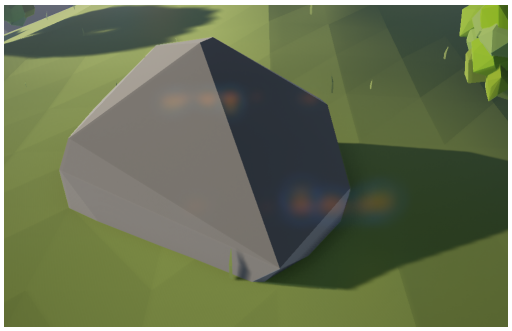


Figure 12: A rock in Seamscape generated with normal settings.

Performance considerations were also made for the rocks. Particularly, they were subject to precomputation. With precomputation, the generator stores n unique stones. Because of this, the performance increases at the expense of uniqueness. A drawback of the rock generation method is that generated rocks are convex.

A final concern of the rock generation is the possible future features. The simplicity of the rock generator makes it appropriate for use for both rocks and leaves. However, it cannot be used for more complex shapes. A simple improvement for the future would be to produce two rocks in the same place. The dual generation approach gives rise to non-convex shapes.

4.1.4 Wind

This section mainly discusses why the wind could not be part of the result. There were goals regarding exhibiting wind in the world because it adds to the immersion. Immersion is a vital consideration for game environments. Additionally, a goal of the project was to investigate how to apply the wind to procedurally generated vegetation.

Despite it not showing in the result, the method described in Ota et al.'s paper [36] was implemented. It was specifically tailored for use with control points produced by the project's turtle such that wind force could be calculated for any generated tree. The project finished much of the wind implementation, but not all. For example, the function $\Theta(t, p)$ has been implemented as have the function integrating $\Theta(t, p)$.

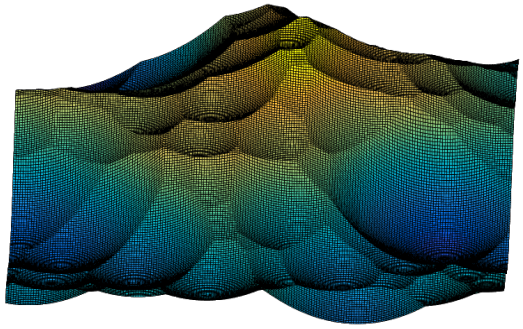
There were several problems which hindered the wind feature's integration into the final product. Firstly, the emerging difficulty of transforming a set of vertices without producing disjoint results. The wind generation method described in Ota et al.'s work [36] was more complicated than, for example, using only linear algebra. Deformation of the meshes was also time-consuming to develop. If the project had chosen to use a simpler model for wind, it might have finished in time.

Finally, the project prioritized plant generation. For the project to have vegetation in the final product, the wind development had to be canceled. Minor progress was made on defining pseudocode for the transformation of vertices. The canceled wind implementation spanned over 600 lines of source code. At the same time, the efforts were not completely in vain. A tree data structure used for wind calculations was namely used in the turtle.

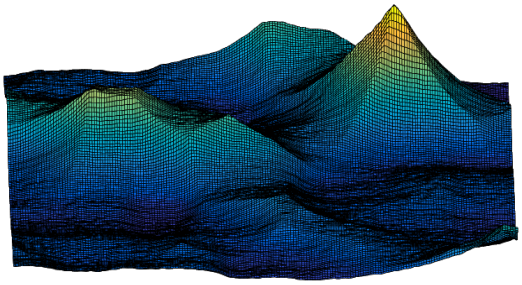
4.1.5 Terrain

There were two goals set for the terrain, its coherence and the presence of more than one type of elevation map. A complete scene showing terrain can be seen in Figure 9. This section describes the resulting terrain and also proposes improvements for future development.

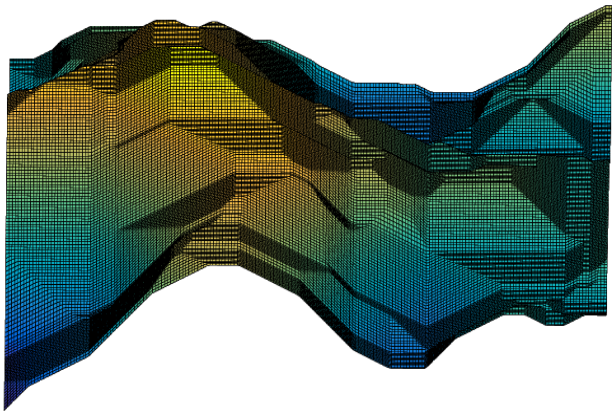
There are several heightmaps apart from the demonstrated one in Figure 9. For example, Figure 13 shows several styles obtained by combing layers of Perlin and Worley noise. To sum them up, Figure 13a consists of Worley noise using an average of the four smallest distances. Figure 13b uses five layers of Perlin and Worley noise by multiplication of the two smallest distances. Also, both 13a and 13b uses the Euclidean distance metric. Figure 13c uses Worley noise by taking the average of the four smallest distances and uses the Manhattan distance metric.



(a) Stylized heightmap



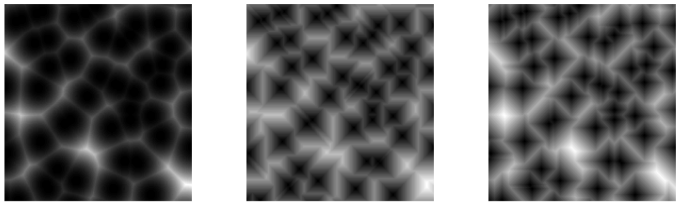
(b) Heightmap with both smooth and sharp features



(c) Heightmap with flat plateaus

Figure 13: Heightmaps generated with different noise combinations.

Using different metrics allowed further customization of the heightmaps. The use of Chebyshev metric for the Worley noise yields a different result than the Euclidean, but similar to the Manhattan metric, see Figure 14.



(a) Euclidean (b) Chebyshev (c) Manhattan

Figure 14: Worley noise with different distance metrics.

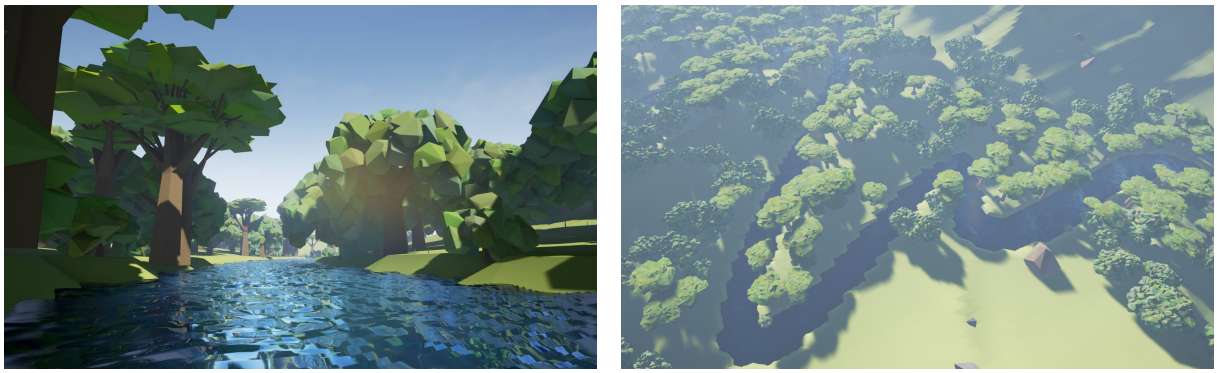
A problem with Seamscape's terrain maps is that they are somewhat repetitive in nature. As mentioned in the theory section 2.1.1, Perlin noise tends to have a periodic behavior. Therefore, terrain using Perlin easily becomes monotonous. However, in the project, the river generation algorithm and the Worley noise compensates to some degree. Despite the compensation, further improvements could be made. For example, it can be achieved using different fractal noises for separate regions. Although, this variation would give rise to another problem; how to obtain a smooth transition between the fractal noises. Currently, each noise layer for each sub-region are generated independently and are summed together into a fractal noise. A continuous transition function which describes the weight of each fractal noise could potentially solve this problem. The problem then lies in creating a suitable transitional function.

Another improvement for the future could be to add cavities to the terrain. For example, caves, arches, and overhangs. It could potentially be done by extending the noise functions into three dimensions. However, this introduces new problems. For example, how to interpret a 3D noise map. A 3D noise map cannot be directly translated into a heightmap. Also, the problem of combining the river generation algorithm and object placement with 3D noise map. A possible solution could be first to generate the heightmap using 2D noise. Then an algorithm could be applied which generates arches and caves onto it.

A final thought on the terrain concerns the coloring. Currently, the coloring of the terrain is done using a single Unreal material. The material changes hue depending on the slope and height. However, the color transitions only fit some part of the terrain. In Seamscape, this was solved by using similar hues of brown, as can be seen in Figure 9. By using materials depending on the terrain region, the coloring could be adapted more carefully for each region. However, if several materials are used, the transitioning between them may pose an issue.

4.1.6 Water

One goal of the project was to generate bodies of water such as rivers, streams, seas, and lakes. Out of these, only water networks with rivers and streams were implemented. The rivers were procedurally generated and allowed control of behavior through parameters. An example of a river network can be seen in Figure 15 both as a close-up and from above. This section describes what hindered the completion of water bodies. It also examines potential future improvements to the water network. Finally, it discusses the possible methods for developing water bodies in the future of Seamscape.



(a) Water network close-up (b) Water network from above

Figure 15: Seamscape's finalized water networks.

Initially, the project planned to create coastlines and islands. However, it was left out. The cause is tied to the lack of terrain gradients in the editor. For example, it would be possible to create island biomes by joining mountainous terrain with water. They would allow the combining of terrain features. The gradient as a design choice would be integrated into the editor interface. However, it was never finished. The coastlines and island were dependent upon the gradients' completion.

Furthermore, water physics had to be left out of the current version of Seamscape. A prolongation of the project could be to include it, which would allow for arbitrary slopes and the presence of waterfalls. Simultaneously, the current water networks are already compatible with water physics. Namely, the networks are hollow, meaning they can be filled with physically accurate water if it had been implemented.

Another goal regarding water was to create inland water bodies such as lakes and ponds. However, the project had to down-prioritize water bodies to ensure the existing water networks worked correctly. Simultaneously, there were two ideas in this category which were considered but not implemented.

The first idea for creating water bodies was to use fractal noise. This technique would be compatible with the terrain model since it also uses noise. However, the new water body model would have to be integrated into the existing water network.

The second technique for producing water bodies involved adapting the currently used network model. It could be possible to extend the river network model to include other water bodies. For example, a selection of curves from the network, as seen in Figure 15b, could have been converted to larger pools.

Additionally, there are possibilities for improvement on the river network model. The main source for improvements lies in the network generation. Currently, all inner points of a curve are tested against being the closest point. Possibly, the use of data structures would decrease the average number of required tests.

A final improvement would be to combine two generation stages; the *value association* stage and the network generation stage. Combining the two would diminish the problem of curves on different heights being placed unnaturally close.

4.1.7 Distribution of environmental objects

This section discusses the goals for logical placement in the world, and world coherence. One objective of the project was that when the camera returns to an area in the world, it should be identical to how it looked previously.

The coherence is currently true for the position, scale, and rotation of the object. However, individual objects will not look the same. The incoherence of individual objects can be attributed to the L-system, which produces items stochastically. The generation of rocks and vegetation uses a C++ standard library function to generate random-values. The values currently used are not initialized using a seed. At the start of the generation process, a set of objects is produced for placement. They are not directly connected to a position or anything that can be used as a seed later on during re-generation. Therefore, modifications to the process structure would be required to solve this issue.

An additional improvement concerning coherence and logical placement is to ensure plants are

rooted at ground level. The improvement involves placing objects at the correct position of the z-axis. Figure 16 illustrates the issue. Currently, height values from the world-tiles are extracted from each corner, and as described in the method, some approximation were made. However, if this were the only issue, the height difference would not be as visible as it currently is. More specifically, each position is determined on a low level of detail, and thus the height values are equally inaccurate. The inaccuracy can be corrected by adjusting the height position after the terrain increases the level of detail. However, this improvement could not be implemented in time.

Figure 16: A floating tree in Seamscape.

A final possible improvement would be to ensure objects are not placed inside each other. This would be possible by, before placing a object, searching for all previously placed objects in the world within the same area. However, without deriving a sophisticated approach, this would slow down the distribution process. At the same time, objects rarely ended up placed inside another. Refinement of the placement could, therefore, be down-prioritized.

4.2 Performance

The project aimed to generate environments in realtime. Therefore, it is vital to analyze the software performance. In general, the generation time until the landscape is completed varies depending on the size of the terrain. It naturally follows that it is complex to describe the performance of the software. This section discusses the performance of the program and attempts to point out eventual bottlenecks.

4.2.1 Landscape generation bottlenecks

Here follows a discussion the performance of the landscape generation. A vital aspect of the performance is the bottlenecks in the generation phase. The most visible bottleneck in Seamscape is the plants. Mainly, it is due to the exponentially growing complexity of the plant models in each generation step. It can grow past the threshold which bars run-time generation. However, the plant performance is closely tied to the rock generation. The plants namely use rocks as leaves.

In the old version of Seamscape, trees were generated with step-size six. Using that size, trees such as the one in Figure 17 were generated. They took approximately 90 ms to generate, as did variations within the same species. A tick in the process loop can at most take $1\overline{6}$ ms when aiming to run at 60FPS. Therefore, the 90 ms generation time per tree had to be reduced.

Figure 17: A tree of step-size six. Drawn in an old version of Seamscape.

Several actions were taken to diminish the performance issues regarding plants. One substantial bottleneck for plant production lay in the leaves. Before, it was possible to produce approximately 90 leaves per second in real-time. However, this is far too little for generating forests. Therefore, precomputation of leaves was introduced. The precomputation involved caching a set of leaves using a fixed set of parameters. By precomputation, the average time spent generating a single leaf was decreased by 11%. See Table 4 and Table 5 (Appendix D) for raw data. For the plants of step-size six, the production time was lowered from 90 ms to 60 ms. The reduction was a distinct improvement. However, the precomputation alone did not reduce the time enough.

The was another principal bottleneck for the plants apart from the leaves, namely the step-size. Therefore, the step-size of the L-systems had to be lowered. The L-system stepped six times to generate trees like in Figure 17. For each leaf, 40 vertices were extracted from a point cloud.

In the current version of Seamscape, and as can be seen in Figure 18, the L-system modeling the trees use step size five. For each leaf in the figure, there are 20 vertices in a point cloud.

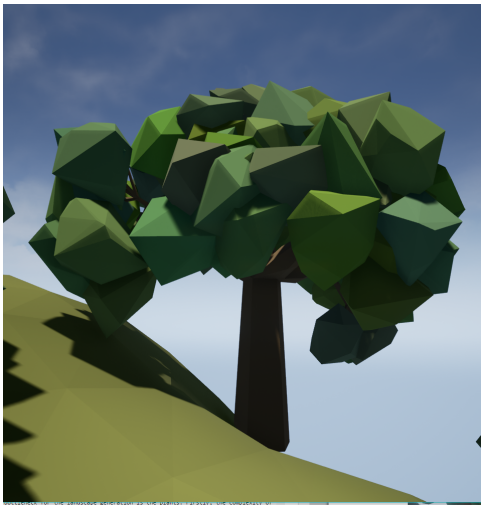


Figure 18: A tree of step-size five.

There was a particular reason why lowering the step-size gave comprehensive results. Namely, the number of leaves on the trees declined exponentially. To further reduce the computation

time, the resolution per leaf was reduced from 40 to 20. These two changes had a fruitful impact on performance. Time spent generating a tree dropped from 60 ms to 3 ms. Figure 18 displays a tree of step-size five.

Even though the step-reduction led to performance gains, it had negative side-effects. For example, branching structures vary less using step-size five compared to six, as in Figure 17. Also, leaves in Figure 18 are more acute than in Figure 17. While the trees in Figure 17 are visually preferable, ultimately the trees from Figure 18 were chosen. Using step-size six prevented real-time performance. For further detail, performance tests surrounding the plants are presented in Appendix D Table 6.

While both the step-size and rock-generation was optimized, still no more than five trees could be generated per frame. See Appendix D Table 6 for more details. Therefore, precomputation of trees was introduced. The gains in performance came at an expense; only 40 unique trees were allowed appear in the environment. The project conjectures that it is hardly noticeable that all plants are not unique. Especially when each plant has a different orientation. Appendix D Table 7 shows the performance of the tree generation after precomputation was introduced.

After the mentioned bottlenecks were addressed, an average of 793 trees could be produced within 10 ms. The average time spent generating a single tree decreased by 95.12%. These results meet the real-time criteria.

Finally, a possible bottleneck, namely the placement of objects was examined. However, the placement took on average 19.56% of the total time. In general, about 0.017452 s was spent on generating objects. See Table 2 and 3 in Appendix D for further information. This data shows that the distribution is not a current bottleneck. Therefore, little effort has been made to optimize this algorithm.

4.2.2 Memory use implications

The landscape generator can produce terrains of different sizes. It so happens that a limiting factor of the landscape size could be computer memory. Therefore, the memory usage of the generator was measured. Measured memory usage with and without vegetation were 4078 MB and 12 MB respectively, and as such memory usage in the terrain is not a memory issue. The raw data can be found in Appendix D Table 8. Only speculations have been made as to the source of the memory bottlenecks.

4.3 Choice of graphics engine

At the beginning of the project, the difficulty of using the Unreal Engine(UE) was underestimated. Even though UE had much built-in functionality, the documentation was lacking. Although Unreal Engine was poorly documented, its positive functionality started to show at the end of the project. The ease of using the built in Material Editor to make materials for the terrain, vegetation, and water. These materials are shaders which are used to color and shade the corresponding elements correctly. For example, the grass is green, and the water is reflective.

Unreal Engine is primarily a game engine and not a graphics engine. It is highly focused on the work flow of game development, where the visual components of the game is created by artists and put together in a game level during development. In this project the components where instead to be constructed during runtime and support for this in Unreal Engine is poor, which

limited the project heavily.

While there was some support built into the engine, documentation regarding a specific component we had to use was very sparse. As mentioned in Section 3.5.5, Procedural Mesh Component was used as an interface to render the procedurally generated models. We think the reason documentation was poor for this specific component was because of its experimental nature. Partly because of this, much time and effort had to be expended to properly understand and use it.

4.3.1 Visual effects

Since the project was developed in the Unreal engine, some of the engine's graphical features were utilized. Post-process effects such as lens flare and ambient occlusion were used to enhance the style. Figure 19 shows the ambient occlusion effects and Figure 20 displays the lense flares.

(a) No AO (b) AO

Figure 19: Seamscape trees illustrating Unreal engine's built-in ambient occlusion.

Figure 20: Unreal engine's built-in lense flare feature.

A day and night cycle system were implemented, and Figure 21 shows the terrain at different hours. The cycles improved the versatility of mood in the landscape, by setting different scenes.

(a) Midday (b) Evening (c) Midnight

Figure 21: The day-night cycle system with Unreal engine's visual effects.

4.4 Seamscape in a sustainable development perspective

While Seamscape is a computer application, it is still of importance to discuss its implications regarding sustainable development. Firstly, there are no notable effects of the software for the environment, as it is dependent only on the accompanied computer. Additionally, it is not a process which requires the computer running for more than at most a couple of minutes, unlike for example server farms which need a continuous surge of electricity for extended periods of time. Its purpose is to reduce the amount of time required for landscape creation and has limited implications on electricity use. In an economic perspective, it is sustainable as it is both free, and is aimed only to be an asset to game developers. Finally, regarding social aspects, Seamscape does not infringe human rights or have any direct implications on mental or physical health.

4.5 Seamscape in comparison to similar software

There are several software applications which can procedurally generate landscapes. This section analyzes what features are unique to Seamscape in the procedural generation of 3D scenes industry. Two of them are Terragen 3 and Outerra. In this section, the capabilities of Seamscape are compared to those of Terragen 3 and Outerra.

Terragen 3 is a photorealistic 3D environment renderer and design tool which uses procedural methods to generate landscapes [40]. For a scene shown in Terragen [40] the rendering time was over 19 minutes. Seamscape is capable of rendering scenes generated procedurally in real-time using rasterized rendering as opposed to ray-tracing. Seamscape produces landscapes with a low polygonal style which Terragen 3 does not.

Outerra is another software which utilizes procedural techniques to achieve seamless zooming from the ground to the atmosphere. Outerra renders terrain in real-time, as does Seamscape. Outerra can import real world elevation data and generate terrain which resembles the real world. However, Outerra does not produce landscapes with a low-polygonal style. Outerra's feature list does not mention whether Outerra is capable of generating vegetation models or rivers. Seamscape is capable of producing vegetation models in rivers in real-time. [41]

In conclusion, while the other two 3D landscape generating systems are refined and can produce realistic environments, Seamscape has some features which are unique between the three. In comparison to said software applications, Seamscape is the only one which is capable of generating landscapes which have a low-polygonal style.

5 Conclusion

The purpose of the thesis was to create a 3D landscape generator by utilizing procedural generation methods while maintaining real-time frame rate. There were goals regarding terrain, vegetation, water, the wind and a graphical user interface (GUI). While not all features made it into Seamscape, such as wind and water bodies, the ones which made it adheres to the low-polygonal style. The procedural generation of the landscape was performant and therefore the chosen methods were suitable for real-time generation of scenes.

However, there are several points of improvement which can be considered in Seamscape's future. Increasing the variation in each plant species could contribute to more diverse environments. Another feature is the morphing of species to produce new ones. Furthermore, using other models for the leaves could yield more variation both within the trees and in the biome as a whole.

Apart from the vegetation, the wind implementation could be resumed in the future. Also, the water model could be extended to use physics. Using water physics the terrain could have waterfalls, and non-horizontal water streams. Possible improvements for the terrain includes overhangs and local variation.

The project also had big plans for the GUI, which had to be cut off for the final deadline. This GUI could provide features such as choosing the type of biome, the time of the day, and the sharpness of mountains or terrain elements.

Furthermore, Seamscape has a combination of features which are relatively unique compared to similar products. For example, Terragen and Outerra are both capable of generating realistic landscapes. However, Seamscape can create them in real-time, and uses a low-polygonal style.

Finally, the project managed to create a landscape generator with a visual result. It also succeeded in producing terrain in real-time. It features stylized environments combining many heterogeneous procedural generation methods into a homogeneous whole.

6 References

[1] G. J. Mitchison, "Phyllotaxis and the fibonacci series", *Science*, vol. 196, no. 4287, pp. 270–275, 1977, ISSN: 00368075, 10959203. [Online]. Available: http://www.jstor.org/stable/1743115.

[2] P. Prusinkiewicz, J. Hanan, F. Fracchia, A. Lindenmayer, D. Fowler, M. de Boer, and L. Mercer, *The Algorithmic Beauty of Plants*, ser. The Virtual Laboratory. Springer New York, 2012, pp. 31, 1–35, ISBN: 9781461384762. [Online]. Available: https://books.google.se/books?id=4F7lBwAAQBAJ.

[3] *First use of procedural generation in a video game*. [Online]. Available: http://www.guinnessworldrecords.com/world-records/first-use-of-procedural-generation-in-a-video-game.

[4] J. Hammes, "Modeling of ecosystems as a data source for real-time terrain rendering", in *Digital Earth Moving*, Springer, 2001, pp. 98–111.

[5] D. Williams, "Structure and competition in the us home video game industry", *International Journal on Media Management*, vol. 4, no. 1, pp. 41–54, 2002, ISSN: 1424-1277. DOI: 10.1080/14241270209389979. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/14241270209389979.

[6] P. Software. (2015). First look at terragen 4, [Online]. Available: http://terragen4.com/first-look/ (visited on 04/27/2016).

[7] J. Tidwell, *Designing User Interfaces*. O'Reilly, 2006.

[8] I. Thekla. (2016). Team, [Online]. Available: http://the-witness.net/news/team/ (visited on 04/27/2016).

[9] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes, "A survey on procedural modelling for virtual worlds", *Computer Graphics Forum*, vol. 33, no. 6, pp. 31–50, 2014, ISSN: 14678659. DOI: 10.1111/cgf.12276.

[10] X. Qin, "A generalized cellular texture basis function", in *Proceedings of the Twelfth Annual Graduate Symposium on Computer Science*, University of Saskatchewan, Apr. 2000, p. 158. [Online]. Available: https://webdocs.cs.ualberta.ca/~yang/Technical_reports/cellular\%20texture\%202000.pdf.

[11] K. Perlin, "Making noise", 1999. [Online]. Available: http://www.noisemachine.com/talk1/index.html.

[12] D. Mount, *Procedural generation: 1d perlin noise*, University Lecture, Mar. 2016. [Online]. Available: https://www.cs.umd.edu/class/spring2016/cmsc425/Lects/lect11.pdf.

[13] D. P. K. P. S. W. David S. Ebert F. Kenton Musgrave, *Texturing and Modeling: A Procedural Approach*. Academic Press, 1994, pp. 74–57,83–84,296.

[14] S. Worley, "A cellular texture basis function", pp. 291–293, 1996.

[15] G. Zadora, A. Martyna, D. Ramos, and C. Aitken, *Statistical Analysis in Forensic Science: Evidential Values of Multivariate Physicochemical Data*. John Wiley and Sons, 2014, pp. 85–89, ISBN: 978-0-470-97210-6.

[16] B Benes and R Forsbach, "Visual simulation of hydraulic erosion", *Wscg'2002, Vols I and Ii, Conference Proceedings*, pp. 79–86, 2002, ISSN: 01492136. DOI: 10.2118/10504-PA. [Online]. Available: {\textless}GotoISI{\textgreater}://000176365300012.

[17] J.-D. Génevaux, É. Galin, E. Guérin, A. Peytavie, and B. Beneš, "Terrain generation using procedural models based on hydrology", *ACM Transactions on Graphics*, vol. 32, no. 4, p. 1, 2013, ISSN: 07300301. DOI: 10.1145/2461912.2461996. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2461912.2461996.

[18] A. D. Kelley, M. C. Malin, and G. M. Nielson, "Terrain simulation using a model of stream erosion", *ACM SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 263–268, 1988, ISSN: 00978930. DOI: 10.1145/378456.378519.

[19] Y. Collet, *Xxhash - extremely fast non-cryptographic hash algorithm*. [Online]. Available: https://github.com/Cyan4973/xxHash.

[20] I. Parberry, "Designer worlds: Procedural generation of infinite terrain from usgs elevation data", Denton, Texas, USA, Tech. Rep., Aug. 2013.

[21] T. Archer. (). Procedurally generating terrain, Morningside College, [Online]. Available: http://micsymposium.org/mics_2011_proceedings/mics2011_submission_30.pdf.

[22] FreeTheNation. (2011). An in depth cell noise tutorial, AFTbit, [Online]. Available: https://aftbit.com/cell-noise-2/ (visited on 05/15/2016).

[23] C.-J. Rosén, "Cell noise and processing", 2006. [Online]. Available: http://www.carljohanrosen.com/share/CellNoiseAndProcessing.pdf.

[24] N. Blevins. (2015). Procedural patterns and noises, [Online]. Available: http://www.neilblevins.com/cg_education/procedural_noise/procedural_noise.html (visited on 05/15/2016).

[25] S. Gustavson. (2005). Simplex noise demystified, Linköping University, [Online]. Available: http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf (visited on 05/14/2016).

[26] G. Croft. (2013). 6 ways speedtree® "speeds" the creation process, [Online]. Available: http://blog.speedtree.com/2013/10/6-ways-speedtree-speeds-creation-process/ (visited on 05/13/2016).

[27] ——, (2014). Procedural vegetation generation at runtime for games?, [Online]. Available: http://www.speedtree.com/forum/showthread.php?3227-Procedural-vegetation-generation-at-runtime-for-games (visited on 05/13/2016).

[28] A. Runions, B. Lane, and P. Prusinkiewicz, "Modeling trees with a space colonization algorithm", *Natural Phenomena*, pp. 63–70, 2007, ISSN: 18160867. DOI: 10.2312/NPH/NPH07/063-070.

[29] P. Prusinkiewicz and J. Hanan, "Extensions to the graphical interpretation of l-systems based on turtle geometry", pp. 1–56, 1997.

[30] I. M. Dart, G. De Rossi, and J. Togelius, "Speedrock: Procedural rocks through grammars and evolution", in *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, ACM, 2011, p. 8.

[31] H. Si, "Tetgen: A quality tetrahedral mesh generator and a 3d delaunay triangulator (version 1.5 — user's manual)", pp. 19, 33–34, 2013. [Online]. Available: http://www.wias-berlin.de/techreport/13/wias_technicalreports_13.pdf.

[32] E. Berberich, J. Reichel, and F. Cacciola, *Cgal 4.7 - manual*, 2015. [Online]. Available: http://doc.cgal.org/latest/Manual/installation.html (visited on 03/28/2016).

[33] *Tetgenlink introduction*, 2016. [Online]. Available: http://reference.wolfram.com/language/TetGenLink/tutorial/Introduction.html (visited on 03/28/2016).

[34] MetalGameStudios. (2014). Unreal engine 4 tutorial: Wind effect (english), [Online]. Available: https://www.youtube.com/watch?v=bAzKVaH0IZg (visited on 05/12/2016).

[35] J. Diener, M. Rodriguez, L. Baboud, and L. Reveret, "Wind projection basis for real-time animation of trees", *Computer Graphics Forum*, 2009, ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2009.01393.x.

[36] S. Ota, M. Tamura, T. Fujimoto, K. Muraoka, and N. Chiba., "A hybrid method for real-time animation of trees swaying in wind fields", *The Visual Computer*, pp. 19, 33–34, 2004.

[37] J. H. Robert Bridson and M. Nordenstam, "Curl-noise for procedural fluid flow", 2007. [Online]. Available: `https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph2007-curlnoise.pdf`.

[38] F. Anastacio, P. Prusinkiewicz, and M. C. Sousa, "Sketch-based parameterization of l-systems using illustration-inspired construction lines and depth modulation", *Computers and Graphics (Pergamon)*, vol. 33, no. 4, pp. 440–451, 2009, ISSN: 00978493. DOI: `10.1016/j.cag.2009.05.001`.

[39] M. Sanchez, O. Fryazinov, T. Vilbrandt, and A. Pasko, "Morphological shape generation through user-controlled group metamorphosis", *Computers and Graphics (Pergamon)*, vol. 37, no. 6, pp. 620–627, 2013, ISSN: 00978493. DOI: `10.1016/j.cag.2013.05.009`. [Online]. Available: `http://dx.doi.org/10.1016/j.cag.2013.05.009`.

[40] P. Software. (2016). Terragen 3, [Online]. Available: `http://planetside.co.uk/products/terragen3` (visited on 05/16/2016).

[41] Outerra. (2016). Outerra, [Online]. Available: `http://www.outerra.com/wfeatures.html` (visited on 05/16/2016).

Appendices

A Green tree L-system definition

The green trees shown in Figure 9 and Figure 18 was generated by the following L-system:

$$\omega : T(140, 90, 40, 100)$$

$$P_1 : T(r, g, b, \alpha) \xrightarrow{1} \#(r, g, b) > F(6, 1)[T(\mathbf{t})][+(\tfrac{\pi}{3} + X)T(\mathbf{t})][-(\tfrac{\pi}{3} + X)T(\mathbf{t})][\&(\tfrac{\pi}{3} + X)T(\mathbf{t})][\wedge(\tfrac{\pi}{3} + X)T(\mathbf{t})]$$

$$P_2 : F(dist, numCircles) \xrightarrow{1} F(1.7 * dist, numCirlces) >$$

$$P_3 : T(r, g, b, \alpha) : \alpha < 98 \xrightarrow{25} F(8, 1)L(20, 0, leaf.r + Y, leaf.g + Y, leaf.b + Y, 1)$$

$$\textit{where } X \sim \mathcal{U}(-\tfrac{\pi}{6}, -\tfrac{\pi}{6}) \textit{ and } Y \sim \mathcal{U}(-30, +30)$$

An explanation of each variable follows:

- The variables r, g, b constitute the color of a cylinder. α is a variable used for conditioning production rule P_3 such that leaves become very probable on the fourth step.
- $dist$ is the distance the turtle should travel.
- $numCircles$ correspond to how many circles the bent cylinder should consist of.
- The angles are randomly perturbed by X.
- The color of a leaf generated by L is determined by variables $leaf.r, leaf.g, leaf.b$ which are set programatically.
- The color per leaf is perturbed by Y, such that the colors of leaves vary.
- The first parameter of L determines the number of vertices in the point cloud generated by a stone generator.
- The second parameter of L determines shape of the ellipsoid, where $0 = sphere, 1 = oblate, 2 = prolate$, which defines the shape of the point cloud.
- The last parameter of L determines displacement variance σ.

The parameters to T are written as $\mathbf{t}$ $(=r, g, b, 0.99\alpha)$ for the sake of brevity. This system was stepped four times.

An brief explanation of the production rules is presented bellow:

- P_1 generates the structure of the tree.
- P_2 makes branches closer to the root taller and branches further away from the root thinner.
- P_3 makes leaves very probable at top of branch on step 4.

B Mathematical model for 2D noise maps

The purpose of this section is to obtain the coordinates needed as input to the hash functions in order to generate the noise outputs.

The entire noise map which models the heightmap or some other application is divided into a two dimensional grid of tiles and also a 2D grid of noise cells. Every tile represent a subregion of the noise map with a given level of detail. The level of detail defines the number of noise output points in a region. Border points overlaps on neighbouring tiles in order to avoid discontinuity. We define a 2D-cartesian coordinate system which axes are parallel with the axes of the tile grid. The origin of the cartesian system is defined as one the corners of the $(0, 0)$-tile. This coordinate system will be referred as the global coordinate system and it is the global coordinates which are used as inputs for the hash functions.

Consider an arbitrary tile T and let the indices which defines the position of T in the grid be denoted $x_T, y_T \in \mathbb{Z}$. Figure 3 illustrates an example of the tile grid. Every tile has local origin denoted $\vec{O_T}$ and it is defined as the point in the tile closest to global origin. The local origin for the $(0, 0)$ tile coincides with the origin of the global system, as shown in the Figure 3.

If d_t denotes the side length of a tile, the coordinates of the local origin $\vec{O_T}$ of tile T will be at the following coordinates in the global system:

$$\vec{O_T} = d_t x_T \hat{x} + d_t y_T \hat{y}$$

$\hat{x}$ and $\hat{y}$ are unit vectors along the x and the y-axis of the global system. x_T and y_T are indices of tile T in the world grid.

The noise output points are distributed over the tiles through rectangular lattices(one lattice for one each tile). The number of lattice points are as mentioned before defined by the level of detail for the given tile. Let N denotes the number of dimensions for the noise map and n the number of lattice points in one dimension. The number of lattice elements for a tile are n^N and in our case $n \times n$ number of lattice elements. The lattice spacing d for a given n are determined by following formula:

$$d(n) = \frac{d_t}{n-1}$$
$$1 < n \in \mathbb{N}$$

The model supports only n as an integer larger than 1. Note that the dependency of n causes the lattice spacing d to not be constant and independent for every tile.

Let an arbitrary lattice element for tile T be denoted $T_{i,j}$ where i, j are lattice indices.

$$0 \leq i, j < n$$

The lattice indices define the corresponding position of element $T_{i,j}$. This position can be expressed in the global system:

$$\boxed{\vec{r}(T_{i,j}) = d(i\hat{x} + j\hat{y}) + \vec{O_T} = (d_t x_T + id)\hat{x} + (d_t y_T + jd)\hat{y}} \quad (1)$$

$\vec{O_T}$ is point in T closest to the global origin.

Now consider a pseudo random noise with cell length L. Just like the grid of tiles, there is a grid which consists of noise cells, see Figure 4 in Section 3.5.4 for an illustration. For the sake of simplicity the origin of noise grid coincides with the tile grid origin. Every noise cell also has

a local origin $\vec{O_{cell}}$ which is defined as the point closest to the global origin within the noise cell. In order to generate a noise value $T_{i,j}$ at a certain point $\vec{r}(T_{i,j})$ information about what noise cell the point lies within is needed. In order to generate all lattice elements of a tile all noise cells which cover some part of the tile must be known. In some cases such as the Worley noise, even the surrounding cells i needed as well. The number of cells N_{cells} which cover some part of the tile in **one dimension** can easily be determined by:

$$N_{cells} = \lceil \frac{d_t}{L} \rceil$$

$\lceil x \rceil$ is the ceil function which rounds up the value x to the nearest integer larger than or equal to x. Now we want to find the local origin $\vec{O_{cell}}$ of the cell which covers the local origin of the tile $\vec{O_T}$. One must consider a possible displacement between $\vec{O_T}$ and $\vec{O_{cell}}$. The origin $O_{cell,O}$ can be done in the following way:

$$\vec{O_{cell,O}} = \lfloor \frac{d_t x_T}{L} \rfloor L\hat{x} + \lfloor \frac{d_t y_T}{L} \rfloor L\hat{y}$$

$\lfloor x \rfloor$ is the floor function which rounds down the value x to the nearest integer smaller than or equal to x. The origin of unit cell which cover the point in the tile furthest from the tile origin $\vec{O_T}$ can be found at:

$$\vec{O_{cell,furthest}} = \vec{O_{cell,O}} + N_{cells} L(\hat{x} + \hat{y})$$

Now define a local grid system of noise cells for tile T, with the cell which contains $\vec{O_T}$ as origin $(0, 0)$. Since this is a two dimensional model $N = 2$ the number of cells in the grid will be $N_{cells} \cdot N_{cells}$. For most noise functions such as Perlin and Worley we need to identify what cell an arbitrary point $\vec{r}(T_{i,j})$ lies within. The origin $\vec{P_{cell}}$ in the cell grid which contains $\vec{r}(T_{i,j})$ can be found:

$$\vec{P_{cell}}(T_{i,j}) = i_{cell}\hat{x} + j_{cell}\hat{y} = \lfloor \frac{r(T_{i,j})_x - x_T d_t + \Delta x}{L} \rfloor \hat{x} + \lfloor \frac{r(T_{i,j})_y - y_T d_t + \Delta y}{L} \rfloor \hat{y} \tag{2}$$

$$i_{cell}, j_{cell} = 0, 1, 2, 3...$$

i_{cell} and j_{cell} are indices and defines the position of the cell in the cell grid. Δx and Δy are the displacements between $\vec{O_T}$ and $\vec{O_{cell,O}}$.

$$\begin{cases} \Delta x = O_{Tx} - O_{cell,O,x} \\ \Delta y = O_{Ty} - O_{cell,O,y} \end{cases}$$

Now can all the needed input coordinates for the hash functions can be found. The corresponding position for a lattice element $T_{i,j}$ are found with (1). Our Perlin noise implementation uses the corner positions of every noise cell in order to generate the corner gradients and the Worley noise implementation uses $\vec{P_{cell}}$ as input to hash functions to generate the feature points within that cell. The corner positions $\vec{P_{corner}}(T_{i,j})$ of the noise cell enclosing $\vec{T_{i,j}}$ can be found by adding L in the x and/or y-direction.

$$\vec{P_{corner}}(T_{i,j}) = \vec{P_{cell}}(T_{i,j}) + L\left(k_x\hat{x} + k_y\hat{y}\right) \tag{3a}$$

$$k_x, k_y = 0, 1 \tag{3b}$$

C Code snippets

```
auto exampleCondition = [](const auto& params) { return (params[1] *
    params[2] / params[0]) < 4.0f; };
```

Code 1: A condition which states that a production rule is legal if $\frac{\beta\omega}{\alpha} < 4$

```
using namespace std::chrono;
auto start = high_resolution_clock::now();
auto obj = <call to expensive function or method>;
auto end = high_resolution_clock::now();
float elapsed = duration_cast<duration<float>>(end - start).count();
static float totElapsed = 0;
totElapsed += elapsed;
static int n = 0;
n+=1;
UE_LOG(LogTemp, Warning, TEXT("%fs, totElapsed:%fs, avg:%fs, n:#%i"),
    elapsed, totElapsed, totElapsed/n, n);
```

Code 2: Measuring time spent in a function or method using std::chrono library

D Raw data

Ecosystem		**Elev.**	**Rel. Elev.**	**Slope**	**Plants**	**Density**	**LoD**	**Scale**
Highland	Min	70	-1	0	CherryTree	8	1	8 to 20
	Max	260	5	1.4	Stone	1	0	8 to 13
	Sharpness	2	1	2				
Marshland	Min	-10	-5	-0.4	GoldTree	8	2	12 to 20
	Max	50	0	0.6	Shrub	5	2	15 to 25
	Sharpness	2	1	2				
Plains	Min	-50	0.7	-0.4	GreenTree	8	2	6 to 11
	Max	350	3	1.6	Grass	4	4	3 to 8
	Sharpness	2	1	2				
Slope	Min	-50	0	1.4	Stone	1	1	6 to 18
	Max	350	10	2.4				
	Sharpness	2	1	2				
Scarp	Min	-50	-10	2.8	Stone	1	2	6 to 18
	Max	350	10	8.4	DeadShrub	1	1	90 to 100
	Sharpness	2	1	8				

Table 1: A definition of ecosystems and plants we use.

Run	Total elapsed time (s)	Average elapsed time (s)	Number of objects placed
1	2.278152	0.016751	29841
2	2.900134	0.021325	28573
3	1.823192	0.013406	29354
4	1.954309	0.014370	30102
5	2.911668	0.021409	28988
Avg	2.373491	0.017452	29372

Table 2: Object placement performance, with generation of objects

Run	Total elapsed time (s)	Average elapsed time (s)	Number of objects placed
1	0.579892	0.004264	30847
2	0.476891	0.003507	29190
3	0.445251	0.003274	29855
4	0.430308	0.003164	29124
5	0.387165	0.002847	28927
Avg	0.463901	0.003411	29572

Table 3: Object placement performance, without generation of objects.

For both Table 2 and Table 3, measurement was carried out using code given in Appendix C Code 2. The measurements in Table 2 and Table 3 were obtained on a computer with the following specifications: *Intel Core i5-4690K @ 3.50GHz, 8GB DDR3 1600MHz RAM, Kingston SSDNow V300 120GB, Asus GeForce 760GTX DC2 2GB.*

Run	Total elapsed time (s)	Average elapsed time (s)	Number of generated rocks
1	0.397389	0.000057	6962
2	0.380898	0.000055	6968
3	0.386329	0.000055	6983
4	0.361518	0.000056	6471
5	0.389698	0.000056	6969
Avg	0.383166	0.000056	6870

Table 4: Rock generator time measurement, without precomputation.

Run	Total elapsed time (s)	Average elapsed time (s)	Number of generated rocks
1	0.322295	0.000050	6502
2	0.365891	0.000053	6954
3	0.350722	0.000050	6987
4	0.346283	0.000050	7971
5	0.342787	0.000049	6981
Avg	0.345595	0.000050	6981

Table 5: Rock generator time measurement, with $\#precomputes = 40$

For both Table 4 and Table 5, measurement was carried out using code given in Appendix C Code 2. The measurements in Table 4 and Table 5 were obtained on a computer with the

following specifications: *Intel Core i5 6500 @ 3.20GHz, 16GB DDR4 2133MHz RAM, Samsung 850 EVO 250GB, Asus GeForce 970GTX DirectCU Mini.*

Run	Total elapsed time (s)	Average elapsed time (s)	Number of generated trees
1	2.140936	0.002855	749
2	2.239591	0.002619	854
3	1.920525	0.002574	745
4	2.183934	0.002912	749
5	2.362100	0.002763	854
Avg	2.169417	0.002744	790

Table 6: Tree generator time measurement

Run	Total elapsed time (s)	Average elapsed time (s)	Number of generated trees
1	0.106382	0.000133	797
2	0.104018	0.000127	817
3	0.108752	0.000127	855
4	0.101714	0.000136	749
5	0.110624	0.000148	749
Avg	0.106298	0.000134	793

Table 7: Tree generator time measurement, with $\#precomputes = 40$

Measurements presented in Table 6 and Table 7 were carried out, using code given in Appendix C Code 2, on a machine with the following specifications: *Intel Core i5 6500 @ 3.20GHz, 16GB DDR4 2133MHz RAM, Samsung 850 EVO 250GB, Asus GeForce 970GTX DirectCU Mini.*

Case	Generated mesh data (MB)	Process memory usage (MB)
10x10 tiles with vegetation	4078	9483
10x10 tiles without vegetation	12	724
4x4 tiles with vegetation	824	2799
4x4 tiles without vegetation	8	600

Table 8: Memory usage reported after generation by Unreal Engine in different cases.