Bachelor of Science Thesis

Generating Compelling Procedural 3D Environments and Landscapes

Oscar Blomqvist | Pierre Kraft | Hampus Lidin

Rimmer Motzheim | Adam Tonderski | Gabriel Wagner

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Generating Compelling Procedural 3D Environments and Landscapes

Oscar Blomqvist
Pierre Kraft
Hampus Lidin
Rimmer Motzheim
Adam Tonderski
Gabriel Wagner

Supervisor: Staffan Björk, Department of Interaction Design and Technologies
Examiner: Arne Linde, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Göteborg
Telephone +46 (0)31-772 1000

Typeset in LaTeX
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2016

Generating Compelling Procedural 3D Environments And Landscapes

Oscar Blomqvist
Pierre Kraft
Hampus Lidin
Rimmer Motzheim
Adam Tonderski
Gabriel Wagner
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Bachelor of Science Thesis

Abstract

The game industry of today puts a lot of pressure on game developers. More than ever, the expectations of what developers can deliver are high, with increased computational power common in video game consoles and gaming PCs. For big budget games, the company or companies behind it often need to rely on hundreds of graphical designers to populate the gaming world with interesting content. This creates problems for small-scale independent game companies with limited resources, who want to compete on the game market. In order for smaller game companies to create comparable gaming worlds with substantially fewer developers, one technique they might use to achieve this is called *procedural content generation* (PCG). This technique utilizes the performance of a computer to generate assets based on mathematical algorithms.

In this thesis project the goal has been to create an open source PCG engine for game developers to use, in which procedural content generation plays an integral part. Implemented as a pipeline, the input to the engine yields deterministic output based on what modules are used with it. The modules can work on many different scales, from creating flowers to generating 3D landscapes. With a conveyor belt style approach, it is possible to generate large worlds filled with interesting content in a linear, deterministic way.

The result of the project consists of a handful of modules capable of generating 3D terrain with multiple biomes, with interpolation techniques to blend the different landscapes together. An L-system for generating trees has been implemented as well, along with a module for scattering objects in the game world in a natural way. The pipeline is connected to an external rendering engine. However, the modules have not been successfully integrated with the pipeline.

Keywords: game developer, independent, procedural, generation, pipeline, deterministic, modules, algorithms.

Sammanfattning

Dagens spelindustri sätter stor press på spelutvecklare. Förväntningarna av vad utvecklare kan leverera blir allt större, allteftersom datorer och spelkonsoler blir snabbare och kraftfullare. För storskaliga spel så krävs det att företaget eller företagen bakom det ofta behöver förlita sig på de hundratal grafikdesigners som fyller spelvärlden med intressant innehåll. Detta skapar problem för småskaliga, oberoende spelföretag med begränsade tillgångar, som vill konkurrera på spelmarknaden. För att dessa spelföretag ska kunna jämföra sina spelvärldar med de från storskaliga spel, finns det en teknik som småföretagen kan använda som kallas *procedurell generering av grafiska objekt* (*procedural content generation* på engelska, förk. PCG). Denna teknik utnyttjar prestandan hos en dator för att generera grafikobjekt och annat innehåll baserat på matematiska algoritmer.

I detta kandidatarbete har målet varit a skapa en PCG-motor med öppen källkod för spelutvecklare att använda, vari procedurell generering spelar en viktig roll. Motorn implementeras som en *pipeline* bestående av flera moduler. Data som skickas in till motorn ger deterministisk utdata baserat på vilka moduler som används ihop med motorn. Modulernas uppgifter kan arbeta på olika storlekskalor, från att skapa enskilda blommor till att generera hela 3D-landskap. Med detta rullbandslika tillvägagångssätt, är det möjligt att generera stora spelvärldar fyllda med interesant innehåll på ett linjärt, deterministiskt sätt.

Resultat från projektet består av en handfull moduler kapabla till att generera 3D-miljöer med olika biom, med interpoleringstekniker för att sammanfoga de olika terrängerna. Ett L-system för att generera träd har också blivit implementerat, tillsammans med en modul för att sprida objekt i spelvärlden på ett naturligt sätt. Motorns *pipeline* är sammankopplad med en extern renderingsmotor. Emellertid så har modulerna inte blivit integrerade med motorn.

Nyckelord: spelutvecklare, oberoende, procedurell, generering, pipeline, deterministisk, moduler, algoritmer.

Acknowledgements

All of the group members would like to thank our supervisor Staffan Björk, for providing us with advise and feedback throughout the project, and for putting up with us on the afternoon meetings.

Dictionary

API - acronym for *Application Programming Interface*.

Asset - a graphical object used in the game world, such as a building or a tree.

Barycentric coordinates - Barycentric coordinates are triples of numbers (t_1, t_2, t_3) that shows where a point P is located in a reference to a triangle. The barycentric coordinates of P are defined as the distance to each vertex of the triangle in relation to its opposite edge [1].

Bitmap image - an image format where each pixel is defined by color and transparency.

Chunk - a sub-set of data.

Interface bridge - an intermediary software component to make one API compatible with another.

LOD - acronym for *Level Of Detail*. The term is used for graphical objects with multiple layers of detail, e.g. the resolution of bitmaps or the number of polygons in a polygon mesh.

MVP - acronym for *Minimum viable product*. This is a product with just enough functionality to satisfy the core use cases, and that allows for validated learning about the product and its continuous development.

Noise - a collection of random values, represented as numbers.

Perlin worms - worm like structures generated using Perlin noise.

Voxelized - rendered as voxels.

Contents

List of Figures

1 Introduction

Digital games today are ever increasing in scope and detail which poses some challenges to the gaming industry. More and more people are required to develop a game, with projects often exceeding hundreds of graphical designers and programmers [2]. These kind of games are often referred to as *AAA* games, or *triple-A* games, which is a classification of the highest budget games on the market. An example is the franchise Assassin's Creed from Ubisoft who are able to release a new Assassin's Creed title every year.

The small scale game companies have a hard time keeping up with the high expectations of consumers, due to having smaller workforces compared to triple-A game companies. The small companies often compensate for large stunning worlds with innovate game mechanics, level design and storytelling instead, making them stand out in other areas. These games are often referred to as *indie games*, or *independent games*, and could often be seen as the opposite of triple-A games, although in some cases its hard to differentiate the two.

The production costs of triple-A games are increasing, and its becoming more difficult for companies to meet the consumer demands [3]. One technique that is sometimes employed is called *procedural content generation*, or shortened as PCG. This technique relies on the computational power of hardware to generate the game world assets, instead of having large amounts of people handcrafting every single asset in the game. Another advantage of PCG is that new and varied content can be generated easily. By consistently having the user provided with new content, the game experience is kept interesting.

An example where PCG can be used is for terrain landscapes. The landscape generator could make use of mathematical phenomena and algorithms in order to create mountain chains, valleys and large open fields. The generator would only take a single input, called a *seed*, which is processed in logical steps incomprehensible to the human mind, and the output generated appears random in a structured, natural manner. The process can be deterministic, meaning that a certain seed input will always generate the same output. This is important in order to produce content "on the fly", without having to store any data other than the seed and the algorithms producing the content.

1.1 Background

The first big game using PCG was *Beneath Apple Manor* [4] released in 1978. Back then the lack of memory restricted games from having rich game environments. PCG was a way to enrich game maps without using up memory. Early examples of this include *Rogue* [5] and *Diablo* [6],

which uses the technique to generate the caves that the player traverses. As computers became more powerful, using PCG for reducing memory load was no longer as common. But this also resulted in games becoming larger and larger. Nowadays hundreds of designers are involved to create high budget games. The use for PCG has subsequently changed from freeing up memory to reducing the work load, being useful whenever extensive variation is required.

In *Borderlands* [7] for example, PCG is used to generate (seemingly) endless combinations of weapons that the player can acquire in the game [8]. The most recent PCG game at the time of writing this paper, is *No Man's Sky* [9]. The game is set in a science-fiction inspired universe, with several galaxies in which the player can travel around. Each galaxy has billions of star systems with orbiting planets, and every one of them is filled with life, in the form of animals and plants, which the player can discover. This would be a massive amount of work without the aid of PCG. In fact, it would be practically impossible for any-sized team of developers to come close to handcraft even just a small fraction of what's generated in No Man's Sky.

There are currently some PCG tools to generate terrain and game assets. *Voxel Farm* [10] is a fully featured tool for generating and rendering procedural worlds, although it has a commercial license and seems to have limited support for endless worlds. *SpeedTree* [11] is a tree and vegetation generator tool used by many triple-A game companies, that integrates with game engines, such as *Unity* [12] or *Unreal Engine* [13]. There are also various smaller plug-ins for game engines that help with generating content procedurally. However, to our knowledge there are currently no frameworks that can generate an endless world from start to end.

1.2 Purpose

The purpose of this project is to start the development of an API, with the ability to generate interesting procedural environments for games. When finished, the API will act as an extension library, that must be integrated with an existing (although arbitrary) game engine in order to render the content. The API is also meant to be modular, allowing developers to create their own modules for the library, that can alter the generation process and ultimately the game world output

Furthermore, finding the most optimised methods of generating content will not be the major concern in this project. The main purpose is to generate interesting and compelling assets. However, when facing two or more ways of performing a certain task, speed will be considered when choosing the most suitable method.

Finally, the API is intended to be free for game developers, hobbyists or anyone interested in the technology. Free is emphasized as not only being free of charge, but as what is called *free software*, i. e. software that anyone can access, modify and share. With this freedom, it is possible to have several experienced, independent engineers contributing on the future development of the API. The belief is that the developers of the game industry would have a personal interest in this type of technology, which would drive the development further. In this project however, there will be no attempt to create, or obtain, a community around the API, but rather open the doors for the future.

1.3 Project goals

This project will focus on building the foundation of the principles that have been discussed. The goals of the implementation can be further divided into two tasks. The first task of the project is to build the engine structure that will produce deterministic game world data. This includes the core, in the form of a *pipeline*, and a set of modules. The pipeline will handle all the input and output of the modules and the possible metadata about the generated content. The second task is to connect the API with an existing game engine, to show that the pipeline is able to transfer the generated content from the modules.

1.3.1 PCG engine

The main task for this project is to develop an engine which is able to generate arbitrary assets, with a core pipeline of interchangeable modules to define the generation process. As a reasonable restriction, this project will only explore the generation of three-dimensional assets and voxelized terrains. The engine will therefore consist of a handful of basic modules, in order to showcase and validate the functionality of the core. The PCG engine will not be dependent on a specific existing game engine, but rather on the game engine specific interface bridges, that will have to be built to incorporate the PCG engine with the game engine. It should also be able to continuously stream all the data necessary to render the world.

The initial stages of the pipeline will focus on generating the general terrain and landscape of the world. This includes generating landmass, height maps and biome data. This information will be used to create the actual terrain, in this case represented as voxel data. The later parts of the pipeline will fill the world with objects and generate more specific terrain components, such as rivers and caves. As for techniques used for the environment generation, the terrain will be generated using noise functions. To procedurally generate interesting objects, L-systems will be investigated. This technique could be particularly useful for (but not limited to) trees and vegetation.

1.3.2 Rendering

While the main project task is to create an engine with PCG functionality, the next task is to render the generated worlds using an existing game engine. A custom rendering engine that has been developed by project member Rimmer Motzheim will be used to render the output of the PCG engine. However, when testing the individual modules, temporary rendering means will be used.

2 Theory of PCG Algorithms

Before moving on to discuss the chosen methods of generating a procedural world, some general concepts and algorithms must be introduced. Most of these methods have the same aim: achieving interesting, non-repeating and realistic content. Realistic, in this case, does not necessarily mean obeying all laws of physics, but being somewhat familiar and making aesthetic sense. A good example would be islands that are floating in the sky. There are two fundamentally different ways of approaching this [14].

The first approach is to use an algorithm that produces the desired content by imitating the end result. These methods are called *ontogenetic*. For example, waves in the ocean look a lot like the mathematical sine-wave. Therefore one could generate a procedural ocean by combining different sine functions. This approach does not take any consideration of the underlying physics, but can still produce very realistic content. The main advantage is usually the generation speed, since the real world aesthetics are mimicked in an optimised way.

The other group of methods are called *teleological* and they take the opposite approach. They generate realistic content by using physical laws and processes as the basis. The previous example of generating ocean waves could use the Navier-Stokes equations[1], combined with tidal forces and heat currents in the ocean. This method could generate incredibly realistic waves, at the cost of performance.

In the following sections, techniques for generating random data will be discussed, followed by some different graphical representation techniques for terrain. Furthermore, algorithms for producing various environmental attributes will be explored, followed by some methods for dividing up an arbitrarily large, generated world.

2.1 Generating random data

Procedural generation works by having a process or an algorithm which generates some specific content, following some set rules and a given input. The strength of using procedural generation lies in being able to generate large amounts of content using computer performance instead of creating the content by hand. However, a purely deterministic procedural process will always generate the same result. Introducing some random factors in the process can lead to seemingly infinite amounts of variation of the generated content, avoiding repetition. Using purely random factors leads to content having no controllable reproducibility, which may not be desirable in some cases. Since the PCG engine in this project is specified as being deterministic, methods

[1] A set of equations that govern fluid dynamics.

for making a process random, but reproducible are required.

A very common way to produce random numbers is to use *pseudo-random number generators*, or PRNGs for short. These generators consist of algorithms that emulate truly random number generators (RNGs). Because PRNGs are based on mathematical operations, as opposed to RNGs, they are entirely deterministic[2] and depend entirely on their initial state. The initial state, which can be a number or a vector of numbers, is commonly referred to as a *seed*.

One of the earliest examples of a PRNG is the *Middle-Square Algorithm*, presented by John von Neumann in 1951 [15]. Today there are modern PRNGs that are much more sophisticated, for example xorshift generators [16] and the Mersenne Twister [17]. It is important to remember that they can never be truly random, and are therefore bound by certain limitations (that vary between algorithms). Also, it is never safe to assume that the standard random generators of modern languages implement one of the "good" algorithms. *Java*, which is one of the most popular languages in programming, uses a linear congruential generator [18].

2.2 Noise

When talking about noise one commonly refers to white noise, which can be recognised as the black and white dots on an old television set when there was no signal. When represented with numbers, white noise is a series of random uncorrelated values. For example a one dimensional white noise function could be a function that for a given input returns a decimal value between 0 and 1. Uncorrelated means that two noise values $f(x_1)$ and $f(x_2)$ will have no correlation unless $x_1 = x_2$.

Noise is a common method to use for randomness, since it can be defined by deterministic PRNGs, allowing for reproducibility. However, a surface generated using white noise would be very spiky and not resemble natural terrain. Instead what is desired is something that is seemingly random, but with the property that nearby points generate similar noise values.

2.2.1 Coherent noise

In 1983 Ken Perlin created a type of noise called *Perlin noise*. Unlike regular noise, which is generated using only a seed, Perlin noise is generated using a seed and some other input. Perlin noise is deterministic in regard to the input used to generate the noise. For example, in two dimensions a noise value could be a function f of some coordinates x and y which will, given a specific seed, always return the same value $f(x, y)$.

In addition of being deterministic, Perlin noise is also coherent. This means that the noise value $f(x, y)$ is not significantly different from a noise value in $f(x + \Delta x,\ y + \Delta y)$, for small Δx and Δy. The result is a type of semi-random noise.

In 2001 Ken Perlin developed a new type of noise called *simplex noise*, which is an improvement and a successor to Perlin noise. Simplex noise has a lower computational complexity, scales better for higher dimension of input and fixed some problems that Perlin noise had [19].

Coherent noise has three main characteristics:

[2]Given the same parameters and initial state.

- A small change in input generates a small change in the output.
- A big change in input generates a random output.
- The noise value generated from an input is deterministic, meaning the output will change only if the input changes.

These properties make coherent noise more aesthetically pleasing in graphical applications. Ken Perlin created Perlin noise to help artists create more complex and natural looking textures, like smoke or marble textures, using computer graphics [20]. An example of a smoke texture generated using simplex noise can be seen in Figure 2.1. With time many new applications have been found for Perlin and simplex noise, such as generating landscapes. A noteworthy example is the game *Minecraft* [21], which uses a system based on 3D Perlin noise to create a seemingly infinite game world [22].

Figure 2.1: Visualization of 2D simplex noise, generated using simplex noise implementation by Gustavson [23] and plotted using Matlab.

2.2.2 Combined noise

Noise can be combined by adding the noise from different or the same function. This can increase what can be called the *resolution* of the noise.

One method which is widely used is multi-octave noise. Increasing an octave means adding noise of double the frequency to the previous noise. This can be repeated as many times as required, resulting in the sum:

$$f_N(x) = \sum_{i=0}^{N} f(2^i x) \tag{2.1}$$

where $f_N(x)$ is the total octave-noise of order N (one less than the number of octaves) and $f(x)$ is the noise value, in the coordinate x. It is practical to normalise by $N + 1$ for f_N to remain in the original range of $f(x)$.

Another useful combination of coherent noise functions is ridged, or turbulent, noise. This method results in clear, coherent and narrow bands (Figure 2.2). This is accomplished by squaring the noise value over multiple octaves:

$$f_N(x) = \sum_{i=0}^{N} f(2^i x)^2 \tag{2.2}$$

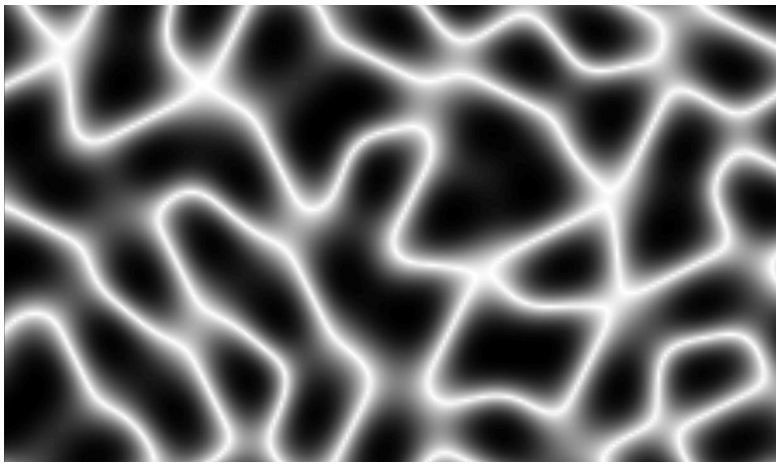


Figure 2.2: Turbulence noise created by modifying simplex noise. The modification was squaring the noise value for each octave before adding them together. The result is a connected net of ridges.

2.3 Terrain representation

While there are many different ways to create terrain, both procedurally and by hand, the final result needs to be rendered by a computer. In real-time rendering, such as in games, the only way to draw a scene with acceptable performance is through the GPU, which means that the terrain needs to be converted to triangles. This gives some constraints to how the final result can be stored in memory, since the representation needs to be able to be triangulated.

The most common ways of storing terrain are through direct final geometry (polygon mesh), heightmaps, and voxels. Direct geometry is the most straightforward, since the stored data can be drawn directly by any GPU. It also gives full control to artists, as they can directly modify the final geometry, but requires the most storage space and largest memory bandwidth. Heightmaps are most commonly used for terrain that can be represented adequately with only a scalar height at each point, while voxels can represent terrain with 3D-features such as overhangs.

2.3.1 Heightmaps

A heightmap is, as the name suggest, a map from coordinates (x, y) to the surface elevation z in that coordinate. In other words, it is a 2D projection of a 3D terrain, which can be stored as a bitmap image. The main advantages of this are:

- Light memory usage.
- Easily modified (can even be drawn by hand).
- Very efficient rendering and LOD.
- Fast collision detection.

There are however some limitations with heightmaps, one of them being the inability to represent complex 3D formations such as caves, overhangs, bridges, etc. It is also quite difficult to modify the terrain after it has been generated.

2.3.2 Voxels

The other commonly used representation is by using voxels. A voxel describes a cell in a three-dimensional grid. Therefore any shape can be described with this representation as long as the cell size is small enough. This form of data representation was massively popularised by the game Minecraft, but has actually long been used in many gaming and non-gaming applications, e. g. for storing the data from MRI-scans.

Since a voxel is such a general concept it can contain any data. For example, Minecraft assigns a "block type" to every voxel, which describes its material (water, air, grass, rock, etc.). In medical applications, they can store vector information such as blood flow rate. It is however important to try to minimize amount of data stored in each voxel since a world will consist of extremely many voxels. A quick example: if each voxel stores 1 byte of data and represents a size of $1\,\mathrm{m}^3$ the memory requirements for storing a $1000\,\mathrm{m} \times 1000\,\mathrm{m} \times 200\,\mathrm{m}$ world are 200MB.

Figure 2.3 shows how Minecraft renders its terrain. The voxel nature of Mincraft's representation is clearly visible in the form of cubes. It is important to note that voxels do not have to be rendered this way. A technique called *iso-surface extraction* can be used create a polygon mesh from voxel data. An algorithm which implements iso-surface extractions is *Marching Cube Algorithm*, presented by Lorensen and Cline in 1987 [24].

Figure 2.3: The game world of Minecraft consists of large blocks, called voxels. The terrain is generated using PCG techniques. CC0 1.0 [25].

2.4 Bilinear interpolation

In some situations, it is desirable to have a way to smooth out discontinuous or non-smooth consecutive data points by *interpolating* between them. Interpolation is done by determining the value of an unknown data point by looking at existing data points. One of the most basic interpolation methods is *linear interpolation*. Linear interpolation finds the value of a point P by looking at two points, A and B, that are located on the opposite ends of a straight line that goes through P. By calculating the average of A and B, weighted by their distance to P, the value of P is deduced [26].

Bilinear interpolation can be seen as an extension of *linear interpolation*, where the idea is to perform *linear interpolation* in two directions. The value of a point P is therefore calculated by looking at four known points surrounding P, and the interpolated value is acquired by taking the weighted average of these four points (Figure 2.4).

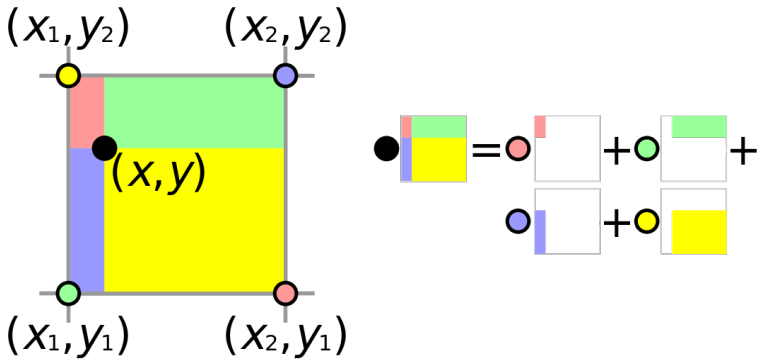


Figure 2.4: Geometric visualization of bilinear interpolation. CC BY-SA 3.0 [27].

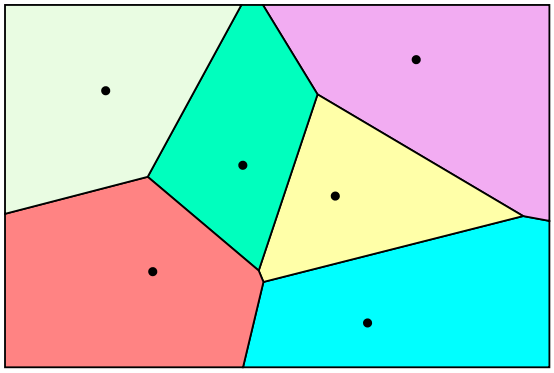
2.5 Voronoi diagrams

When generating large continents and oceans for large scale terrains, *Voronoi diagrams* can be used. Voronoi diagrams can describe simple shapes like squares and hexagons, but it can also form highly complex, unstructured graphs.

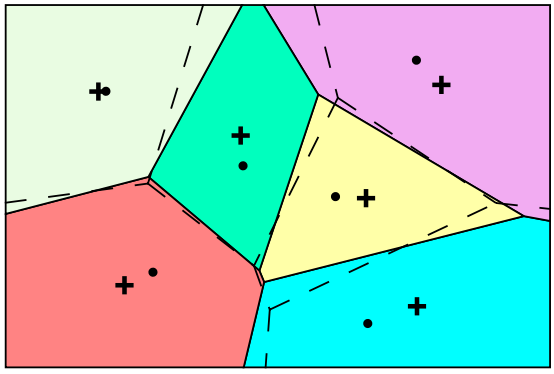
Voronoi diagrams are defined as planes partitioned into polygons based on a set of predefined points called *sites* (Figure 2.5a). The polygons are defined as the subset of the plane whose points are closer to a particular site than to any other site. This will indeed divide the plane into several polygons, each subsequently having exactly one site contained in them. Therefore, by having a larger set of sites, the partitioning will become more detailed. The diagram can then be used to "fill in" each polygon with either land or ocean. To calculate a Voronoi diagram, *Fortune's algorithm* [28] can be used.

To get somewhat regularly sized cells in the Voronoi diagram, a low discrepancy random generator is needed. Lloyd's algorithm, also known as Voronoi iteration [29], is a technique that gives this effect in the form of *blue noise*, which is usually characterized as noise with few low-frequency components [30].

The algorithm is simple; it takes the set of sites in a Voronoi diagram and reposition them in the centroids of the respective polygons. This is repeated as many times as required, making the sites more evenly spaced (Figure 2.5b).



(a) A Voronoi diagram is defined as a plane subdivided into polygons. The polygons are defined as the set of points which are closer to a particular site than to any other site.



(b) The same diagram with the sites of first iteration of Lloyd's algorithm marked as crosses. The dashed polygons indicate the Voronoi diagram of the new relaxed sites.

Figure 2.5: Example of Voronoi diagram and the use of Lloyd's algorithm.

2.6 L-systems

The modelling of biological structures, such as plants and trees, can be done using a technique called *Lindenmayer*-systems (L-systems). L-systems was introduced by Aristid Lindenmayer in 1968 [31] and it suggested a framework of formal rules and definitions used to generate linear or branching filaments. Since then, several suggestions and variations have been made to the L-system, and it still is a popular way of modelling plants and trees using computer graphics.

In the most basic definition, an L-system consists of a set of symbols and rules. These rules are called *productions* and they are performed during each iteration of the generation. The structures are represented with symbols, that may or may not have a production defined to it. If they do, then these symbols are classified as *variables*, meaning that they will change after each iteration. If no production is defined for a symbol, then it is called a *constant*, and are only used as a building block to the structure. The generation begins by initialising the generation string to the *axoim*, which is one of the variables in the L-system. Then, during each iteration, the corresponding production rules are applied to each symbol in the generation string. This basic type of L-system is referred to as a deterministic, context-free L-system (D0L-system).

A variation to this is a *stochastic* L-system. Stochastic L-systems provide another degree of complexity to the generation, where a symbol may have two or more productions defined. Each production will then have a certain probability of being chosen, and thus it provides a way of generating many different structures of similar types.

To be able to draw structures onto a canvas, a *turtle interpreter* can be used. A turtle interpreter

holds a state of its position and orientation. Some or all variables in the L-system are assigned an action that either outputs a graphical change on the canvas, alters the state of the turtle, or both.

For example, consider the example in L-system 2.1. This D0L-system has the axiom ω and the two productions p_1 and p_2. The + and − symbol are interpreted by the turtle as changing its state by rotating itself around a specific axis by the angle δ. Since we have a two-dimensional L-system in this case, + means rotate counter-clockwise by 45° and − means rotate clockwise by 45°. The [and] symbols respectively, are push and pop operations for the turtle state. In other words, [saves the turtles current position and orientation to the top of a separate stack, and] overloads the turtles current position and orientation with the values from the top of the stack. The variables F and X simply means draw forward from the current position and in the current direction. In Figure 2.6 you can see the turtle drawings of the different iterations of the L-system. For more two-dimensional L-system examples, refer to Appendix B.

$$
\begin{aligned}
\delta &= 45^\circ \\
\omega &: X \\
p_1 &: X \rightarrow F + F[X] - F - F[-X] + F[-X] + X \\
p_2 &: F \rightarrow FFF
\end{aligned}
$$

L-system 2.1: Example of a D0L-system with axiom X and two productions p_1 and p_2. The rules are applied to the generation string (initially the axiom) linearly each iteration. The generation string can then be interpreted by the turtle.

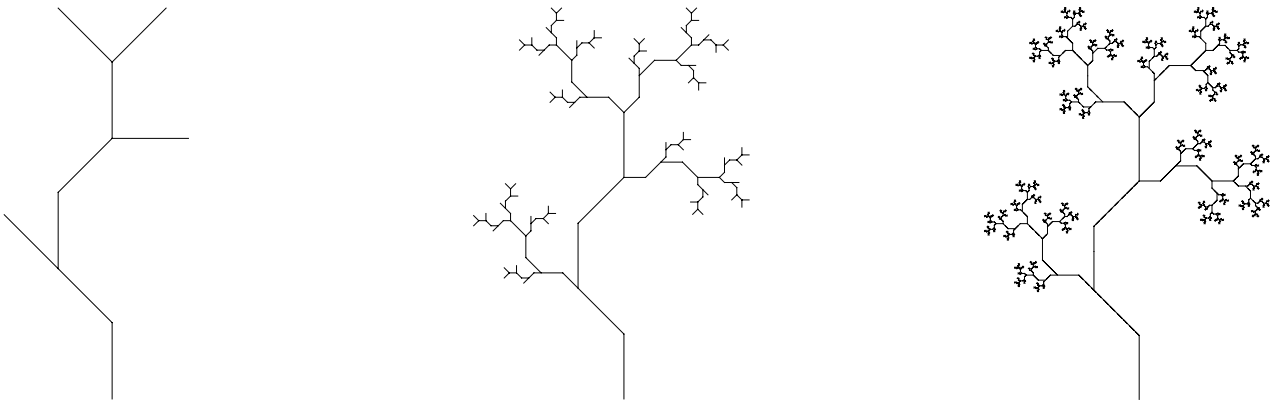


Figure 2.6: A D0L-system represented with turtle graphics. These are iterations 1, 3 and 5 of L-System 2.1.

The turtle interpreter can be expanded to draw in 3D space instead of on a flat canvas. Thus the state of the turtle has to be re-imagined. The position is trivial, since it only requires adding one extra coordinate. The orientation is however slightly more complicated. We define its orientation by keeping track of three vectors, that points in the turtles left ($\vec{L}$) , heading ($\vec{H}$) and up ($\vec{U}$) directions, respectively [32]. Together they form the turtles orientation matrix $\begin{bmatrix} \vec{L} & \vec{H} & \vec{U} \end{bmatrix}$ (illustrated in Figure 2.7).

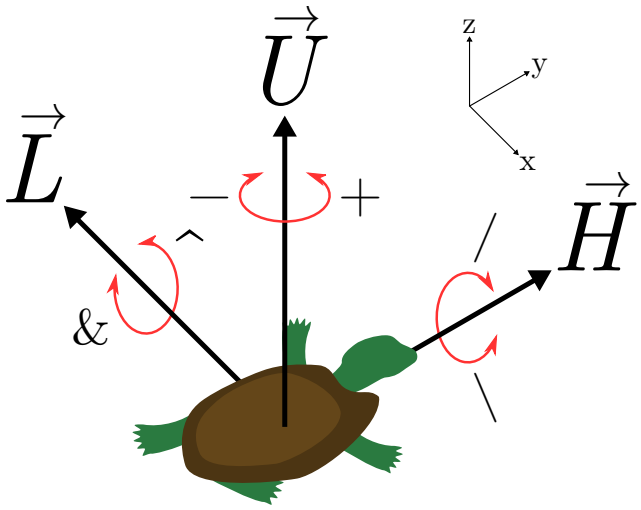


Figure 2.7: The orientation of the turtle is controlled by the symbols associated with a given direction vector $\vec{L}$, $\vec{H}$ or $\vec{U}$.

The turtle can be rotated by performing the matrix multiplication $\left[\vec{L}'\ \vec{H}'\ \vec{U}'\right] = \left[\vec{L}\ \vec{H}\ \vec{U}\right]\mathbf{R}$, where $\mathbf{R}$ is one of the following rotation matrices:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \tag{2.3}$$

$$\mathbf{R}_y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix} \tag{2.4}$$

$$\mathbf{R}_z(\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

By having one extra dimension, the number of ways the turtle can be controlled increases. Below are the definitions of the symbols [33] in Figure 2.7:

$+$: Turn left by angle δ, using the rotation matrix $\mathbf{R}_z(\delta)$.
$-$: Turn right by angle δ, using the rotation matrix $\mathbf{R}_z(-\delta)$.
$\backslash$: Roll left by angle δ, using the rotation matrix $\mathbf{R}_y(\delta)$.
$/$: Roll right by angle δ, using the rotation matrix $\mathbf{R}_y(-\delta)$.
& : Pitch down by angle δ, using the rotation matrix $\mathbf{R}_x(\delta)$.
^ : Pitch up by angle δ, using the rotation matrix $\mathbf{R}_x(-\delta)$.

2.7 Random distributions

Most game environments need to be populated by various objects. Which objects that are needed depend on the environment, for example a forest has trees and grass. Nevertheless, some method of placing these objects is required. This is usually done by a level designer, who often manually places each object. It is also possible however, to use a procedural approach.

The most basic way of procedurally producing coordinates, or points, is using random distribution. However, purely random distributions will usually have clumped points or empty spaces (Figure 2.8). For a natural placement of objects, such as trees, the desired result is an evenly spaced distribution. Two popular methods of producing spaced distributions are *random scattering* and *Poisson disc sampling*.

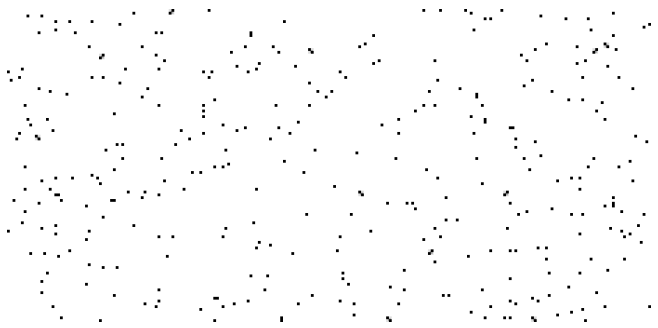


Figure 2.8: Random distribution of points. Randomly distributed points tend to group in unnatural ways, as can be seen in the figure, which in many cases is unsuitable for realistic object placement.

2.7.1 Random scattering

Random scattering can be described as placing points in an ordered structure relatively close to the desired result, for example in a square or hexagonal grid. Each point is then moved a random distance within an allowed area. In the case of a square grid the allowed area would be a grid cell. This method produces a collection of points which are closer to a purely random distribution, but with restraints on minimum distance between points (Figure 2.9).

2.7.2 Poisson disc distribution

Poisson disc sampling works by placing out an initial starting point and then start placing new points around it. The newly placed points are then used as starting points to place the next batch of points. New points are placed within the interval $[r, 2r]$, where r is the distance from their starting point. Another restriction on new points is that they can only be placed if they are no closer than r from all other points. The result is a collection of points which are no closer than r and no further than $2r$ to their surrounding points (Figure 2.10).

An example where Poisson disc sampling is useful is in the simulation of a patch of foliage. Each plant needs to be a certain minimum distance from all other plants for a plant to thrive. In the case of a forest with trees the minimum distance could be the size of each trees crown,

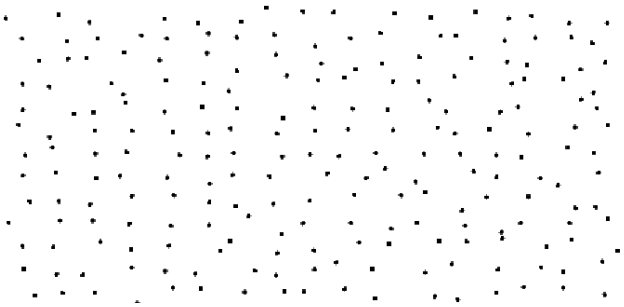


Figure 2.9: Random-scattered points. Each point is placed in a square grid and then moved randomly within the square it is placed. The result is still somewhat grid like as some columns can still be discerned.

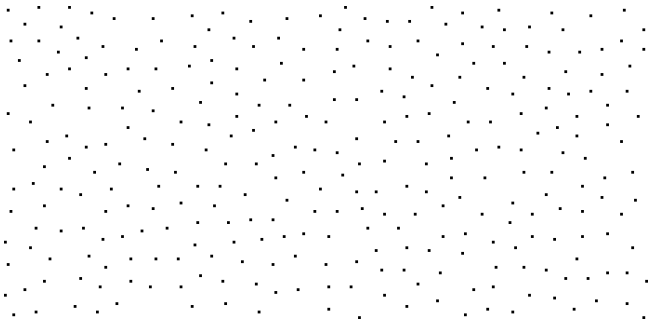


Figure 2.10: Poisson distributed points. Characterized by each point being at a minimum distance from every other point, creating a packed yet spaced distribution.

so that each tree does not compete for sunlight. Other plants could perhaps be based on water availability to simulate a natural growth pattern.

2.8 Dividing arbitrarily large worlds

It is often desirable to only generate a subsection of the world, as the generated world can be arbitrarily large and thus potentially larger than any computer can store in memory. Generating a subsection of the world can also be used to increase performance and reduce loading times. In this report, the technique of only generating a "chunk" of the world is called *chunking*, although the concept have been used by previous games, such as Minecraft. In this section, research within the project about chunking will be discussed, with what was found to be the most optimal solution to meet the project goals.

2.8.1 Challenges

Even though only smaller parts of the world are generated, it should seem to the end user as if the world is endless and fully existing when the world is explored. New parts need to be generated and connected to the existing parts in a seamless way.

An issue with chunking is that one part of the world can be dependent on other parts of the

world. Let's say part A is dependent on part B. This means that before the system can generate part A, part B needs to be generated. If part B also is dependent on part A, there is a circular dependency which cannot be fulfilled, as both parts need to wait for the other before they can be generated.

2.8.2 Solutions and restrictions

To ensure continuity, the generation rules can be set up so that all parts will fit together with any other part. While possible, this creates a huge constraint on the generation process.

Another way to achieve seamless transitions between parts of the world, is to use a generation technique that is deterministic for a specific set of coordinates. Having the constraint that every part of the world needs to be generated deterministically is a challenge and limits the system, but after extensive research it was found that this is the only feasible solution. The optimal approach is often a combination of deterministic and non-deterministic techniques, where the non-deterministic techniques, e.g. simulations, are run locally on smaller world segments. For example, the large scale terrain components such as trees can be generated deterministically using noise, while smaller elements such as rocks and flowers can be generated using simulations.

When using simulations second problem with chunking is encountered, which is that one part can be dependent on another part. As described above, a danger with this is that two parts can implicitly become dependent on each other, creating a circular dependency. One method for avoiding this is to have a system where the simulations are categorized in different levels, and where one simulation can only be dependent on data from simulations below its own level. These dependencies can be modelled as a dependency graph, where chunks can have dependencies to earlier simulations in adjacent chunks. As an example, to determine what climate a chunk has, moisture data from adjacent chunks needs to be extracted. Since moisture is one level below climate, this is consistent with the rules.

In order for the world to be deterministically generated, all of the dependencies need to be accounted for and calculated. To achieve this, there has to be a constraint that limits the area of influence of every simulation. Without this rule, the whole world needs to be evaluated in order to find out if there are any simulations that will influence the part that is currently being generated. As the system should be able to handle arbitrarily large worlds, this obviously does not work. This limits the use of large scale simulations, for example creating tunnels using Perlin worms, and to combat this problem there needs to be a trade-off between simulation freedom and the number of dependencies that have to be evaluated.

When combining deterministic techniques with simulations that have these constrains (limited area of influence and no circular dependencies), it is possible to ensure that the world can be generated using chunking. Each part will fit seamlessly with adjacent parts, and while there are a lot of things that influence generation time, there will be only a finite number of dependencies that needs to be evaluated.

3 Execution

In this chapter we present the different steps taken during the different parts of the project. The decisions and methods are based on the project goals and the topics discussed in Chapter 2. However, not all the tasks that were planned for made it into the final product. Secondary tasks like LOD and textures have not been examined as closely as the main tasks. All API in the PCG engine have been written in C++, making it accessible on multiple platforms.

The engine structure was discussed early in the project and the model that was brought forward was a core pipeline, that could have interchangeable modules attached to it. The modules can specify any type of generation that is of interest. The pipeline then handles the output of a module and passes it along to the next module as input (Figure 3.1). The sample modules which have been implemented in this project include the generation of landmasses, oceans, lakes, biomes, trees and placement of game objects.

There are some concrete steps involved in the world generation process. These are divided into four stages: *landmass generation*, *terrain generation*, *feature generation* and *object placement*.

The landmass stage generates a large-scale 2D representation of the world. Mountains, valleys, oceans and other large areas are specified with a rough base height. Additionally, data about the biomes such as plains, forests, desert, etc. is also provided in this stage. The terrain stage then uses this data as a reference to generate a detailed 3D world. Furthermore, it blends biomes together using different interpolation techniques.

The feature stage generates all detailed assets that may be found in the game world. These include trees in the forest biome, large rocks in the desert, etc. The last stage that has been implemented in this project is object placement, which is responsible for placing the features in the game world, depending on what type of biome is used.

Due to the modular structure of the implementation, it was natural to divide the project team into independent subgroups, each assigned to a specific module. The groups were responsible for their own modules, and made the major decisions. Nevertheless, the groups have worked together to give each other feedback and solve project-wide problems.

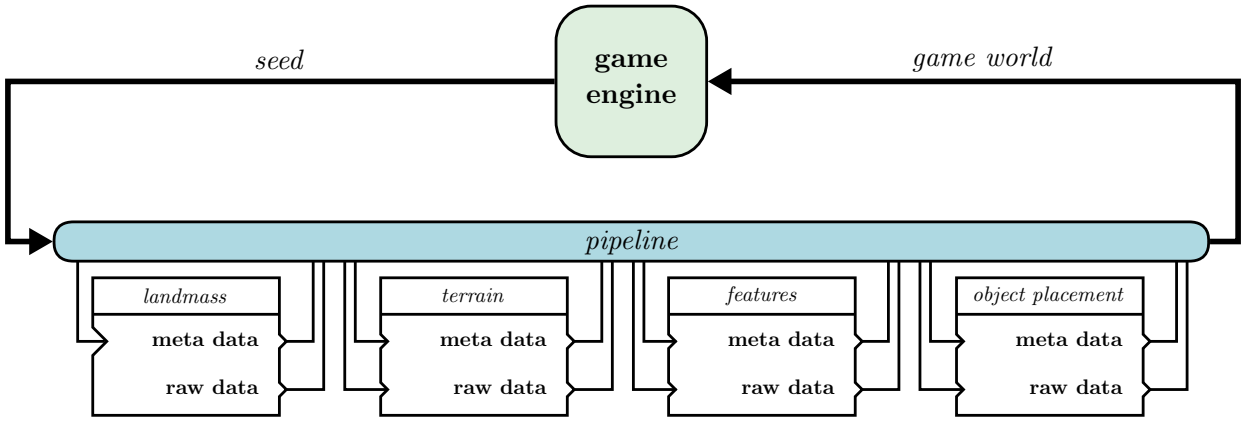


Figure 3.1: An overview of the PCG engine. The game engine is connected via a dedicated interface bridge. The world generation is initialised by a seed, which is provided by the connected game engine. The seed is then transformed in all the modules into different types of data, until it arrives to the end of the pipeline. Finally, the data is converted into a suitable data format for the connected game engine.

3.1 Core structure

We created an initial plan of building a pipeline based on tiled bitmaps, where each stage in the pipeline generates data that is drawn to one bitmap with a specific scale. The stages after would then be able to use this data as input, without being able to read the data written at their own level.

After implementing a simple pipeline in this way using bitmaps, we found out about another way of building the base terrain using Voronoi diagrams. Since using diagrams makes it much easier to generate certain large-scale features such as rivers through walking the graph, we decided to change the implementation to use Voronoi diagrams instead.

The initial plan was also to create a simple interface from the framework to game engine Unreal Engine 4 (UE4). This turned out to have some difficulties, mostly due to UE4 not really being compatible with endless worlds, and its shader system. Because of this, we decided to focus more on creating the pipeline itself, and wait with integrating it to a game engine.

3.2 Landmass module

The first stage in the pipeline generates the base shape of the terrain through Voronoi diagrams (Section 2.5). However, since there were problems with the implementation, we resorted to use the *boost* library [34] instead, which provides a broad spectrum of functionality, including Voronoi diagram generation. This helped to get results on the landmass generation quicker and to move on to other tasks.

In this section, it will be discussed how Voronoi diagrams are used for arbitrarily large game worlds, by splitting up the graphs into chunks. These chunks are able to generate their part

almost independently, with only a little bit of information from the other chunks, making it possible to generate just a subset of the game world.

3.2.1 Chunking Voronoi diagrams

As discussed in Section 2.8, every part of the pipeline has to be made in steps that can be split into chunks. However, Voronoi diagram does not have a natural way to be divided, which makes chunking a bit more difficult.

To ensure that a chunk of a Voronoi diagram is accurate, that chunk is marked as an *area of interest*. The accuracy of the chunk generation can be guaranteed by including every site, such that the polygons in the area of interest provide sufficient information to generate that chunk. In the general case, finding which sites needs to be included to guarantee accuracy requires all sites to be generated and compared.

Comparison between the sites can be eliminated completely by restricting the way they are generated. By dividing the world into a square grid, each cell in the grid can be assigned a site generator. Each generator can only generate sites that reside in the area of its assigned grid cell (Figure 3.2). Ensuring that each generator generates at least one site makes it possible to calculate which generators has to be included to guarantee that the area of interest is accurate. Therefore, the maximum distance between a site and a point on the border of the cell of width w is $\sqrt{2} \times w$, i. e. when a site resides in one of the corners.

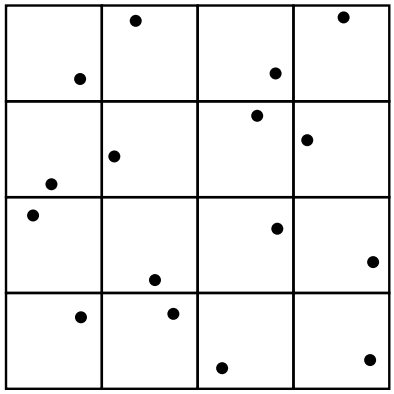


Figure 3.2: An example of restricted site generation. One cell must contain a single site, where the site can be anywhere in the cell.

Since each generator generates at least one site, the accuracy of the area of interest can be guaranteed by including every generator such that their distance to the area of interest is less than or equal to $\sqrt{2} \times w$, i. e. every generator inside the area of interest and those two steps away should be included. The number of sites that are included but won't affect the diagram can be decreased by filtering out the generated sites that are further away than $\sqrt{2} \times w$ from the area of interest. By using even stricter rules on the site distribution this distance can be reduced significantly, further explored in Section 3.2.5.

3.2.2 Using multiple Voronoi diagrams

The first idea for defining terrain was to use multiple Voronoi diagrams and interconnect them during graph traversal. Each diagram is associated with one grid cell. Each cell is considered a separate area of interest and sites are included in the diagram accordingly, i. e. sites from the generator associated with the cell and each cell less than $\sqrt{2}$ number of cells away. As mentioned this distance can be reduced so that only the direct neighbours need to be included.

This approach has a great advantage in that adding and removing grid cells does not depend on the rest of the world and thus has a constant time complexity per cell. Another benefit

is that the diagrams can be created using methods that don't allow for insertions after the diagram is created. Deletion is a constant operation; just remove the diagram associated with the generator.

3.2.3 Using a single Voronoi diagram

Another idea for defining terrain was to have the whole terrain defined a single Voronoi diagram. The necessary sites would then be added and removed as needed for the area of interest to be accurate. As Takao et al. [35] have shown this technique could lead to an average time complexity of $\mathcal{O}(n)$ for adding one site.

The area of interest can be extended and shrunk by adding or removing all sites for a grid cell. Adding multiple sites at the same time has potential for faster insertion times as the seek time can be reduced. As this approach was not the one we choose speed improvements were not further investigated. Nevertheless the sites can be added one by one, with an average time complexity $\mathcal{O}((n+k)k)$ where n is the number of points already in the diagram, and k is the number of points to be added per grid cell.

Another important aspect is whether parts of the structure can be discarded when they're not in view anymore. Being able to unload parts of the structure is important to free memory and to reduce work. Mostafavi et al. [36] shows that a point can be deleted from the Voronoi diagram in $\mathcal{O}(n)$, where the complexity stems from searching the diagram for the point to remove. As the sites in one grid cell will be interconnected in the diagram there is no need for more searches when the first point to be removed is found. Deletion of all the sites in a grid cell can therefore be done in $\mathcal{O}(n)$.

3.2.4 Voronoi diagram indexing

Graph traversal when using multiple Voronoi diagrams, as discussed in Section 3.2.2, requires a method to traverse between diagram neighbours and in effect the whole graph. This method needs to detect when the traversal should switch to a graph and find where to continue the traversal once the switch has been made. A method for performing these operations without any special representation of the graph will be presented first, followed by an alternative storage representation with improved speed.

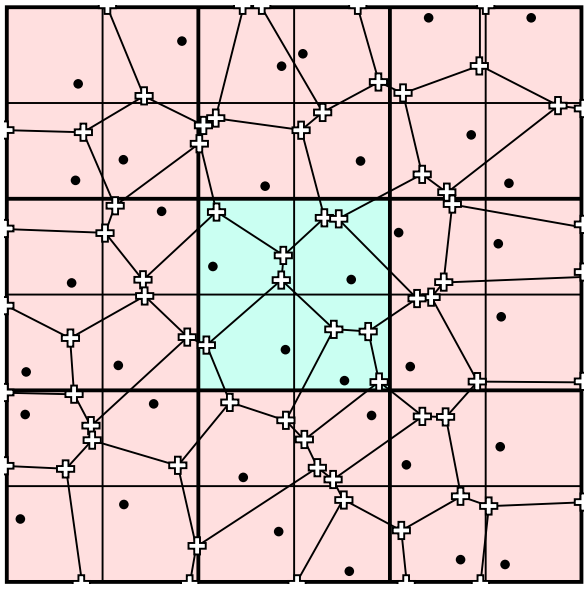
Detecting which graph to traverse when traversing vertices is as simple as calculating which grid cell the vertex is in. Traversing is done in the same way as for the cells in the Voronoi diagram, but instead the sites are used. Edges can be traversed similarly but need a tie breaker when its two vertices are located in different grid cells. Implementing a strict order on the vertices a and b, e. g. $x_a < x_b$ and $y_a < y_b$, and using only the minimum or maximum point of the edge breaks the tie.

Neighbouring graphs don't share any data structures and traversal can therefore not be immediately continued once a switch between graphs has been made. Considering a switch from graph A to graph B, to continue traversal in B, the current edge or vertex in A has to be found B. In the case of the vertices v_A in A and v_B in B, finding v_A in B is done by searching linearly among the vertices of B, such that v_A and v_B have the same position. In the case of the edges e_A in A and e_B in B, finding e_A in B is done by searching linearly among the edges

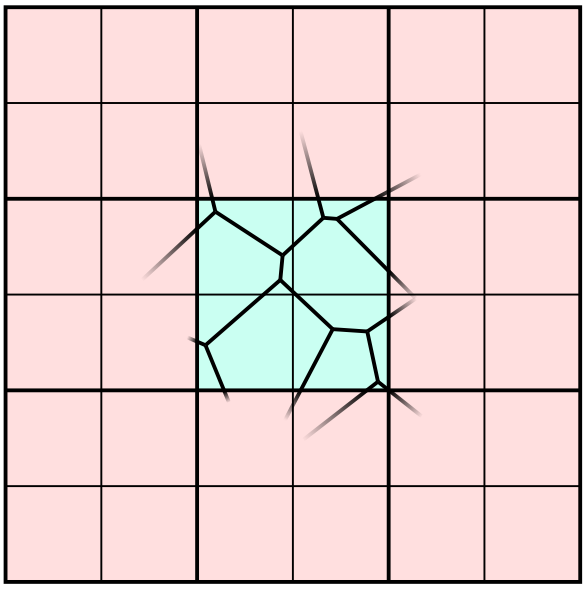
of B, such that the vertices of e_A and e_B are in the same positions, respectively. Since vertex coordinates are stored as floating-point numbers, the appropriate error marginal has to be used when comparing the coordinates.

We define a *graph element* as either a cell, edge, or vertex. A storage format for interconnected Voronoi diagrams needs to assign ownership to a diagram for each graph element, so that shared graph elements are not duplicated, which would otherwise make it difficult to uniquely assign metadata. The storage format also needs to not be circularly dependent on the diagrams neighbours. Assigning ownership is made in the same way as presented for traversal and references to graph elements need to convey this ownership.

Constructing a connected Voronoi diagram without creating circular dependencies is a multi-step process. The algorithm models this as a state-machine with three stages: *buildVertices*, *buildEdges*, and *connectEdges* (Figure 3.3). The output of the final step, *connectEdges*, is a grid cell with a Voronoi diagram that can reference the vertices and edges of its neighbours. *buildEdges* creates edges which references the vertices of the grid cell and its neighbours. The data structure stores which edges are in each Voronoi cell and vertex, but *buildEdges* can only add references to edges that belong to this grid cell. Edges which could not be referenced are stored and the references are later added by *connectEdges* which has the neighbour edges available. *buildVertices* creates the unconnected Voronoi diagram and stores every vertex that belong to the grid cell. A pseudo-code program can be seen in Appendix C.



(a) Here the *buildVertices* and *buildEdges* stages are shown. The former calculates the vertices (white crosses) of the Voronoi diagram cluster around the green grid cell, based on the generated, cell restricted sites (black dots). The latter calculates how edges are connected between the vertices.



(b) The *connectEdges* stage is responsible for selecting and storing only the edges that are used by the Voronoi diagram in the green grid cell.

Figure 3.3: The three stages of the algorithm for connecting multiple Voronoi diagrams with each other.

The algorithm uses a simple data structure, inspired by Alumbaugh and Jiao [37], which stores the relations as indices, which include which grid cell the item can be found in and at what index. This allows for unloading of grid cells, as recreation of the grid cell will produce the same indices and neighbour references, and will therefore still be correct. It does not store edge direction, but direction can be added easily by a simple modification or by encoding it in the metadata.

3.2.5 Site generation

Having well distributed sites is a requirement to create good looking Voronoi diagrams. The goal is to create a diagram without recognizable patterns and without artifacts like Voronoi cells that look like a square or are too small. Patel [38] recognizes that random numbers have a tendency to clump up, creating artifacts, and recommends using Lloyd relaxation to decrease the discrepancy. We evaluated Lloyd relaxation but did not find the results completely satisfactory. The diagram was either considered too irregular and with artifacts, or too regular without variation. Lloyd relaxation is also a slow process, requiring the Voronoi diagram to be recreated multiple times.

Satisfactory results were found by adding random scattering to a structure looking close to the sought after shape. A hexagonal grid is a good base as it has a shape that feels organic and is easy to create. We found that the random scattering has to spread the sites enough to create cells with different amount of edges, as it is very easy to spot a pattern otherwise. However, the random scattering has to be limited so that no sites are generated to close to each other, which would create artifacts. A uniform random distribution within the range $[-\text{hexcellsize}/3, \text{hexcellsize}/3]$ was found satisfactory. The distribution of sites is very controlled, lowering the maximum distance needed in Section 3.2.1 to $1\frac{1}{3} \times$ hexcellsize. For examples of the different site generation methods, see Appendix D.

3.2.6 Large scale features

One of the goals of this project was to explore what methods can be used to create large scale features such as mountains and long rivers. Voronoi diagrams were early identified as the primary tool we wanted to investigate, inspired by the work of Patel [38]. Patel demonstrates a small island with pleasant rivers created by selecting random vectors on mountain corners as the river starting point. The river is created by recursively following the edge with the biggest downward slope. Patel defines height as the distance from the coast leading to an island with a mountain in the middle, and gives each vertex the property that the following the edge with the most slope will always lead to the ocean.

Our implementation uses coherent noise which does not have this property, making river creation more problematic. Using Patel's method in our system leads to a situation where small valleys will trap the river, making it stop and form a lake too early. A broader search for river candidates can be performed, but this will produce rivers that flows in unexpected directions given there are no modifications to the landscape. A simulation based approach has the potential to create the best results as the world can be modified to fit around the rivers. Allowing for modifications near grid cell borders creates unaligned transitions between grid cells, since

the simulations in each grid cell can't be guaranteed to modify the terrain in the same way at a certain point.

Creating a protected area around grid cell borders where modifications are toned down and eventually eliminated, gives the simulation complete control where modifications are safe to perform, and enables a smooth transition between grid cells. The protected area will unfortunately not be able to contain rivers that cross the cell border, creating a dry patch around every grid cell.

To allow for simulations that cross grid cell borders, a pipeline step could be added which proposes world modifications for the grid cell and its neighbours. An integration step which merges the proposed changes would make the final decision. By having the constraint that a grid cell only can influence the closest part of neighbouring grid cells, the area of proposed modifications from several grid cells won't overlap, limiting the simulations area of influence.

The final technique using simulations that was considered takes inspiration from the LOD concept. Mutual dependencies between neighbours are resolved by allowing dependencies to be unresolved when outside the area of interest. Unresolved dependencies lead to temporarily inaccurate results, which shouldn't lead to problems as long as the inaccuracy is contained outside the area of interest. Containing the inaccuracy outside the area of interest requires the same dependency management as the other methods. Simulations that are both inherently mutually dependent between neighbours and can have their dependencies easily computed are a rare case, making it a niche solution.

3.3 Combining landscapes

Landscape interpolation refers to seamless transitions from one landscape type to another, without any artifacts or sharp edges (unless that is desired). Several interpolation techniques were evaluated, which are covered in this section.

3.3.1 Biome interpolation

The output of the landmass stage consists of a tiled Voronoi diagram with attributes in each vertex and edge. These attributes represent world data such as height and landscape type. In order to generate voxel chunks with a smooth transition between the different landscape types, we need to represent the attribute set as a continuous function defined on each point in the world.

There are multiple ways of evaluating this function. One way is by triangulating the cells in the Voronoi diagram, then sampling the vertex attributes of the closest triangle at each position through interpolation using the barycentric coordinates. However, there are multiple problems with this approach: the triangulation is expensive, and the sampling would result in visible interpolation slopes. Instead, we decided to use a vertex-based interpolation. This is performed by calculating the vertex closest to the center of each voxel chunk, and then interpolating the attributes of that vertex and its direct neighbours based on the position of each voxel column. The resulting constants for each voxel are used as input to the terrain generator.

3.3.2 Voxel Interpolation

Since the project uses voxels to create landscapes, the generator uses different systems to determine what type of voxel should be in all of the world coordinates. To achieve a seamless transition between two such systems, the voxel type evaluation needs to be interpolated. In order to perform this interpolation, all of the systems that handles voxel evaluation needs to share either input data or output data. While developing the terrain module, the team tried several different techniques for interpolating the voxel evaluation.

The first method that was evaluated was to have all voxel placement systems share parameters. When the system interpolates between two biomes, all of these parameters are interpolated, which results in a biome that is a mix between the two interpolated ones. The downside with this technique is that if two biomes should be interpolated, they need to share parameters. This is a big constraint as it limits the system to one type of voxel evaluation interface, if all of the biomes should be able to be interpolated. The alternative is to use different interfaces, and accept the fact that some biomes can not be interpolated with each other.

Secondly, we tried using voxel density interpolation. Instead of interpolating the parameters, the method interpolates the output data. This technique works very well with systems that naturally produce a value representing the voxel density as output, such as 3D noise. Some of the systems, however, simply produces output data as `boolean` values, that is if a voxel should be placed at some coordinates or not.

The last evaluated method is a variation of density interpolation, but instead of density this method outputs a height value, which then is interpolated. The main disadvantage with this method is that it requires the voxel evaluation algorithm to produce landscapes with a single height value, such as 2D noise.

The team weighted the advantages and drawbacks of all these methods, and came to the conclusion that density interpolation was the best option. To solve the problem that some systems did not produce density data, we modified these systems to produce "fake" density. For example, normally 2D noise only produces `boolean` data. In order to fix this we modified these systems to output a density, represented as a `float` between 0 and 1, depending on the coordinates relation to the height.

3.4 Terrain module

The terrain module is responsible for converting the previously generated 2D-data (such as height and biome information) to an actual terrain. The following sections will bring up the successive versions of the module, where each iteration focused on tackling a specific problem (such as voxelisation, chunking, interpolation).

Another important part of the terrain generation process was to design algorithms which produced interesting noise. In other words finding ways to combine, modify and control noise to produce the desired results. While there is a lot of thought and logic behind most methods, in the end it involved a fair amount of trial and error to fine-tune all the parameters. Some of the generated terrains, together with a short explanation of the algorithm, can be found in Appendix A.

3.4.1 Generating a terrain surface

The first milestone in the terrain generation was to generate and visualise a 2D surface, in other words a heightmap. Because the pipeline was still under construction we created a simple test program to simulate input data (base height and biome information) and render the terrain (using Matlab).

To generate interesting terrain details, we needed access to noise functions. After some research we decided to use simplex noise. We found an open domain C++ implementation of simplex noise based on the Java version created by Stefan Gustavson [23]. It was extended with additional functionality such as octave noise, ridge noise and, most importantly, the ability to choose a seed.

Once we had access to working simplex noise it was extremely straightforward to create different terrains. The following method was used to calculate the surface:

$$\text{actual height} = \text{base height} + \text{noise offset}(x, y) \tag{3.1}$$

Figure 3.4 shows two different terrain types that were generated using this method.

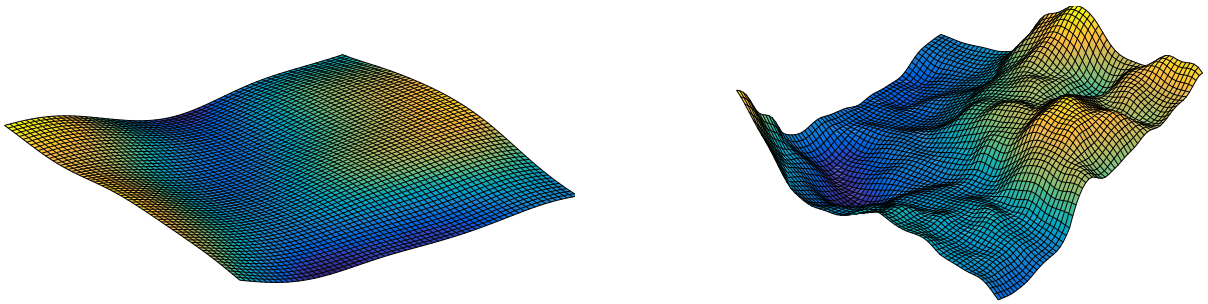


Figure 3.4: Two different surfaces that have been generated with simplex noise. The terrain to the left, which uses a lower frequency and fewer octaves, could be used for plains. The right one, which uses a higher frequency and more octaves, is more suitable for hills or mountains.

This system was easily extendable to an arbitrary amount of chunks. In the simplest case, where each chunk had the same base height, the boundaries between chunks is completely indistinguishable (Figure 3.5). The reason for this is that simplex noise is continuous in the x and y coordinate, so as long as each height is calculated using its position in the world, not locally on the chunk, the terrain will be continuous. In this case we simply decided that origin of the world was in the corner of chunk (0,0). This allowed us to calculate the world position of a point on chunk (a,b) with the following formula:

$$(\text{world x}, \text{world y}) = (a \cdot \text{chunk width} + \text{local x}, b \cdot \text{chunk width} + \text{local y}) \tag{3.2}$$

In the case where chunks had different base height values, there was a clear border between chunks (Figure 3.6a). To solve this problem, we calculated a new base height for each point by interpolating the base heights of the surrounding chunks using bilinear interpolation (Figure 3.6b). While the interpolated case looked much better, there were still some small artifacts. This effect was much less visible with more varied terrains. Since we decided that the height interpolation should be handled by the pipeline, this method was deemed satisfactory for testing purposes.

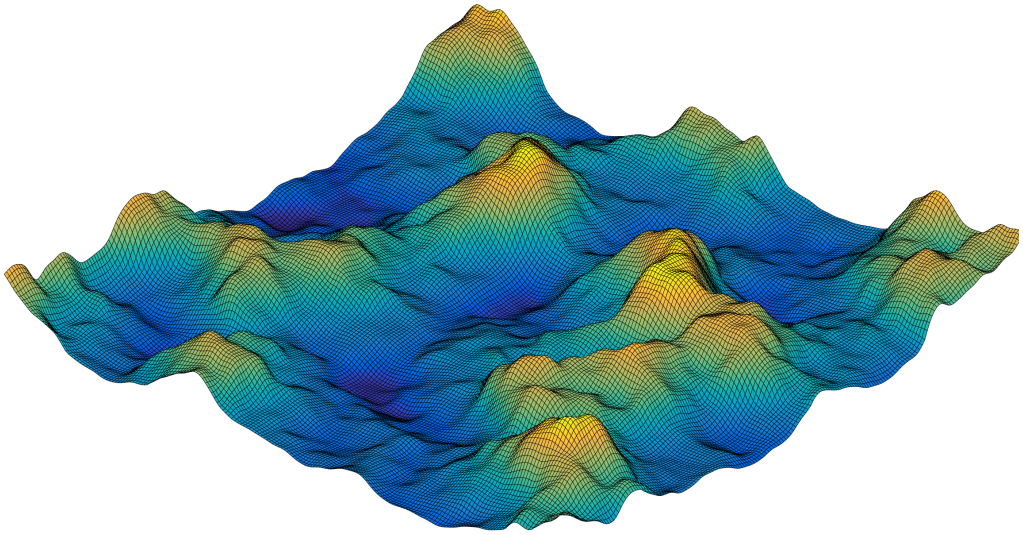
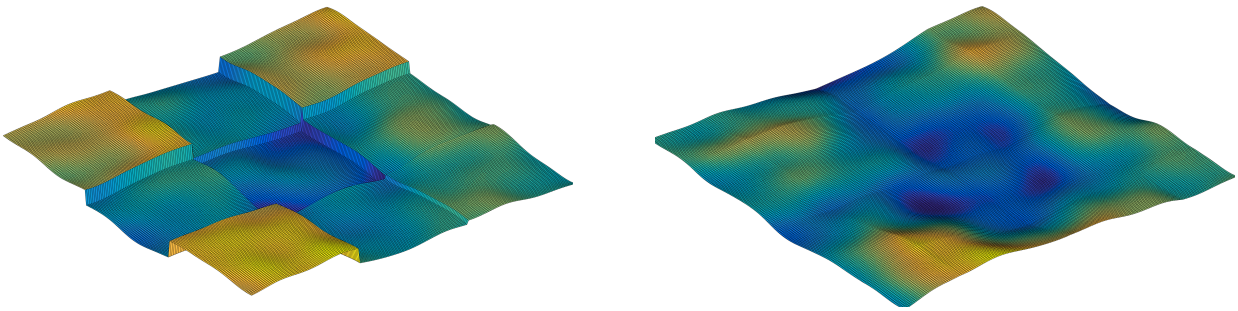


Figure 3.5: A terrain that is composed of 9 chunks arranged in a square (3×3). Since the noise functions are continuous across the chunk borders the resulting surface is seamless. Note that this is a larger rendering of the hills presented in Figure 3.4.



(a) Without interpolation. **(b)** With interpolation.

Figure 3.6: Demonstration of the effects of height interpolation. Both terrains use exactly the same chunk data and noise functions (the plains function from Figure 3.4). The only difference is that the right example uses height interpolation while the left one doesn't.

3.4.2 Generating and visualising voxel terrain

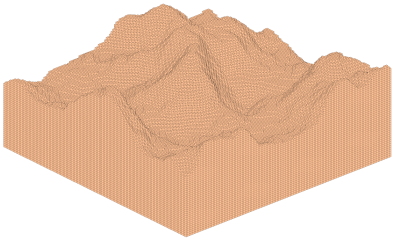
Up to this point we had only generated 2D surfaces. However, as stated earlier, the goal was to generate voxel terrain with details that were impossible to represent with a heightmap. Therefore, the next milestone was generating and visualising voxel terrain. The input data remained exactly the same, but the output was now a 3D array of values which was once again rendered using Matlab.

The terrain generation module needed some modifications to work with voxels. Instead of a heightmap it populated a 3D array of `boolean` values where `true` represented ground and

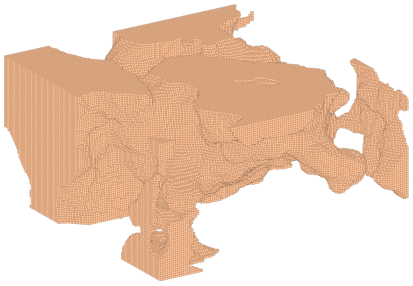
`false` represented air. To determine the value of each voxel a terrain generation function was called.

Even though the terrain was voxelized it was possible to use the heightmap methods developed earlier. For each point we simply checked whether it was above the calculated height. Ff it was we set it to air, otherwise it was set to ground (Figure 3.7a).

The true usefulness of voxels is however much more apparent when 3D noise is used in the generation. In this case the noise function is used as a density, so a point becomes solid if the noise is above (or below) a certain threshold (Figure 3.7b).



(a) Terrain generated using 2D simplex noise. In essence, the first step was to generate a heightmap where each point represented a voxel column. Then, all blocks below the calculated surface height were set to solid and the rest to air.



(b) Terrain generated using 3D simplex noise. The value in each point was treated as a density and only the voxels above a threshold value were set to solid. Notice the complex geometry (such as overhangs) which would be impossible to represent with a heightmap.

Figure 3.7: Voxel terrain using different simplex noise types.

3.4.3 Generating terrain from biome attributes

Combining different biomes had been an recurring problem during the project as described in Section 3.3. The final solution that was agreed upon was that the biome in a point is entirely described by a set of continuous attributes. This required a fundamental change to the terrain module because it was no longer possible to completely separate the generation of different biomes (since all possible attribute combinations were technically possible). Instead, a single function, which we referred to as the *world biome function* (WBF), generated all the terrain.

This forced us to rethink the way in which voxel types were assigned. Previously, the only requirement on a generator function was to be able to assign a type to the voxel which it was evaluating. This meant that a generators method of assigning voxels was could be entirely arbitrary. When the WBF was introduced, it took over the responsibility of assigning voxel types. Therefore, the previously developed terrain generation functions had to be modified to be able to co-exist. Simply keeping them as they were and trying to interpolate their output voxel types was impossible since the continuity of the underlying functions (which is crucial for good interpolation) was lost in the conversion to a voxel type.

Instead we needed a way to combine the different generators before actually evaluating a voxel

type. Fortunately, we realised that all of our previously created generator functions could be categorised as either density-type (where they evaluated voxel type based on a generated density) or heightmap-type (where they evaluated voxel type based on a generated surface height). Furthermore, combining same-type generators proved straightforward.

heightmaps could be combined with the following formula:

$$H(x, y) = H_B(x, y) + p_1 \cdot h_1(x, y) + p_2 \cdot h_2(x, y) + \dots \tag{3.3}$$

where H is the final surface height, H_B is the base height, p_i is the strength of biome attributes i and h_i is the height offset function that is associated with attribute i.

Analogously, density function generators were combined using the following method:

$$D(x, y, z) = p_1 \cdot \rho_1(x, y, z) + p_2 \cdot \rho_2(x, y, z) + \dots \tag{3.4}$$

where D is the final density, p_i is the strength of biome parameter i and ρ_i is the density offset function that is associated with parameter i.

The main advantage of this approach is that the output is guaranteed to be continuous, as long as the underlying offset functions (h_i and ρ_i) generate continuous terrain. Another advantage is that the effect of one parameter can be defined independently of the other parameters. For example, lets say that $i = 1$ represents plains and $i = 2$ represent mountains. Then h_1 could be a low frequency simplex noise with a small amplitude, while h_2 could be a high frequency, high amplitude noise function (just like in Section 3.4.1). Independently, these functions generate plains and mountains, but when combined they produce something more akin to hills (Figure 3.8). The same logic can be applied to the density functions.

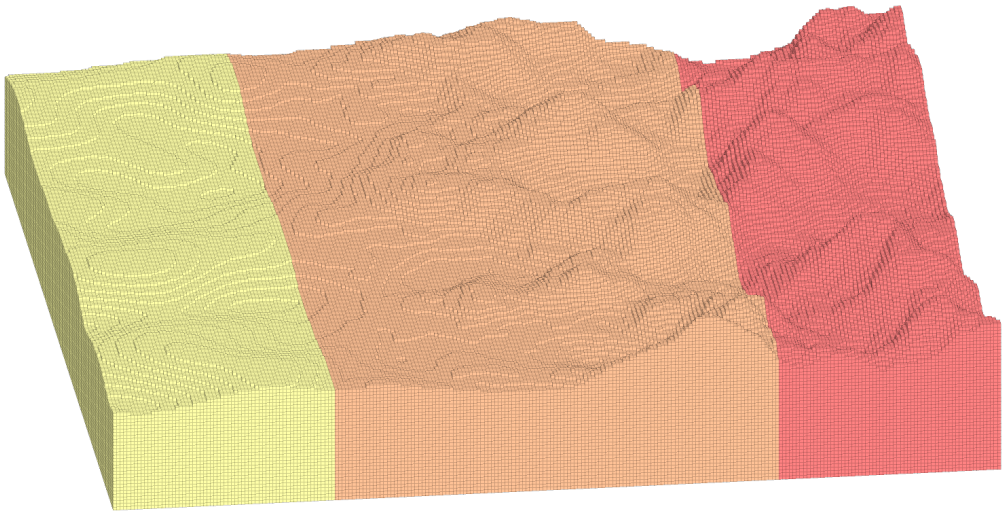


Figure 3.8: Demonstration of terrain interpolation between two different sets of biome attributes. The yellow terrain is purely plains and the red terrain is purely mountains. The orange terrain is interpolated, which results a smooth transition from plains to mountains. Note that the hills in the middle are a natural product of the system.

After finding a solution for inteprolating same type generators we needed to find a good method

of interpolating different type generators. First we attempted to convert the heightmap to a density matrix before the voxelisation, which could then be combined with the density matrix generated by the density functions. So the first step was to calculate the surface height (H) with equation (3.3). Then this was converted to a density value with the following function:

$$D_s(x,y,z) = \begin{cases} 1 & \text{if} z < H(x,y) \\ 0 & \text{if} z > H(x,y) \end{cases} \tag{3.5}$$

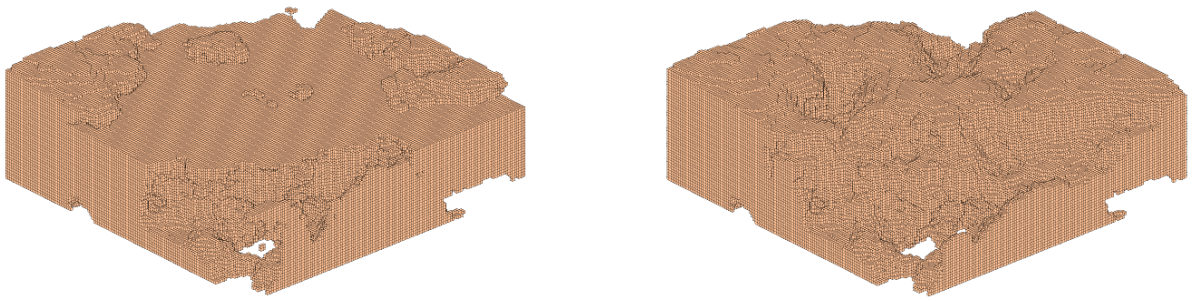
Finally this was combined with the density modification functions by using a modified version of equation (3.4):

$$D(x,y,z) = D_s(x,y,z) + p_1 \cdot \rho_1(x,y,z) + p_2 \cdot \rho_2(x,y,z) + \dots \tag{3.6}$$

While this approach managed to combine the two types reasonably well, the hard cutoff introduced by equation (3.5) was sometimes visible in the terrain. To smooth the transition from air to ground, a linear dampening was used:

$$D_s(x,y,z) = \begin{cases} 1 & \text{if } z < h(x,y) - l \\ 0.5 + \frac{h(x,y)-z}{l} & \text{if } h(x,y) - l < z < h(x,y) + l \\ 0 & \text{if } z > h(x,y) + l \end{cases} \tag{3.7}$$

where l is the distance where dampening is applied. A comparison of the same terrain with and without this smoothing is shown in Figure 3.9.



(a) Without dampening **(b)** With dampening

Figure 3.9: Comparison between two methods of heightmap-to-density conversion. For the sake of illustration the chosen heightmap is completely flat. In the left example no smoothing is used, which results in a clearly visible flat surface. The right terrain, however, uses linear dampening which results in a less unnatural terrain.

3.4.4 Removing floating voxels

When generating voxels using 3D noise there are sometimes, depending on the noise function and the parameters, voxels that "float" in the air. In some instances this might be desirable,

such as when generating alien landscapes. However, when generating more earth-like terrains, that should comply with our planets laws of physics, these floating voxels need to be removed.

One approach to removing floating voxels is inspired by Kruskal's algorithm as it divides the voxels into partitions. All voxels that are adjacent to each other are assigned to the same partition. The next step is to decide which partitions that are considered to be "connected with the ground" and deleting the rest of the voxel belonging to the rest of the partitions.

A simplified description of how to divide the voxels into partitions is as follows:

1. Assign all the voxels to its own partition.
2. Pick a random voxel.
3. Do a breadth-first search over the graph of the voxels. Voxels are considered connected with adjacent voxels.
4. For every voxel, combine its partition with all adjacent voxels.
5. Repeat steps 2-4 until all voxels has been checked. The voxels should now be divided into partitions so that all adjacent voxels are included in the same partition.

The next problem is to choose which partitions that should be kept and which that should be removed. It is safe to remove all of the partitions that doesn't have any voxels at the edge of the chunk. It is trickier to remove floating voxel clusters that are traversing several chunks.

This can be seen as a simulation problem over several chunks, which is covered in Section 2.8. As mentioned in this section, any simulation needs to have a limited area of influence. In the case of removing voxels, the system needs to have a limited number of chunks that is checked to determine if a voxel partition is part of the ground or not.

3.5 Tree module

Prior of doing any implementation of the L-system that was described in Section 2.6, some reading was done on the subject. Arguably the most common source of the underlying theory of L-systems can be found in the book *The Algorithmic Beauty of Plants* by A. Lindenmayer et al. [33]. This book covers virtually all the basic and advanced mathematics behind algorithmic patterns specifically found in plants. Much of the implementation of the module draws inspiration from the ideas presented in this book.

After having gained a fundamental understanding about modelling L-system structures, particularly the tree-like ones, the implementation efforts were straightforward. The string generation was done by simply traversing the string (initially set to the axiom of the L-system) and replacing each character with its corresponding rule. The L-system mechanics were wrapped in a `struct` type (`TreeLSystem`), allowing the ability to subclass and override the production rules. This way a new tree type could easily be created by having a convenience constructor that initialises the `TreeLSystem` parent type with the appropriate production rules, branch lengths, etc.

Some other attempts were made by replacing the `string` type of the generation, by having a dedicated type to perform the task. This idea was dropped early, since it became apparent that the `string` type had more flexibility.

The turtle was implemented as a method, which is called from a `TreeLSystem` instance. The characters with meaning, i. e. [,], +, −, \, /, &, and ^, were wrapped in an `enum` type. The push and pop operations uses a stack to save the turtle state. The rotations were done by using rotation matrices for objects in three-dimensional space. As the turtle traverses through the string, the data structure for the tree builds up by adding to it, branch by branch.

The first attempt when constructing the data structure for the trees was to make each branch a mesh, i. e. a collection of polygons. To have it rendered and tested, an interface bridge for the *SceneKit* API for the Swift and Objective-C languages was implemented. Fortunately, C++ is easily bridged to Objective-C through Objective-C++, and Swift is directly compatible with Objective-C code, so the rendering was easy to set up (Figure 3.10). Matlab was used to debug the L-system structure and render only the endpoints of the branches, making it a sort of connect-the-dots picture in 3D. The rendering verified that the L-system worked as expected, so the rules could now be tweaked to yield more realistic results.

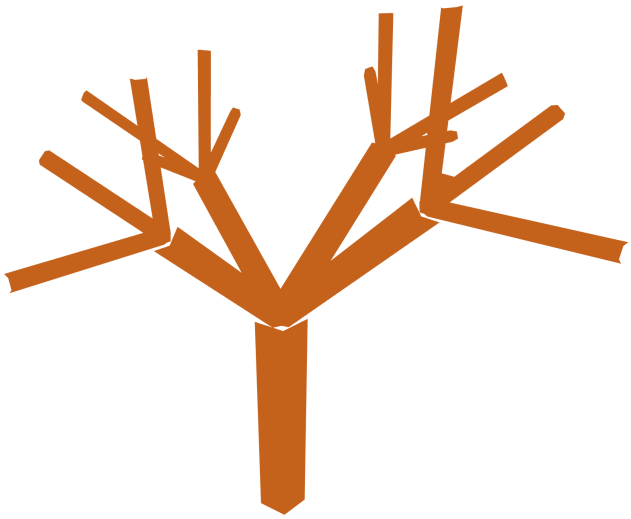


Figure 3.10: A tree rendered with a polygon mesh using the tree module.

The next attempt was to migrate from the polygon based structure of the trees and use tiny voxels instead, making it more consistent with the terrain module. Due to using voxels, some other problems appeared. The branches were approximated to cylinders, and therefore it was necessary to have a way to approximate a circle in the discrete space that voxels reside in. One algorithm to use is *Bresenham's circle algorithm*, which approximates one arc of the circle. This arc can be duplicated and rotated so that it forms a full circle. This algorithm is then applied to each slice of the cylinder.

Another problem that arose out of the use of voxels, is that they don't rotate very well. In order to make the voxels "snap" into the voxel grid, a specialised rotation operation has to be used. The approach that was used to solve this is to shift each slice using the closely related *Bresenham's line algorithm*. This algorithm is applied to the rotated axis of the branch, so that each slice of the cylinder has a corresponding point in the axis.

3.6 Object placement module

We looked at different options for placing objects, starting with finding out how some current games place objects into their worlds. The methods that we found most prevalent are random scattering and Poisson disc sampling. We considered Poisson disc sampling to be the better candidate because it produced better looking results. A more detailed description of these algorithms can be found in Section 2.7.

After researching the algorithm we found an implementation in C++ on Github [39]. The implementation worked by placing one point at a time until the space available was filled, in other words points would cover as much ground as possible. To allow for spaces where no entities are placed, we added an algoritm that we call *pruning* (Figure 3.11). Pruning works by using a so called mask which holds data about where things may and may not grow. The mask was represented using a 2D array of floating-point values values between 0 and 1. To determine whether or not an entity should be removed, a random number was generated, and if the random number was above the corresponding mask value it was removed. If the mask value was 1 there was a 100% chance of removal, and similarly if it was 0 instead there was 0% chance of removal.

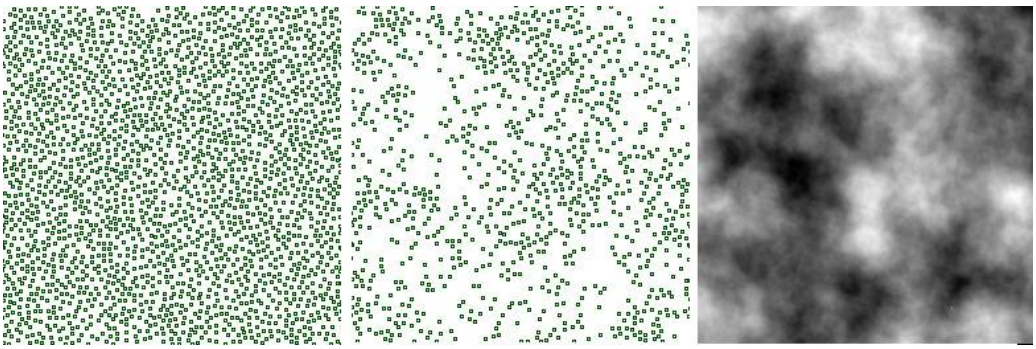


Figure 3.11: Poisson distributed points in a 256×256 area. Left image unpruned with points of radius 2. Middle image is the same as first image but after pruning using noise which can be seen in the rightmost image.

The implementation was fairly simple and needed to be modified further to suit our needs. The modifications can be summarized as giving each point more metadata. First we gave each point a type, or an ID. The IDs allowed for identification after the algorithm was finished, an example where this was useful was in graphical representation to tell each entity apart. At this stage points were instead referred to as entities. We added a minimum distance to each type of entity, instead of a global minimum distance. This allowed for different "sizes" of entities, which was a natural step to represent, for example, trees and grass, which vary greatly in size (Figure 3.12).

With added metadata the complexity of the algorithm went up. One problem we faced was allowing different entities to be at different minimum distances from each other. A good example to understand the reasoning behind this requirement would be that grass can grow under a tree but not inside the tree; the minimum distances are the tree trunk and grass size. Contrary, trees will not grow so that their crowns grow into each other; the minimum distance is the size of the tree crowns. To solve this we grouped similarly sized entities into layers. Large entities

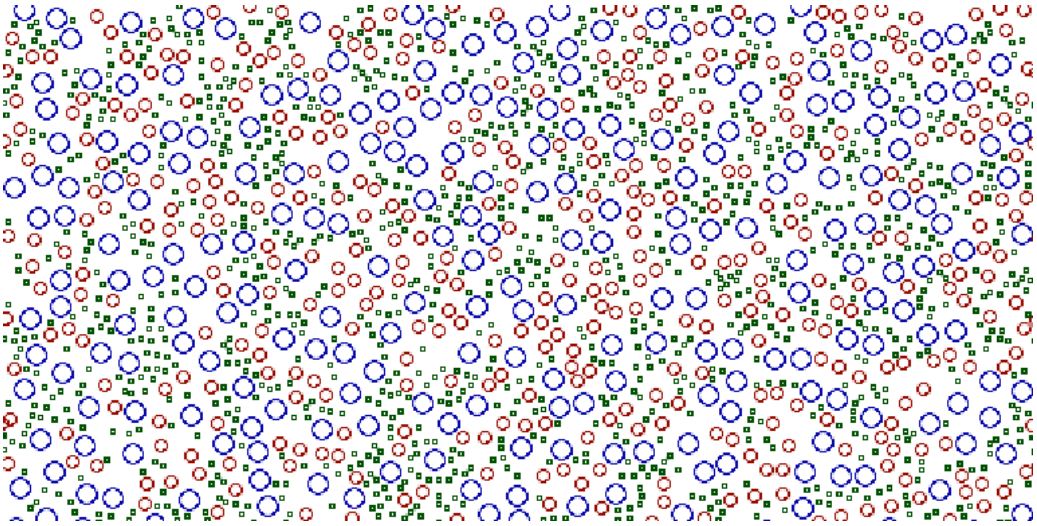


Figure 3.12: Poisson distributed entities of three different types represented in different colors and sizes. Dark green has radius 2, brown radius 4 and blue radius 6.

were placed first, and smaller entities were allowed to grow between the larger (Figure 3.13). Because entities were now placed in layers, it came naturally to prune each layer individually before starting on the next layer. This opened up for having areas with no large entities but many smaller. For a real world example this could simulate glades in a forest.

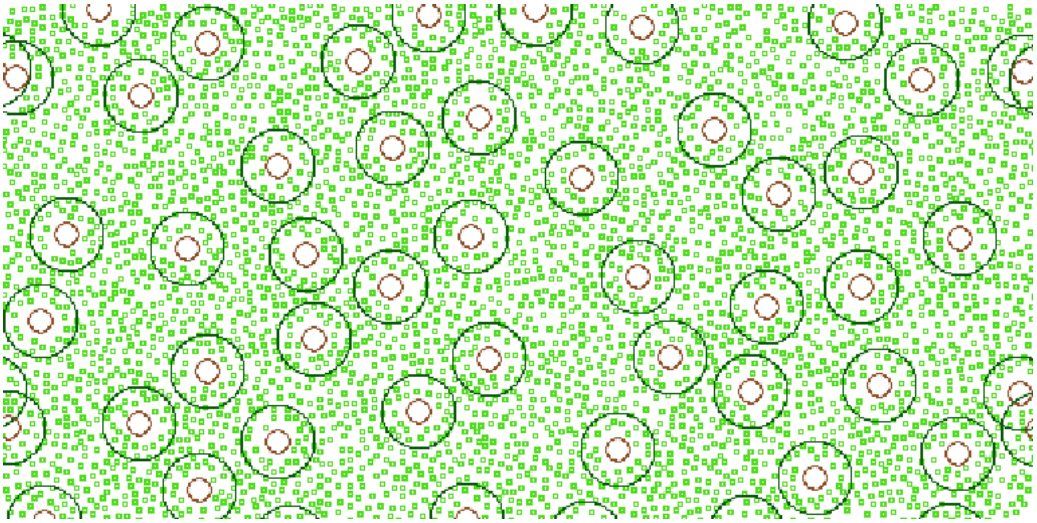


Figure 3.13: Poisson distributed entities in two layers. Upper layer with radius 7 (brown) with minimum group distance 14 which is visualized using dark green circles. In bottom layer light green entities of radius 2 are placed. The group distance between bottom and top layer is 0, which can be seen as the smaller entities can be closer to big entities than big entities can be to other big entities. Some bugs are visible in the right and left borders of the image where bigger entities do overlap. Due to the time limitation of the project the source of the bug was not found.

In later stages of experimentation with the algorithm, we found that interesting results were produced when parameters were added to control the distribution of entities. We added two

main parameters called *population* (N) and *self spawn chance* (P_s). These parameters gave us more control over the expected number of entities of each type. As explained in Section 2.7, the algorithm works by placing an entity, which can be called the origin entity, then placing new entities around it.

In an example with an origin entity of type A with a high self spawn chance, there would be a high probability of entities with type A to be placed around the origin entity. This would lead to clusters of entities with a high self spawn chance, as can be seen with the dark green entities in Figure 3.14. As long as the chance to self spawn was not 100% eventually a different type of entity needed be placed. The type of the new entity was selected by random, but still controlled using the spawn chance. This was done using the population value. The value could be seen as a statistical "number of entities to be placed".

The population value was used to create an interval between 0 and 1 for each entity. So for example in the case of two entities, the population value could be 40 and the second 60. The corresponding intervals would then be $[0, 0.4)$ and $[0.4, 1]$. This would give the first entity a 40% chance of being placed and the latter 60%. Selection was done by generating a random decimal value between 0 and 1, and then using binary search to find the entity with an interval containing the generated number. Self spawn chance also played a big role in the number of entities spawn of a type, therefore this was accounted for when calculating the intervals of each entity. First the total population needed to be calculated which was done using the formula:

$$N_{total} = \sum N_i(1 - P_{s,i}) \tag{3.8}$$

after which the interval for entity A_i could then be calculated using:

$$a_i = b_{i-1} \tag{3.9}$$

$$b_i = \frac{N_i(1 - P_{s,i})}{N_{total}} \tag{3.10}$$

where a_i is the start of the interval and b_i the end. For the case $i = 0$, a would be 0.

The effects of the population values can be seen in Figure 3.14, where red entities have a very low population value and light green have a high population value.

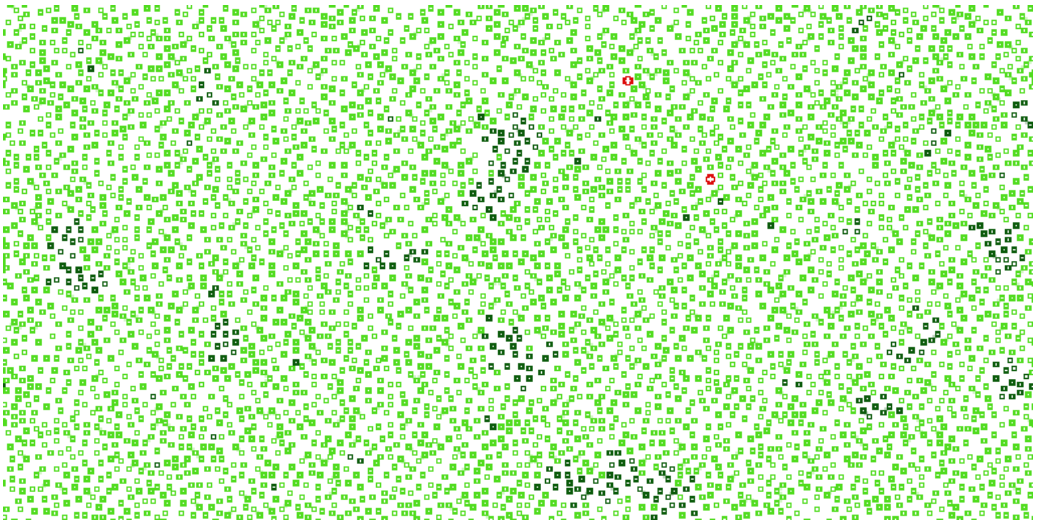


Figure 3.14: Entities distributed using Poisson disc distribution. Light green entities have self spawn chance $P_s = 0$ chance and population value $N = 500$, dark green entities have $P_s = 85\%$ leading to clusters. Red entities have $P_s = 0$ and $N = 1$ which can be seen in their low numbers in the figure. Light green entities have the highest population value and subsequently dominate in numbers.

4 Results

This project has ended with results in several different forms. The "actual" product that was envisioned came in the form of the core structure, consisting of the pipeline, along with a handful of modules. The pipeline has been integrated with Rimmer Motzheim's custom rendering engine, which he had worked on prior to this project. However, the modules have not yet been integrated into the pipeline, due to the many problems that arose during the development.

The four modules that have been implemented, namely the landmass, terrain, object placement and tree modules, have all yielded individual results. The modules have been rendered in *Matlab*, *OpenGL Utility Toolkit* (GLUT) and Apple's *SceneKit* to showcase their attributes for this report.

4.1 Core structure

The basic usage structure of the final framework is best described as a pipeline of stages, where each stage can contain multiple sub-tasks for that category. There are multiple reasons for this choice:

- Using a pipeline structure allows dividing the task at hand into several distinct stages. This makes it easier for a user to understand the process of generating the world by only having to think about the input and output of one stage at a time.
- By limiting stages within the pipeline to reading data from the stages before only, we can automatically prevent users from inadvertently creating recursion within their generators. Any form of dependency to data at the same stage would immediately cause an infinite recursion.
- Dividing the generation into stages makes it easier to implement LOD management, which is all but required to be able to handle large worlds. Each pipeline stage has its own base LOD it operates on.

This pipeline defines the full state of the world generator, and is walked through for each generated point in the world. Between each main configurable stage in the pipeline there is a conversion stage that takes care of converting the output format of the previous stage to the correct input format of the next stage. This allows the user code to focus on the generation itself without thinking about making it work together.

The pipeline itself is wrapped in a manager structure that manages storing and loading the

generated data. Since the world generator has no concept of a bounded world, the manager has to transparently manage tiling and chunking of the data from each stage, as well as storing it at the appropriate LOD. Through the manager it is possible to load and store data at any point in the virtual world, even if that point is outside of any area that has been used before. For this reason the data structures used by this manager have to be very efficient, and they form some of the most error sensitive code.

Due to the flexibility and configurability of the framework, these parts are the only ones that are required to use it. However, the framework also contains a set of additional helpers, default implementations and common functionality that most projects will want to use. Some of the most used include:

- A world manager that can be integrated in a game or engine, which will manage the set of currently visible chunks based on game data. This includes generating new parts of the world and paging out chunks that are too far away.
- A voxel generator that converts voxel data created by the pipeline to geometry that can be rendered by the game. Note that the framework has no requirement of using voxels - this is a common use case, but any sort of geometry generation can be bound to the pipeline.
- A common structure that can be used as the core for the generation of most world types. This includes being able to define different biome types, generating those types through attributes in the landmass stage, interpolating between the different biomes and generating terrain through noise functions.

Two more important attributes of the framework are that it is independent of any game engine, and it doesn't take control from the user over the work flow. It is implemented simply as a set of functions and classes that are called by a game using it, and it is the game that decides what happens next. The framework only provides functionality that can help with that decision.

4.2 Landmass module

The first stage in the pipeline is called the landmass stage. The main use of this stage is to define the base shape and attributes of the terrain at a large scale, where details are filled in by later generators. In our implementation, the initial land shape is represented through a Voronoi diagram with a set of attributes on each vertex. Using a Voronoi diagram makes it easy to implement a multitude of walking simulations that would be difficult to implement in a grid-like structure. Since the generator needs to be able to generate a subset of the world independently, it uses a grid of Voronoi tiles that are connected lazily. Landmass generation uses a linear generation workflow for each generated tile, where each tile is in one of the following stages:

- Firstly, a set of source points is generated using one of the point distributors. Most use cases use the random scattering distributor, which gives the best semi-uniform distribution for most world types.
- A Voronoi diagram is created from the generated points. This diagram is stored until the tile has generated enough data, after which the source diagram is not needed anymore.
- A final set of vertices is calculated from this tile's diagram and the diagrams of its neigh-

bouring tiles.

- A set of tile edges is created from the vertices and neighbouring diagrams. This includes information about which edges end in a neighbouring tile.
- The edge set is connected to the edges of neighbouring tiles. This allows later simulation stages to walk over the tile boundary without any handling of special cases.
- Finally, a set of simulation stages defined in the main pipeline are performed on the final structure. Each simulation can read the graph structure and any attributes defined by previous simulations, while it also writes one or more attributes to the graph. These simulations are performed in the same way as the above stages, where each neighbour in a tile has performed all simulations prior to the current one. This allows simulations to transparently use attributes defined by previous ones.

The main output of this stage is the graph shape and its attributes. These attributes are used by the next pipeline stage to generate the actual terrain. Examples of attributes that can be added to graph vertices are height, precipitation, landscape type, and more. The graph also supports defining attributes on edges, which can be used as source data for the generation of large-scale features such as rivers and roads. An unlimited number of attributes can be defined, although a large number can make the later generation very complex. A 2D renderer was created to visualize the properties which helps in debugging (Figure 4.1).

One important aspect that makes this system work in the general case is that for each stage in the generation of a Voronoi segment, all of its neighbours are generated recursively up to the previous stage. This allows attribute generators to use the source data for not only their own segment, but for all neighbouring segments as well. This facilitates a multitude of use cases without any special handling of dependencies between segments, and allows simulated features to use a continuous set of source data. Note though that, like everything in the simulation, only data from previous stages can be used.

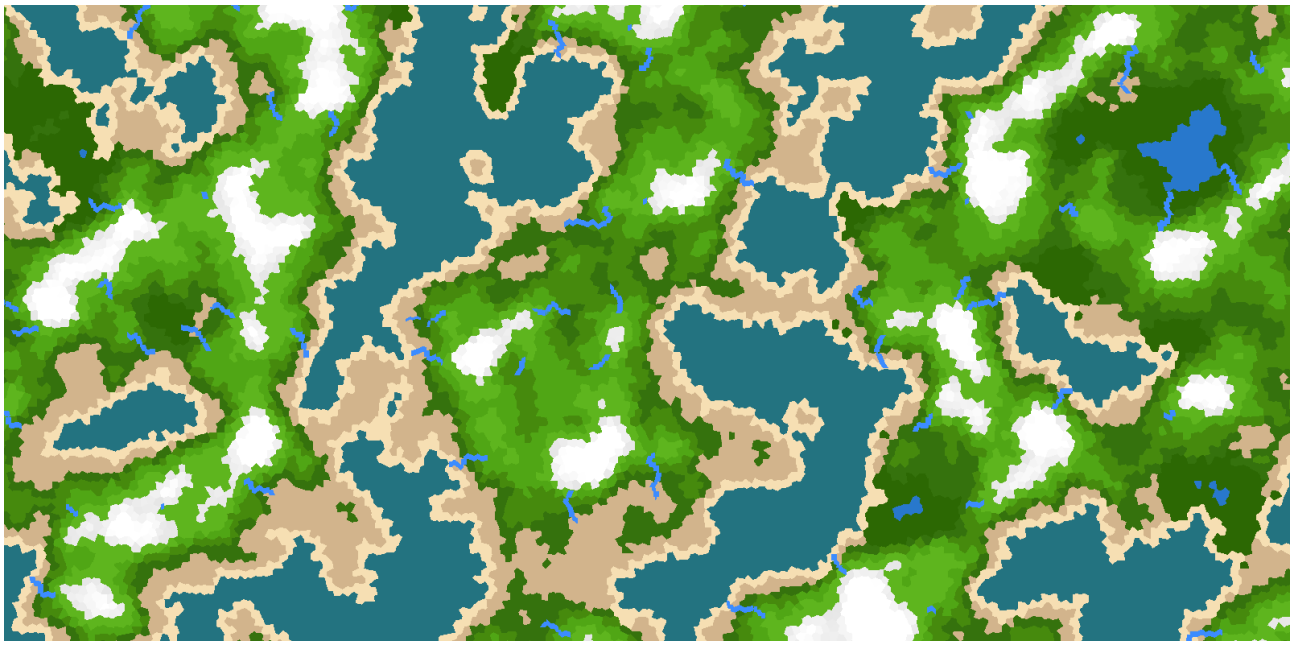


Figure 4.1: Rendering of the 2D representation of the landmass.

4.3 Terrain module

After the base land shape has been generated, the actual terrain has to be generated from it. The input to this stage is a tile of graph data from the landmass module. This tile is then divided into many smaller chunks, where each chunk contains a section of terrain that can be generated independently when needed. For each chunk, the pipeline calculates the graph vertex closest to the center of that chunk. This vertex and its direct neighbours are then used to interpolate the attributes from the landmass graph.

Our implementation generates the final terrain using voxels. Voxels are easy to integrate into the pipeline, since the voxel generator can take the provided interpolation vertices and interpolate the corresponding attributes based on the position of each voxel in a chunk. These interpolated values are then sent to the terrain generator.

To actually generate a voxel its position and interpolated attributes are passed to the *world biome function* (WBF), which is responsible for producing all terrain. Based on these parameters the WBF uses a number of voxel evaluation algorithms. Their combined output is then evaluated to decide the actual type of the generated voxel.

The main advantage of this system is that as long as the WBF maintains continuity in all biome parameters, the terrain can be interpolated by simply blending the input parameters. It is a very flexible solution, but it puts a lot of responsibility on the WBF, since it needs to make sure that the output terrain is continuous. It also goes well with the stated goal of modularity since the terrain generation can easily be modified by changing the workings of the WBF.

Our example implementation of the WBF consisted of two main steps. First a terrain surface height was calculated by evaluating the parameters that affect surface height. The following equation was used:

$$H(x,y) = H_B(x,y) + p_1 \cdot h_1(x,y) + p_2 \cdot h_2(x,y) + \dots \tag{4.1}$$

The next step was to apply 3D terrain modifications. First the surface height value was converted to a density by applying a linear fallof over the length l around the surface with the following equation.

$$\rho_s(x,y,z) = \begin{cases} 1 & \text{if } z < h(x,y) - l \\ 0.5 + \frac{h(x,y)-z}{l} & \text{if } h(x,y) - l < z < h(x,y) + l \\ 0 & \text{if } z > h(x,y) + l \end{cases} \tag{4.2}$$

Then all the density affecting parameters were allowed to apply their 3D terrain modifications:

$$D(x,y,z) = p_1 \cdot \rho_1(x,y,z) + p_2 \cdot \rho_2(x,y,z) + \dots \tag{4.3}$$

Last of all, the final density was converted to a material, in our case air if it was lower than 0.5 and otherwise ground. Figure 4.2 demonstrates some of the main features of this system. These features are seamless blending between different terrain, terrain layers (e. g. surface and underground) and complex details such as caves.

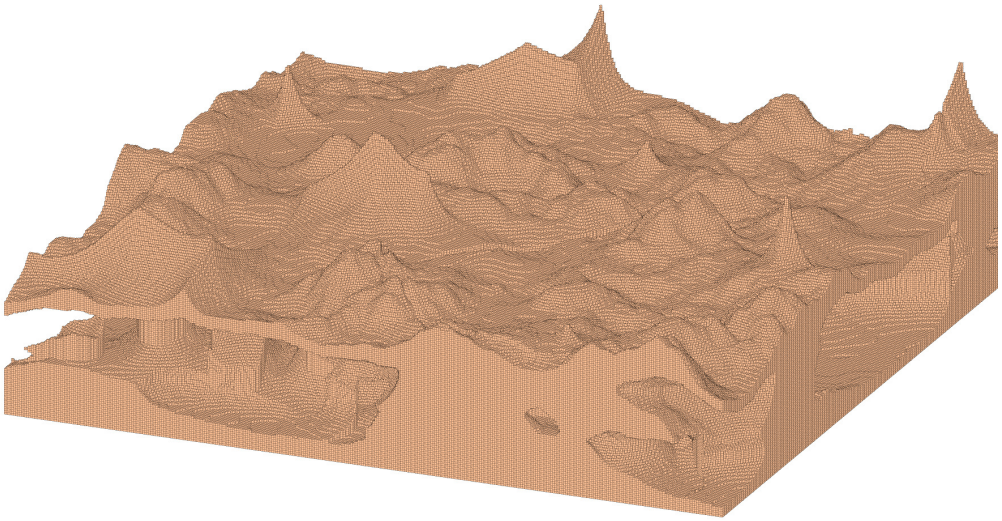
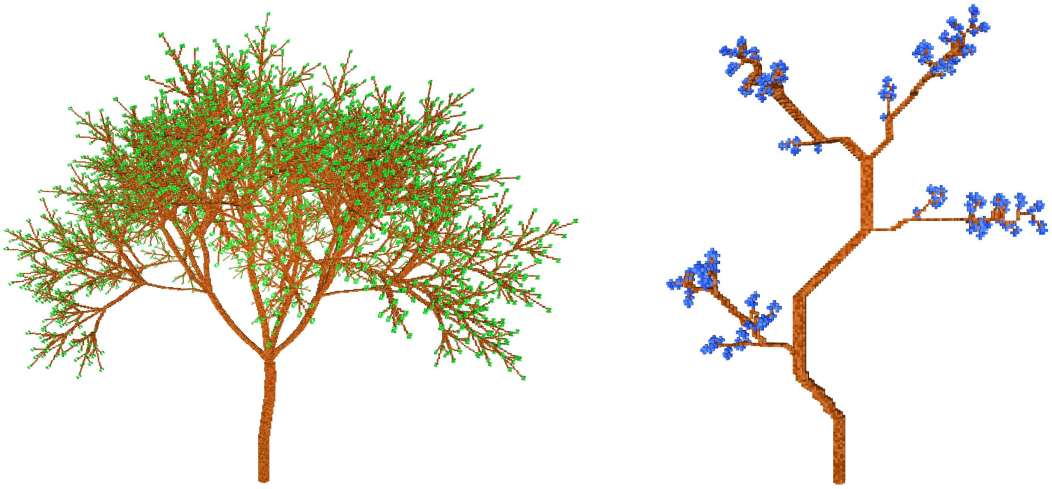


Figure 4.2: Voxel terrain generated using the terrain module. Three primary terrains have been used and interpolated on the surface layer: plains, regular mountains and ridged mountains. In the underground layer caverns and tunnels have been used. The terrain was plotted using Matlab.

4.4 Tree module

The resulting tree module implements an L-system for one specific tree structure, although minor variations occur within this system, i. e. the width, length and rotations of the branches have some noise added to them. The L-system and its string generation are implemented as structures of actual strings from the C++ `std::string` library. Furthermore, the 3D turtle that was mentioned in Section 2.6 reads each character in a generated string and interprets the symbols defined in the same section. Rotations are implemented directly in the module, reducing the need of external libraries such as *Generic Math Template Library* (GMTL) [40]. The generated string is then sent to a voxel generator which transforms the instructions encoded in the string to a collection of 3D coordinates and colors. This data would normally be passed on to another module, but has in this case been extracted from the module directly and then been rendered in Xcode (Figure 4.3a, the rules are shown in L-system B.3).

As a bonus example, the L-system 2.1 from Section 2.6 has been modified to be rendered in 3D (Figure 4.3b).



(a) A tree generated by the tree module and rendered in Xcode with *SceneKit* using an Objective-C/Swift interface bridge.

(b) The abstract tree constructed with a modified version of L-system 2.1 from Section 2.6 generated in the tree module.

Figure 4.3: The results of the tree module, using L-systems to construct trees.

4.5 Object placement module

The final version of the object placement module is based on Poisson disc distribution. The module is currently made for placing a predefined `struct` called an entity. Entities contain data used in the algorithm. The module supports having groups of entities where different groups can have an added minimum distance. In an example of a simulated forest it allows for grass to grow under the crowns of trees but keeping trees from having overlapping crowns (Figure 4.4).

The algorithm needs a set of input. The first step is instantiating entities to be placed by algorithm. Instantiating an entity is done by giving it an ID, a self spawn chance and spawn chance. Lastly an entity must be assigned to a group by setting its group ID variable. It is possible to have only a single group, in which case all entities will be placed on the same layer. The fields and structure of an entity can be seen in appendix C.2.

A group is defined by setting the minimum distance a group has to every other group including itself. For the algorithm to be able to prune entities a 2D decimal array must be provided, i. e. a mask. If no pruning is desired the algorithm will work without it. The last input is what is referred to as k value by Robert Bridson in his paper about Poisson disk sampling [41]. The k value is the number of tries to place an entity, around a previously placed entity, the algorithm makes.

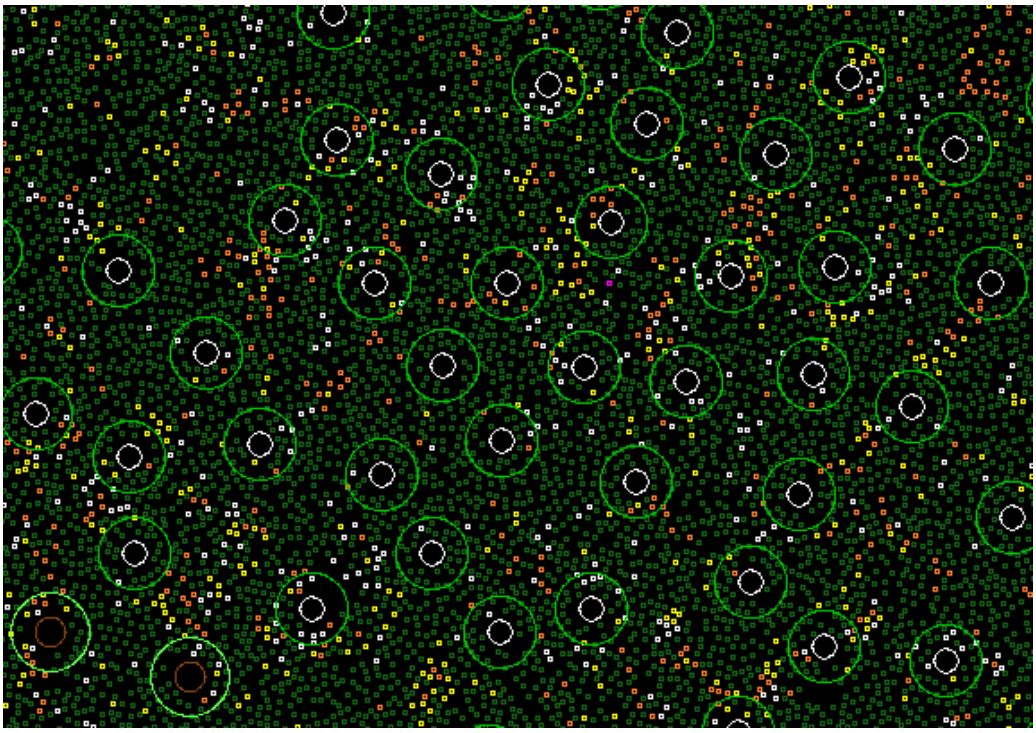


Figure 4.4: A simulation of a simple forest made using the feature placement module. Top layer consists of trees, represented by white and brown circles surrounded by bright green circles which represent tree crowns. No trees have overlapping crowns. In the bottom layer grass (small green dots) and flowers (small colored dots) have been placed. Grass has a visibly higher chance of being placed, leading to grass dominating the ground. Additionally flowers have a high chance of placing other flowers around, leading to the small clusters of flowers seen in the figure. Grass and flowers can not overlap with tree trunks but are allowed to grow under the crowns, a product of layering the entities.

5 Discussion

In the next following sections, the results will be discussed. The choice of methods will also be discussed to see if the result could have been improved in some way. The validity and generalisation of the engine will also be discussed, that will answer if the technology can be used in a wider spectrum of applications. Furthermore, future improvements that can emerge from the contributions of the game developer community will also be discussed.

5.1 Results and alternative methods

The work that has been done in the project has produced some satisfying results. Although our initial ambition was to create a fully functional engine with all the modules integrated, the project results are a good start on the path to a free PCG engine. In the following sections, it will be discussed if there was anything that could be improved with the methods chosen to tackle a specific problem, or if other methods would have produced different results.

The project workflow, where each subgroup could work independently, allowed for extensive research to be made for each module. However, due to the modules not being combined with the pipeline until the end of the project, several complications arose. If the modules had been continuously integrated from the beginning, by having a ladder of subgoals, this problem might have been reduced. On the other hand, it would have slowed down the early development, since everything was dependent on the pipeline.

5.1.1 Core structure

Having the generation framework as a separate entity that doesn't dictate anything about its usage, makes it flexible in where it can be deployed. It has no coupling to any game engine or rendering method, so it can be used for almost any use case. This can be a problem with frameworks that require using them in a certain way or in a certain context.

One consequence of this approach, which some may see as a drawback, is that the framework cannot be used stand-alone. This means that in order to use and render worlds with the framework it first needs to be integrated in a supporting game infrastructure, which initializes the pipeline and uses it to generate world data. Our expectation is that if a larger number of users start building projects using the framework, then a common infrastructure will arise with bindings to multiple common game engines.

5.1.2 Landmass module

The main advantage of using Voronoi diagrams as the source for the land shape is that the graph naturally gets an irregular shape, and paths through the graph will not be straight. This makes it easier to generate interesting terrain, especially when it comes to large-scale features (such as rivers) that affect a large part of the terrain.

A major drawback of the Voronoi generation pipeline was that it was very complex and difficult to implement, and took such a large amount of time that we were unable to create a full demo combining each part in the end. This makes it doubtful if the use was worth it for this project, compared to the initial bitmap implementation. Most features of the Voronoi approach can be simulated in other ways for a grid structure, which may make for a simpler implementation overall. Voronoi diagrams are most useful when graph traversal is needed, e. g. in simulations. Simulations were deemed as a method that should be avoided as much as possible, further questioning the use of Voronoi diagrams.

Simulations could have been avoided by using other methods that would have yielded similar results. E.g. Tarboton et al. [42] discusses the fractal nature of river networks, which could potentially be exploited to deterministically generate rivers without dependencies to neighbour data. Making a fractal river fit in the world requires modification of the surrounding terrain. Modifications can be deterministically calculated, solving the dependency problem, but the method creates new problems in making fitting modifications.

5.1.3 Terrain module

The general advantages and disadvantages of using voxels have already been discussed in Section 2.3. In our case we feel that voxels is the optimal choice, as it works well with representing a wide variety of different landscapes. However, since performance is the main drawback and we have not been able to do any live tests, the cost might prove to be too high.

Nevertheless, we are quite satisfied with the general approach of the terrain generation. Describing a terrain as a set of continuous parameters allows, in theory, any imaginable terrain with good looking interpolation. Our implementation has just scraped the surface, with only a couple of biome attributes and no special transitions. For example, in some cases it would be possible to generate a cliff when mountains meet plains instead of gradually transitioning to hills.

Since the system is allowed to use many different types of voxel generation methods, one of the drawbacks is that sub-optimal interpolation techniques are sometimes needed. The advantage of allowing this however, was a great deal of freedom as our only constraint was that the method should be deterministic and output voxel density data. Limiting the system to use one generation method, for example 3D noise, would make it a lot harder to produce certain landscapes.

All of our terrain generation functions have one thing in common: they use simplex as the only noise source. While this was enough for prototyping purposes, we feel that including other types of noise could improve terrain diversity and allow for some features that are currently difficult to produce. For example, some terrain features that required a combination of up to 20 different simplex variations could probably be generated much more easily with a different

noise function. Besides simplifying the development, this would also improve performance, as long as the new functions are as fast, since each call costs approximately the same amount of time.

Our method of removing floating voxels has some compromises, necessary to produce results in a timely fashion. To determine if a voxel-partition is part of the ground, the algorithm is allowed to look at the voxel data of the adjacent chunks to determine if the voxels in the partition should be removed or not. By only checking neighbouring chunks, voxel partitions that in fact are part of the ground can potentially be removed, or in the same manner floating voxel chunks can potentially be kept.

By increasing the range of the chunks that are checked to determine if a voxel partition belongs to the ground or not we can increase the probability that the correct voxels are kept or removed. The downside is that this will decrease performance and increase memory usage. Note that the range has to be finite, if not the algorithm could potentially have to continue indefinitely in an infinite world.

5.1.4 Tree module

The major drawback of our implementation is that the use of voxels do not work as well with L-systems as it would with polygon meshes instead. It is more difficult to rotate voxels in 3D so that they fall naturally into a discrete space. With meshes, these issues don't exist, since the coordinates of the vertices are stored in `float` values (continuous space) instead of the `int` values used for voxels (discrete space).

The variational possibilities of our implementation the L-system lies only in the lengths and widths of the branches. It would be an improvement to extend the module to support multiple production rules per variable, making it a stochastic OL-system. To make the generation deterministic in the sense that it generates the same output for the same input, a specific PRNG could just be used when choosing between productions. This approach would make it possible to have variations in the structure itself, yielding in a larger variational output.

The positive thing about the use of L-systems is that it is highly parametrised. Theoretically any structure can be produced only by altering the production rules, and the appearance can be changed by using other attributes for branch length ratio, different probabilities for the productions, etc.

While L-systems are versatile enough to generate many different biological structures, other methods can also be used. One method that would translate better with the use of voxels is simulating *Laplacian growth*. Although it is rarely used in computer graphics, Theodore Kim et al. [43] have managed to develop an algorithm that exceeds in performance compared to previous algorithms.

5.1.5 Object placement module

The object placement module is strictly a proof of concept in its current state. No optimization efforts have been made. Because the pipeline would need to place any objects during run-time a slow algorithm would cause severe lag. Small changes like comparing squared distances instead

of calculating the square root and fast-estimate versions of sine/cosine functions could speed up the algorithm.

Additionally, finding a good solution of handling collision detection, i. e. checking for nearby entities, would help. In the final version of the module a so called bucket grid is used. Each cell contains a bucket (or list) of entities and is the size of the largest entity. A bucket can contain only one of the largest entities, or many smaller entities. This scales very poorly with size difference of the smallest and biggest entity. As an example with two entities A and B the maximum amount of entities n in a bucket is:

$$n = \frac{\text{size}(A)}{\text{size}(B)} \tag{5.1}$$

where A is the bigger entity. So for a large difference in size the the time complexity increases greatly. A solution might be to have many grids with cells of different sizes. A natural example would be having a grid for each layer of entities, which are already grouped by size. This would mean the maximum number of entities n_i per grid G_i would be smaller. This is of course under the assumption that the groups are made with this i mind, keeping similarly sized entities together.

Lastly the algorithm has some bugs which remain unsolved. In some fringe cases entities can be placed outside the allowed area. When this happens the minimum distance is sometimes disregarded, leading to entities overlapping when the should not be able to. The bug needs to be solved before an implementation is made with the pipeline as it would prove very unaesthetic, but for testing purposes the bug does not cause any real problems.

5.2 Validity

The modules has turned out to be more interconnected than expected. It is not easy to make a module completely generalised. During the development of the system we found that it was convenient to do more in the earlier stages of the pipeline. One example of this was when we decided to do the biome interpolation before the terrain generation module, thus having the WBF expect pre-interpolated data. This arguably reduces modularity, as it adds a constraint to the WBF input.

5.3 Future work

The most basic and critical improvement of the system is to connect all individual modules and make them work with each other. Another important upgrade is to make the system easy to integrate with a wide variety of game engines, such as *Unity* and *Unreal Engine*.

There are also several ways to improve the existing base modules. For example, all the techniques used in the terrain module were based on noise and therefore ontogenetic. It would be very interesting to experiment with improving the realism by simulating environmental effects. Erosion algorithms could be used to simulate the effect of wind and rain on mountains.

There are several features that are not yet supported by the system. Two of the more essential unsupported features are textures and voxel smoothing. Less important, but desired, features includes *sky-boxes*, and LOD at different stages of the pipeline.

Some of the improvements mentioned above, especially module integration, could be considered as part of the MVP. But by its modular nature there are numerous of extensions and refinements which could further improve the system. As mentioned in Section 1.2, the framework is meant to be distributed as free software, which would enable anyone to access the source code. By letting independent developers add and edit features, there could potentially be a large-scale collaboration to improve the framework. This would could give the framework far more functionality and features than what a small team of developers could add. This is however not something we would actively pursue beyond distributing the software.

6 Conclusion

This project has only been the start of the idea of having an open source tool for procedural content in graphics computing. The main goal was to build a foundation based on the conceptual model that was brought forward during the project. A core structure was built to allow for the modularity that the goals promised. Several modules were implemented as well, both as to serve as a base for the system and as a proof of concept.

The landmass module was created with the task of producing a large scale two-dimensional representation of the world. Early in the project, Voronoi diagrams were chosen as a suitable solution for this task, and thus this technique was implemented with satisfactory results. Interpolation between different biomes was initially done by using triangulation between three cells. Later on it was changed so that the interpolation was based on the vertices, producing smoother interpolation thanks to the fact that it uses more control points. The landmass module produced satisfactory result, but in the choice of using Voronoi diagrams was questionable. They were overly complex and offered few advantages compared to simpler solutions.

The team came to the conclusion early in the project that voxels was a good way to represent the world in the base modules. Therefore the terrain module was developed with the functionality to produce 3D landscapes with voxels by the use of different noise functions. Additionally, techniques to improve the results of the noise functions, such as removing voxels that seemed misplaced, were investigated. Good looking transitions between different landscapes proved to be very difficult. The initial idea of patching together independently generated terrains was scrapped in favor of a system where all the terrain was continuously described by the same set of parameters. This solution might seem highly constraining, but it turned out to be very robust while still allowing a high diversity in the generated landscapes. In the end the team felt that the terrain module worked well and could create adequate terrain with sufficient flexibility.

L-systems was from the very beginning perceived as an interesting solution for creating objects. This technique was explored in the tree module, that utilize L-systems to generate trees. Initially polygon meshes where used to represent the trees. Later on the trees were made by voxels. This had some drawbacks with the rotations of the branches with discrete coordinates, but it was consistent with the terrain module which also usesd voxels. Overall the tree module was able to create adequate trees, and L-systems were established as a good and versatile object creation technique.

Several techniques were explored for object placement, such as random distribution and random scattering. As a result of this research it became clear that Poisson disc sampling was a good solution and it was implemented in the object placement module. Furthermore, the module was extended with solutions for handling objects of different sizes, as well as further improvements of grouping similar objects together in a realistic manner. While the object placement tool

needs to be refined, the method of using Poisson disc sampling with some special placement rules was considered to work well.

Not all bits and pieces were able to be integrated fully into the pipeline, leaving the complete implementation a bit fractured. The lesson learnt here is that procedural generation is hard, since it involves so many unforeseeable results between the generation steps. Making it all tie together, and at the same time make it at some level general, is a challenging task.

The future development of the engine could benefit from the open source model. Experienced programmers could give input on the architecture of the pipeline and the generalisations made for the modules. Having the engine open source would allow for developers to add any features or functions they might require, allowing the engine to grow where needed. With time the toolkit could be considered a valuable resource in game development.

Bibliography

[1] Eric W. Weisstein. *Barycentric Coordinates*. en. URL: http://mathworld.wolfram.com/BarycentricCoordinates.html.

[2] Mark Hendrikx et al. "Procedural content generation for games". In: *ACM Transactions on Multimedia Computing, Communications, and Applications* 9.1 (Feb. 2013), pp. 1–22. ISSN: 15516857. DOI: 10.1145/2422956.2422957. arXiv: 1005.3014. URL: http://dl.acm.org/citation.cfm?doid=2422956.2422957.

[3] George Kelly and Hugh McCabe. "A survey of procedural techniques for city generation". In: *ITB Journal* (2006), pp. 87–130. URL: http://www.citygen.net/files/Procedural_City_Generation_Survey.pdf.

[4] Don Worth. *Beneath Apple Manor*. The Software Factory. 1978.

[5] Michael Toy and Glenn Wichman. *Rogue*. 1980.

[6] Blizzard North. *Diablo*. Blizzard Entertainment. 1997.

[7] Gearbox Software. *Borderlands*. 2K Games. 2009.

[8] Julian Togelius et al. "What is procedural content generation?" In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games - PCGames '11*. New York, New York, USA: ACM Press, June 2011, pp. 1–6. ISBN: 9781450308724. DOI: 10.1145/2000919.2000922. URL: http://portal.acm.org/citation.cfm?doid=2000919.2000922.

[9] Hello Games. *No Man's Sky*. Hello Games. 2016.

[10] Voxel Farm Inc. *Voxel Farm*. 2010.

[11] Interactive Data Visualization, Inc. *SpeedTree*.

[12] Unity Technologies. *Unity*. 2005.

[13] Epic Games. *Unreal Engine*. 1998.

[14] Wikidot. *Teleological vs. Ontogenetic - Procedural Content Generation Wiki*. URL: http://pcg.wikidot.com/pcg-algorithm:teleological-vs-ontogenetic.

[15] John Von Neumann. "Various techniques used in connection with random digits". In: *J. Res. Nat. Bur. Stand.* 12 (1951), pp. 36–38. URL: https://dornsifecms.usc.edu/assets/sites/520/docs/VonNeumann-ams12p36-38.pdf.

[16] George Marsaglia. "Xorshift RNGs". In: *Journal Of Statistical Software*. Vol. 8. 14. 2003, pp. 1–6. URL: http://www.jstatsoft.org/v08/i14/paper.

[17] Makoto Matsumoto and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (Jan. 1998), pp. 3–30. ISSN: 10493301. DOI: 10.1145/272991.272995. URL: http://dl.acm.org/citation.cfm?id=272991.272995.

[18] Oracle. *Random (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html (visited on 05/03/2016).

[19] Ken Perlin. "Improving noise". In: *ACM Transactions on Graphics* 21.3 (July 2002), pp. 2–3. ISSN: 07300301. DOI: 10.1145/566654.566636.

[20] Ken Perlin. *Noise and Turbulence*. URL: https://mrl.nyu.edu/~perlin/doc/oscar.html.

[21] Markus Persson. *Minecraft*. 2011.

[22] Markus Persson. *The Word of Notch*. 2011. URL: http://notch.tumblr.com/post/3746989361/terrain-generation-part-1 (visited on 05/28/2016).

[23] Stefan Gustavson. "Simplex noise demystified". In: *Linköping University, Schweden* (2005), p. 17. DOI: 10.13140/RG.2.1.3369.6488.

[24] William E. Lorensen. "Marching cubes: A high resolution 3D surface construction algorithm". In: *ACM SIGGRAPH Computer Graphics* 21.4 (Aug. 1987), pp. 163–169. ISSN: 00978930. DOI: 10.1145/37402.37422. URL: http://dl.acm.org/citation.cfm?id=37402.37422.

[25] Allinonemovie. *Minecraft screenshot*. [Image]. URL: https://pixabay.com/sv/minecraft-berg-videospel-block-655158/.

[26] *Digital Image Interpolation*. URL: http://www.cambridgeincolour.com/tutorials/image-interpolation.htm (visited on 05/16/2016).

[27] Cmglee. *Bilinear interpolation visualisation*. [Image]. URL: https://en.wikipedia.org/wiki/File:Bilinear_interpolation_visualisation.svg.

[28] Steven Fortune. "A sweepline algorithm for Voronoi diagrams". In: *Algorithmica* 2.1-4 (Nov. 1987), pp. 153–174. ISSN: 01784617. DOI: 10.1007/BF01840357. URL: http://link.springer.com/10.1007/BF01840357.

[29] David H. Laidlaw and Joachim Weickert. *Visualization and Processing of Tensor Fields*. Springer Science & Business Media, 2009, p. 181. ISBN: 978-3-540-88377-7. DOI: 10.1007/978-3-540-88378-4. URL: http://link.springer.com/10.1007/978-3-540-88378-4.

[30] Don P. Mitchell. "Generating antialiased images at low sampling densities". In: *ACM SIGGRAPH Computer Graphics* 21.4 (Aug. 1987), pp. 65–72. ISSN: 00978930. DOI: 10.1145/37402.37410.

[31] Aristid Lindenmayer. "Mathematical models for cellular interactions in development. II. Simple and branching filaments with two-sided inputs." In: *Journal of theoretical biology* 18.3 (1968), pp. 300–315. ISSN: 00225193. DOI: 10.1016/0022-5193(68)90080-5.

[32] Harold Abelson and Andrea DiSessa. *Turtle Geometry*. MIT Press, 1986, p. 497. ISBN: 9780262510370.

[33] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants (The Virtual Laboratory)*. The Virtual Laboratory. New York, NY: Springer New York, 1991, p. 240. ISBN: 0387972978. DOI: 10.1007/978-1-4613-8476-2.

[34] *Boost C++ Libraries*. 2012. URL: http://www.boost.org/ (visited on 05/15/2016).

[35] Takao Ohya, Masao Iri, and Kazuo Murota. *Improvements of the incremental method for the Voronoi diagram with computational comparison of various algorithms*. 1984. URL: http://www.orsj.or.jp/~archive/pdf/e_mag/Vol.27_04_306.pdf (visited on 05/10/2016).

[36] Mir A. Mostafavi, Christopher Gold, and Maciej Dakowicz. "Delete and insert operations in Voronoi/Delauney methods and applications". In: *Comp. Geosci* 29.4 (May 2003), pp. 523–530. ISSN: 00983004. URL: http://www.sciencedirect.com/science/article/pii/S0098300403000177.

[37] Tyler J. Alumbaugh and Xiangmin Jiao. "Compact Array-Based Mesh Data Structures". In: *Proceedings of the 14th International Meshing Roundtable*. 2005, pp. 485–503. DOI: 10.1007/3-540-29090-7_29. URL: http://link.springer.com/10.1007/3-540-29090-7%7B%5C_%7D29.

[38] Amit Patel. *Polygonal Map Generation for Games*. 2010. URL: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/ (visited on 05/10/2016).

[39] Sergey Kosarevsky. *poisson-disk-generator*. URL: https://github.com/corporateshark/poisson-disk-generator.

[40] Allen Bierbaum, Kevin Meinert, and Ben Scott. *GMTL Programmer's Guide*. URL: http://ggt.sourceforge.net/gmtlProgrammersGuide-0.6.1.pdf.

[41] Robert Bridson. "Fast Poisson Disk Sampling in Arbitrary Dimensions". In: *Engineering* (2006). DOI: 10.1145/1278780.1278807.

[42] David G. Tarboton, Rafael L. Bras, and Ignacio Rodriguez-Iturbe. "The fractal nature of river networks". In: *Water Resources Research* 24.8 (Aug. 1988), pp. 1317–1322. ISSN: 00431397. DOI: 10.1029/WR024i008p01317. URL: http://doi.wiley.com/10.1029/WR024i008p01317.

[43] Theodore Kim et al. "Fast Simulation of Laplacian Growth". In: *IEEE Computer Graphics and Applications* 27.2 (Mar. 2007), pp. 68–76. ISSN: 0272-1716. DOI: 10.1109/MCG.2007.33. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4118496.

A Terrain Examples

A.1 Plains, mountains and hills

A heightmap of plains, mountains and hills can be generated directly from 2D simplex octave noise with different parameters (Table A.1). By substituting the regular 2D simplex noise from the previous section with turbulent noise (Section 2.2.2) something more akin to mountain ranges can be generated.

Table A.1: Noise parameters to generate different terrains

Landscape	Octaves	Frequency gain	Persistence
Plains	2	0.002	0.4
Hills	3	0.004	0.5
Mountains	4	0.006	0.6

A.2 Tunnels

Turbulent noise can also be viewed as a 2D projection of a tunnel system. In the simplest form a flat tunnel system at height h and with tunnel radius r can be created by the following algorithm:

- Use the turbulent noise to decide the layout of a tunnel network.
- Check if the evaluated point is withing the desired height, that is $|z - h| < r$.
- If the above conditions are fulfilled, set the voxel to air.

More sophisticated tunnels can be created by randomizing the tunnel height h (for example with regular simplex noise), and using the intensity of the turbulent noise to decide the tunnel radius. Also some 3D noise can be used to make the tunnels less uniform.

A.3 Caverns

The trick for creating caverns is to control regular 3D noise with a non-random height based function $d_h(z)$. The purpose of this function is to decrease the likelihood of generating solid voxels in the middle of the cavern and to increase it when approaching the caverns upper or

lower bounds. A very simple example of such a function is:

$$d_h(z) = (\frac{z - 0.5 \cdot h_{\text{cavern}}}{0.5 \cdot h_{\text{cavern}}}) \tag{A.1}$$

This will linearly decrease the likelihood of a voxel being solid near the middle of a cavern. This can easily be modified to produce the wanted distribution. For example changing it to $d_h(z)^2$ will introduce a sharper transition to solid near the roof and floor of the cavern.

A.4 Pillars

In the beginning of the project, pillars were often brought up as an example of difficult terrain to generate and interpolate. The following algorithm, however, produces something resembling pillars at least, with parameters h_s and h_p (surface and pillar height, respectively).

- Use 2D simplex noise with a high cutoff to create a mask of the pillar locations
- If $h_s \leq z \leq h_s + h_p$ make the voxel solid
- (optional) Use 3D simplex noise to erode the pillar to achieve a more weathered look

B L-system Examples

Below are some examples on various L-systems. Figure B.1 and B.2 depict the turtle drawing of the respective L-systems.

$$\begin{aligned}
\delta &= 90^\circ \\
\omega &: X \\
p_1 &: X \rightarrow F[+F-[X]]X \\
p_2 &: F \rightarrow XF
\end{aligned}$$

L-system B.1: A two-dimensional DOL-system with two productions.

$$\begin{aligned}
\delta &= 10^\circ \\
\omega &: X \\
p_1 &: X \rightarrow F[++++++F-Y]+F[----Y+M]F-X \\
p_2 &: Y \rightarrow F[------F+Z]-F[--F-Z]F+Z \\
p_3 &: Y \rightarrow FX \\
p_4 &: F \rightarrow FF
\end{aligned}$$

L-system B.2: A two-dimensional DOL-system with four productions.

Phyllotactic angle: 18°, branching angle: 10°.

$$\begin{aligned}
\omega &: F \\
p_1 &: F \rightarrow Y[\wedge\wedge\wedge\wedge F*]++++++[\wedge\wedge\wedge\wedge F*] \\
&\qquad ++++++[\wedge\wedge\wedge\wedge F*]++++++[\wedge ZF*] \\
p_2 &: Y \rightarrow Y \wedge Z\&++
\end{aligned}$$

L-system B.3: A three-dimensional DOL-system with two productions. The symbol Z has no production rule, and simply means "draw forward", as all the other letters. The $*$ symbol means "draw a leaf".

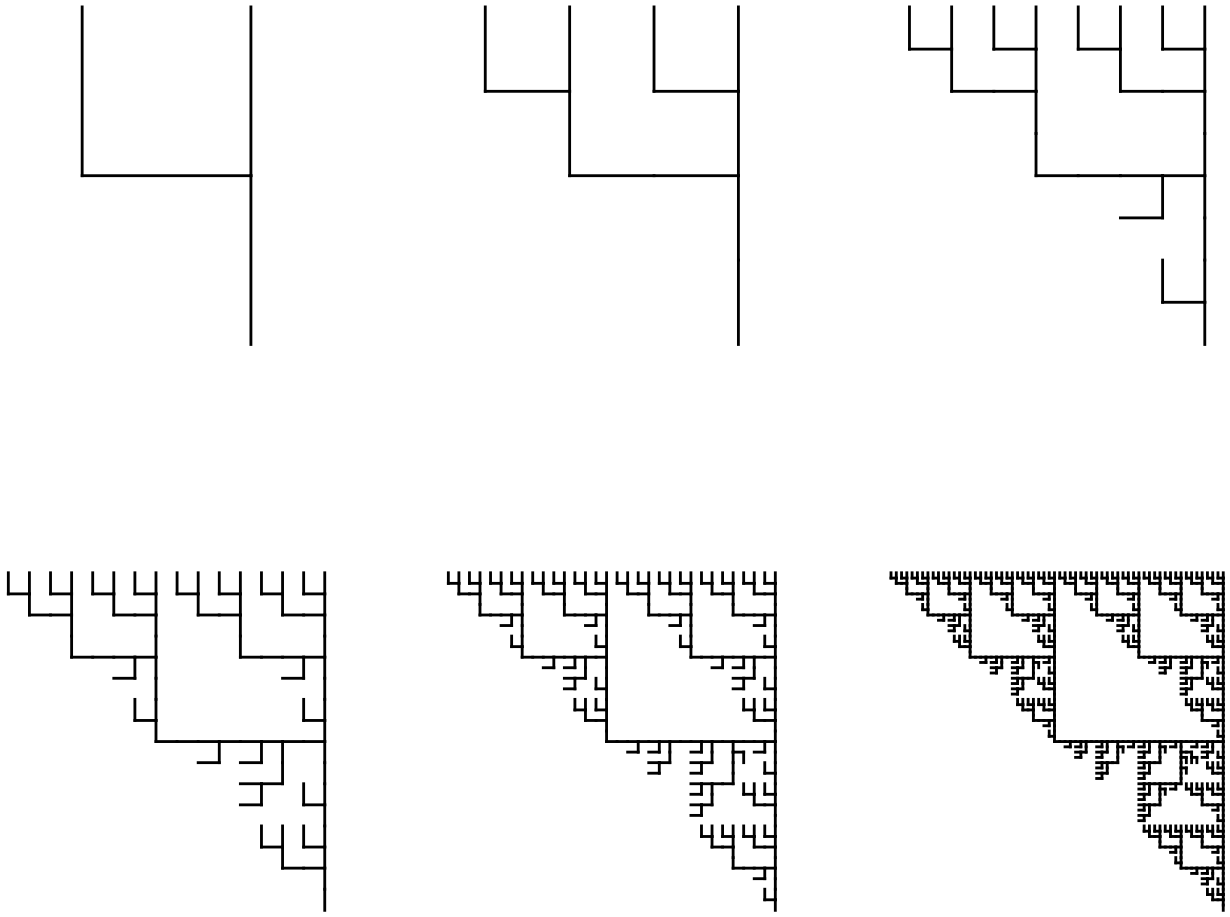


Figure B.1: An abstract tree constructed with L-System B.1 from iteration 1 through 6.

L-system B.3 was used to produce the results in Section 4.4. The *phyllotactic angle* is the angle that children branches spread around the parent branch. This is the same as the angle for the turtle to turn left and right, namely the + and − symbols, respectively. Thus, the turtle turn by 18° for each of these symbols, in the corresponding direction. The *branching angle* is the angle between the axises of a parent branch and a child branch. This is the same as the rest of the orientation symbols for the turtle. Thus, 10° is used for all the remaining symbols.

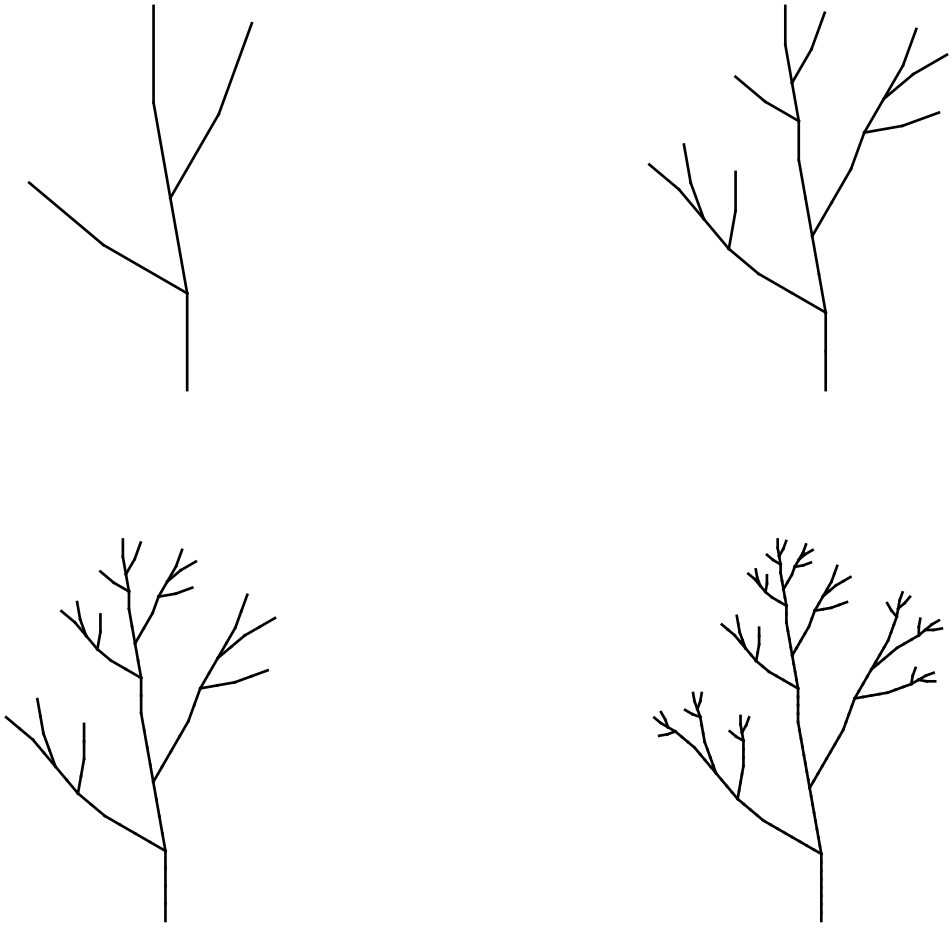


Figure B.2: A more realistic tree constructed with L-System B.2 from iteration 1 through 4.

C Pseudocode

```
1  GridCell:
2      step = Nothing
3      diagram = new VoronoiDiagram()
4      vertices = Vertex[]
5      edges = Edge[]
6      vertexEdges = EdgeIndex[]
7      cellEdges = EdgeIndex[]
8      unconnectedVertexEdges = UnconnectedEdge[][]
9      unconnectedCellEdges = UnconnectedEdge[][]
10
11     buildVertices:
12         if (step >= Vertices):
13             return
14         step = Vertices
15
16         points = getSites()
17         foreach (n in neighbours):
18             points.addAll(n.getSites())
19         diagram.create_from(points)
20
21         index = 0
22         foreach (v in diagram.vertices where isInThisGridCell(v)):
23             vertices.add(v)
24             v.index = index
25             index++
26
27     buildEdges:
28         if (step >= Edges)
29             return
30         buildVertices()
31         step = Edges
32
33         foreach (n in neighbours):
34             n.buildVertices()
35
36         index = 0
37         foreach (v in diagram.vertices where isInThisGridCell(v)):
38             position = 0
39
40             foreach (e in v.edges):
41                 if (isInThisGridCell(e)):
42                     edge = new Edge(createVertexIndex(e.v0), createVertexIndex(e.v1))
43                     edges.add(edge)
44                     edge.index = index
45                     vertexEdges[v.index].add(new EdgeIndex(this, index))
46                     index++
47                 else:
48                     gridcell = findGridCellFor(e)
49                     ue = new UnconnectedEdge(gridcell, e.v0, e.v1, position)
50                     unconnectedVertexEdges[v.index].add(ue)
51
52                 position++
53
54         index = 0
55         foreach (c in diagram.cells where isInThisGridCell(c)):
```

```
56              cell = new Cell()
57              cells.add(cell)
58
59              position = 0
60              foreach (e in c.edges):
61                  if (isInThisGridCell(e)):
62                      cellEdges.add(new EdgeIndex(this, e.index))
63                  else:
64                      gridcell = findGridCellFor(e)
65                      ue = new UnconnectedEdge(gridcell, e.v0, e.v1, position)
66                      unconnectedCellEdges[index].add(ue)
67
68                  position++
69
70              index++
71
72
73      createVertexIndex(v):
74          gridcell = findGridCellFor(v)
75          if (gridcell != this):
76              return new VertexIndex(gridcell, gridcell.findVertexIndex(v))
77          else:
78              return new VertexIndex(this, v.index)
79
80
81      connectEdges:
82          if (step >= Connections)
83              return
84          buildEdges()
85          step = Connections
86
87          foreach (n in neighbours):
88              n.buildEdges()
89
90          foreach (i in 1..vertexEdges.size):
91              each (uv in unconnectedVertexEdges[i]):
92                  index = uv.gridcell.findEdgeIndex(ue.v0, ue.v1)
93                  vertexEdges.addAt(ue.position, new EdgeIndex(ue.gridcell, index))
94
95          foreach (i in 1..cellEdges.size):
96              foreach (ue in unconnectedCellEdges[i]):
97                  index = ue.gridcell.findEdgeIndex(ue.v0, ue.v1)
98                  cellEdges.addAt(ue.position, new EdgeIndex(ue.gridcell, index))
99
100         delete unconnectedVertexEdges
101         delete unconnectedCellEdges
102         delete diagram
```

Listing C.1: Algorithm for connecting Voronoi diagrams

```
Entity:
    int id
    int group
    int count

    float x
    float y
    float radius

    float selfSpawnChance
```

Listing C.2: Entity structure.

D Site Generation Comparison

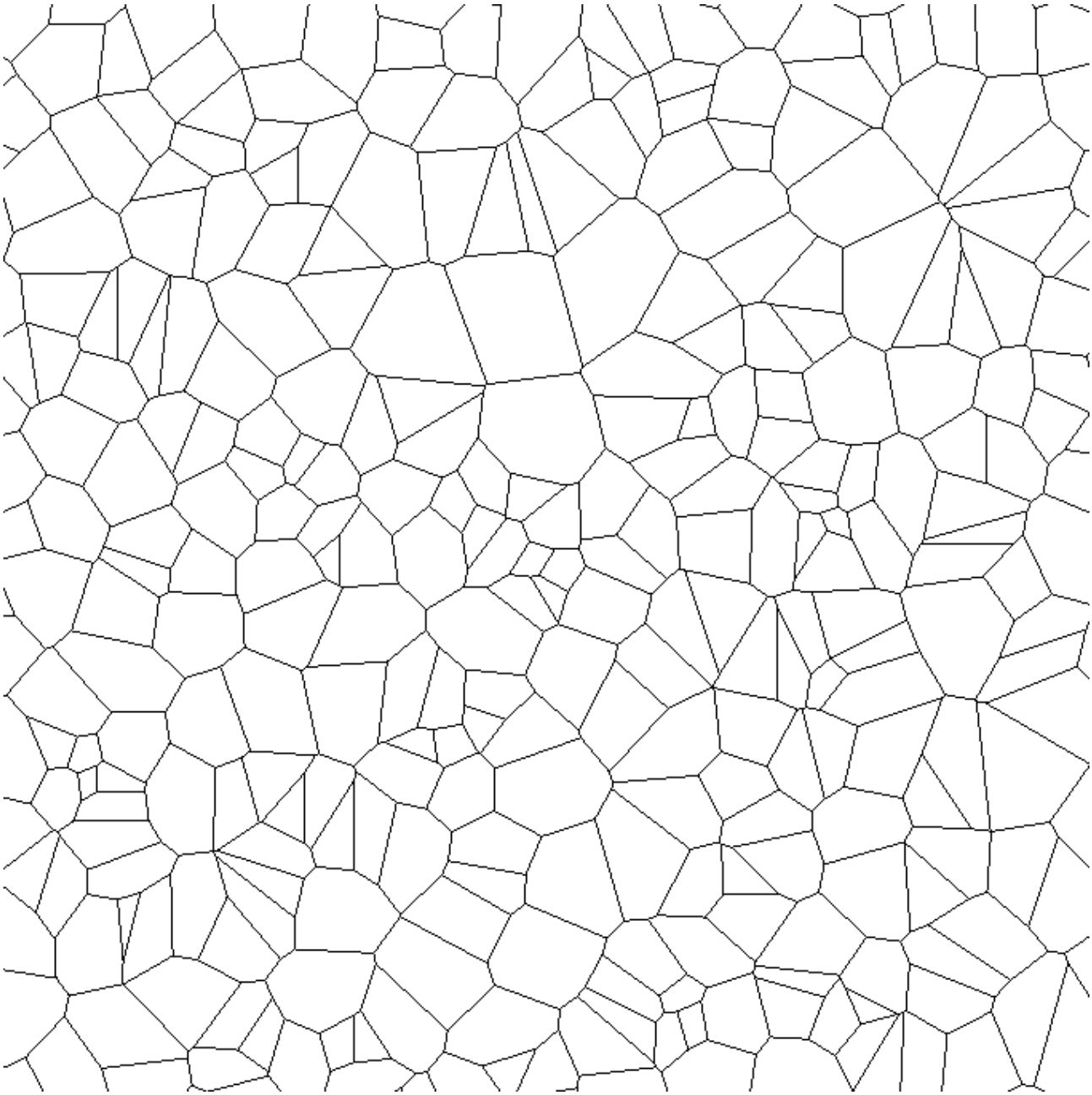


Figure D.1: Uniform random distribution.

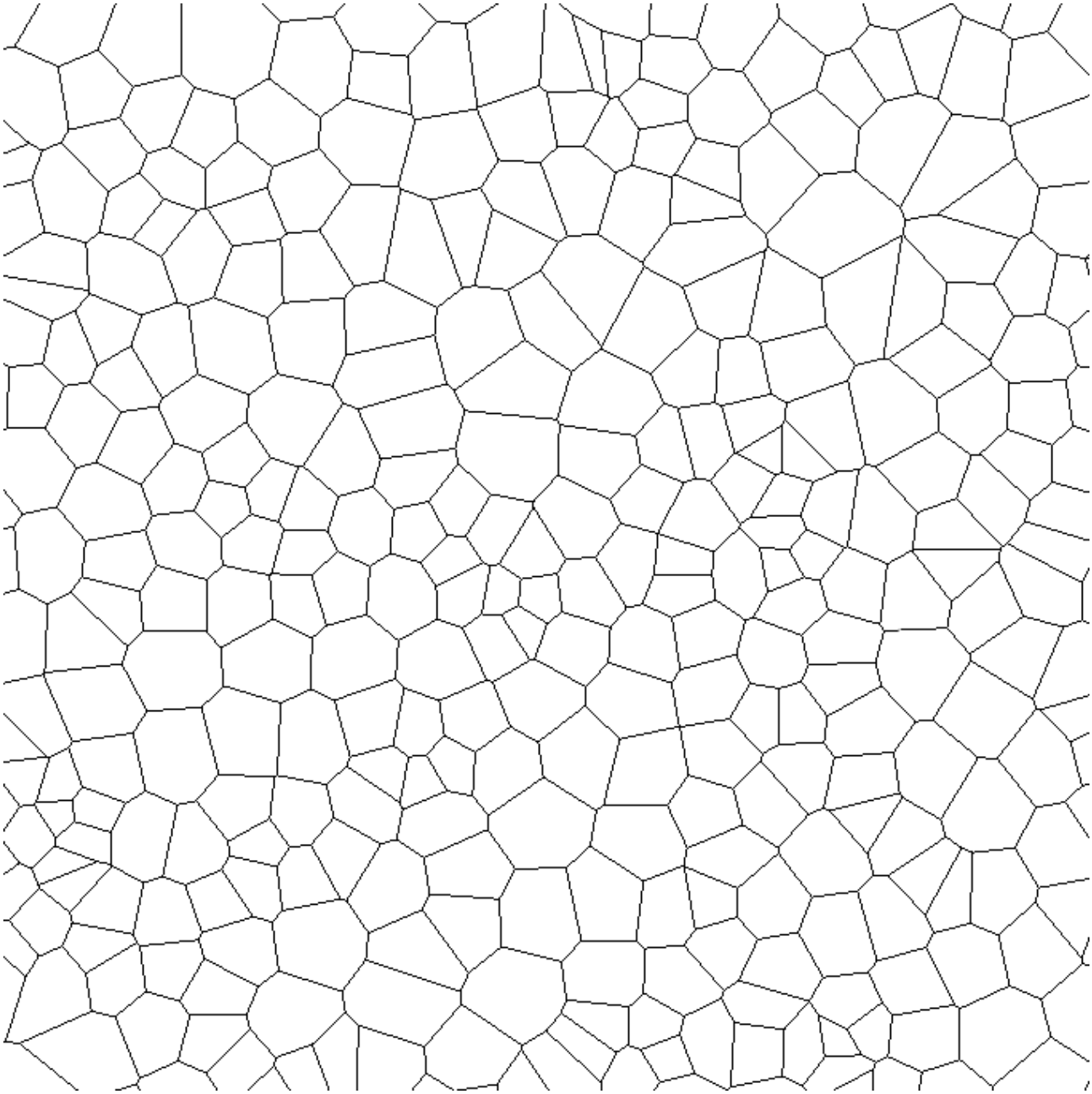


Figure D.2: Uniform random distribution with one iteration of Lloyd relaxation.

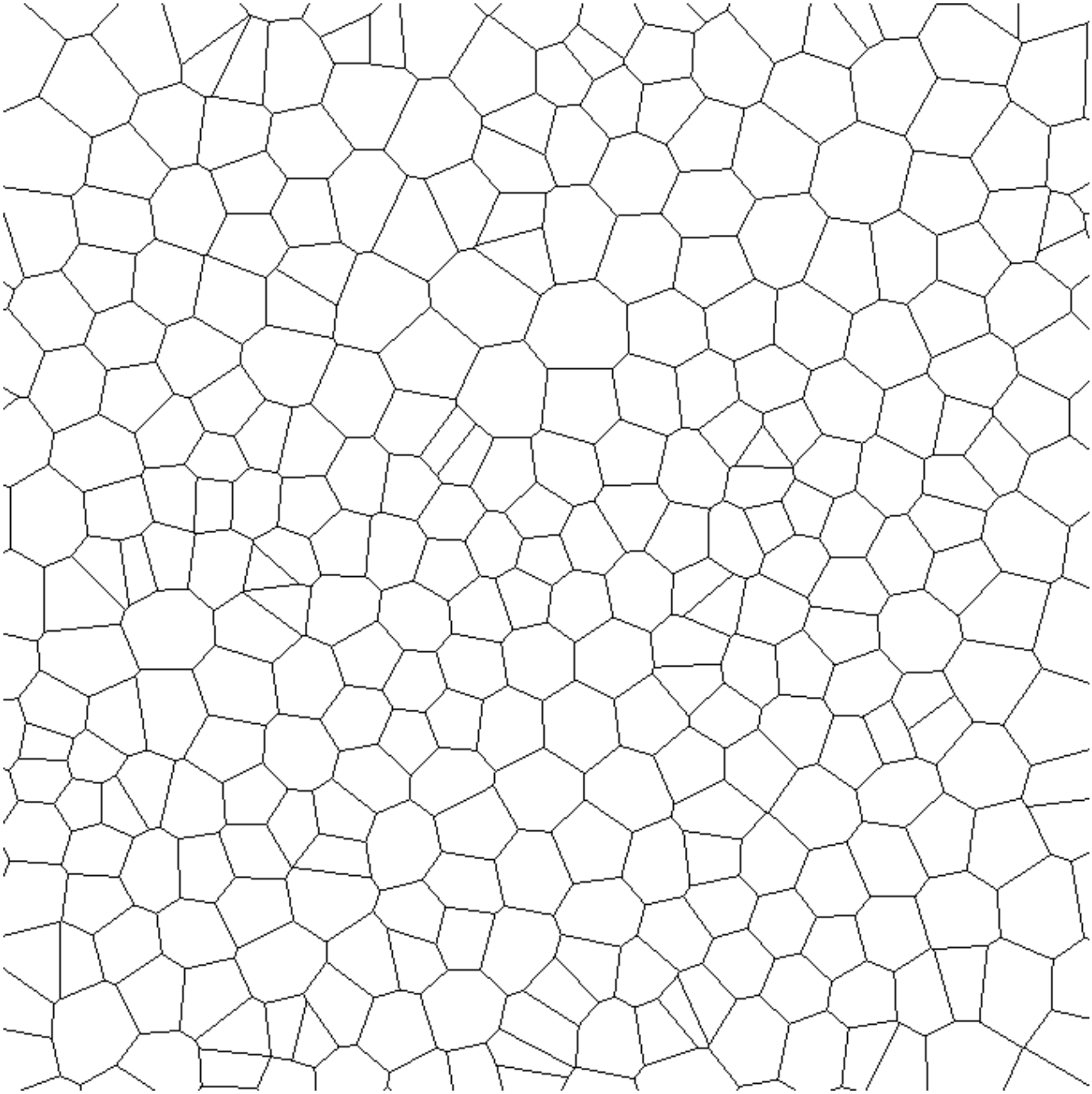


Figure D.3: Uniform random distribution with five iterations of Lloyd relaxation.

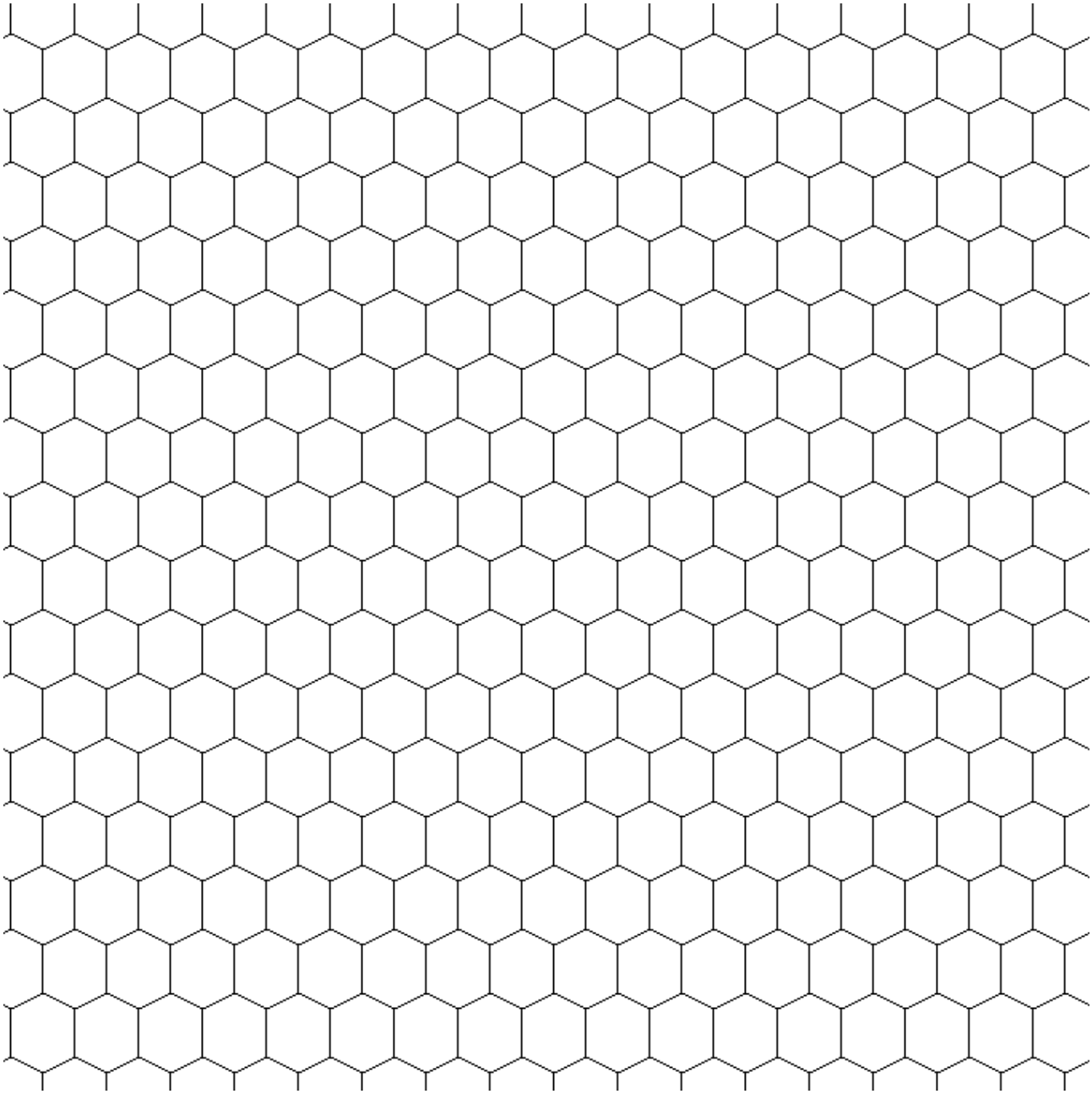


Figure D.4: Hexagonal grid.

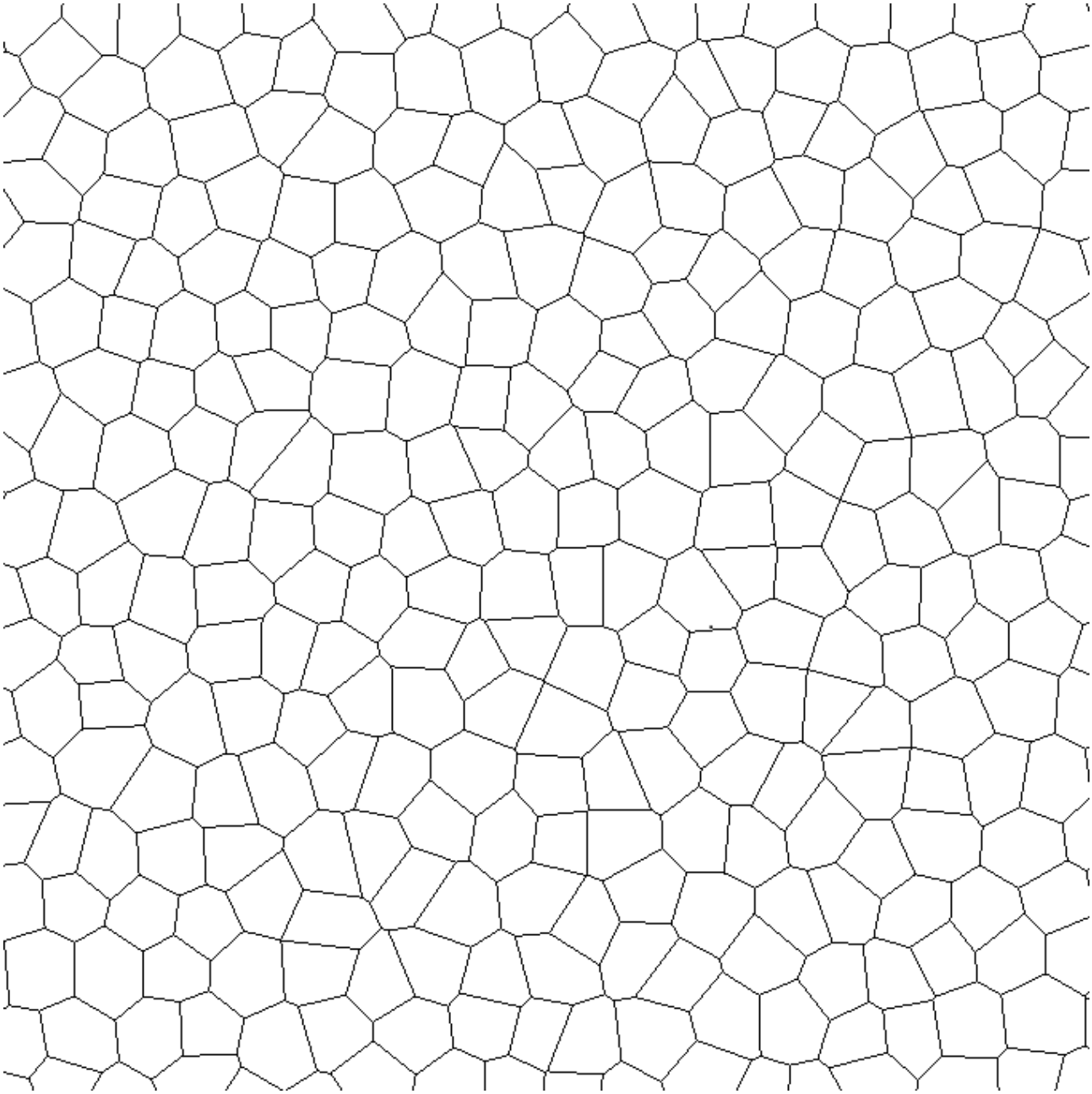


Figure D.5: Hexagonal grid with random scattering.