School of Science & Engineering

Procedural Generation of 3D Game Worlds

Capstone Design

EGR 4402 - Fall 2022

Final Report

Hajar Zaiz

Supervised by:

Naeem Nisar Sheikh

PROCEDURAL GENERATION OF 3D GAME WORLDS

Capstone Final Report

Student Statement:

I, Hajar Zaiz, have applied ethics to the design process and in the selection of the final proposed design. I have also held the safety of the public to be paramount and have addressed this in the presented design wherever may be applicable.

Hajar Zaiz

Approved by the Supervisor

Naeem Nisar Sheikh

Acknowledgements

I would like first to express my profound gratitude to my parents. Thank you for raising me the way you did and supporting me, both financially and emotionally. I am also grateful to my brother, the role from which I learned many valuable lessons.

I would like to extend my gratitude to professor Naeem Nisar Sheikh for his time and guidance not only during this project but during my whole undergraduate journey. I benefited enormously from his excellence as a teacher and as a mentor. I also thank Dr. Violetta Cavalli-Sforza, Dr. Asmaa Mourhir, Dr. Alina Khasanova, and Dr. Julio Bahamon for marking my learning experience.

Contents

List of Figures

List of Tables

List of Acronyms

2D 2 Dimensional

3D 3 Dimensional

CFG Context-Free Grammar

CPU Central Processing Unit

FBM Fractal Brownian Motion

GPU Graphics Processing Unit

LOD Level of Detail

PCG Procedural Content Generation

Abstract

With large-scale modern video games such as “Ghost of Tsushima” or even unbounded game worlds such as “Minecraft”, players are now faced with large and sometimes infinite exploration possibilities. Manually producing vast game worlds with such levels of detail requires costly resources that indie game developers might not have.

In this capstone project, I will explore the procedural content generation of realistic-looking game worlds, specifically ecosystems, using algorithms. The focus will mainly be on overcoming physical memory and CPU limitations. The goal is to produce a full-fledged ecosystem generator and allow game designers and developers to build on top of it to produce playable media for both entertainment and serious purposes.

Keywords: Procedural content generation, 3D ecosystems, Game design, Noise, Fractal Brownian Motion, L-systems, Hydraulic erosion

1 Introduction

Procedural content generation (PCG) refers to the algorithmic generation of content in media with limited human intervention [1]. PCG can be used to create 2D and 3D art, stories, dialogue, music, levels, maps, textures, and other game content. It provides large, interesting, and varied content that enhances the gameplay experience. Additionally, with the increasing use of affective gaming [2], PCG can be used to adjust the game design to personalize content [3] and maximize player satisfaction.

In video games, PCG has been used since the late 1970s but became famous in the early 1980s with games such as Elite by David Braben [4]. The latter procedurally generated entire galaxies with stars, planets, moons, and stations. In recent years, modern games such No Man's Sky [5] are able to procedurally generate infinite amounts of deterministic content. PCG also allows creators of serious games to focus more on the educational aspect rather than environmental design.

Figure 1.1: Left: Elite (Braben 1984). Right: No Man's Sky (Hello Games 2016).

One of the benefits of PCG is overcoming memory limitations, especially for console games. With vast game worlds being resource-intensive, storing algorithms and generating content at runtime is far more efficient than storing a myriad of meshes. It is also time-consuming for artists to generate large landscapes as duplication with slight tweaks can look unnatural. The content is also consistent

and adjustable with pseudo-random PCG allowing indie game studios to produce large game worlds in spite of budget limitations.

Although the advantages of PCG are numerous, it can be difficult to produce natural-looking results. The algorithms used for high-fidelity content production can be CPU costly which is impractical for real-time generation. This report will explore the algorithms used in PCG to produce realistic ecosystems while addressing the previously mentioned challenges. The results can be extended to urban game worlds by adding custom algorithms for buildings and roads.

2 Technology Stack

Concerning the technologies that are used in this project, I settled for C# with UNITY3D as the game engine. Contrary to UNREAL ENGINE 5, UNITY3D is a mature game engine with a larger community for obtaining advice concerning problems and improvements. Additionally, given my technical expertise and familiarity with UNITY3D I prefer to focus on the algorithms rather than picking up new technology. For version control, I used GITHUB to back up my code and keep track of commits.

To implement this project, I was using an *MSI GF65 Thin 9SD-004 15.6" 120Hz Gaming Laptop - Intel Core i7-9750H - GTX1660Ti* and settled for no ray tracing given the limitations of my graphic card. All the listed software and hardware required were available and reliable to ensure the timely delivery of the project.

3 Related Work

In studies closely related to this work, methods for automatically generating natural objects were investigated [6]. For terrain generation, noise maps are created using Fractal Brownian Motion (FBM) [7] based on 2D Perlin noise which generates layers of noise by interpolating points in a random vector grid. Whereas for plants, the Lindenmayer system [8] is utilized as a context-free grammar to simulate the complex branching of trees.

Another work developed a tool named Charack for the real-time generation of pseudo-infinite 3D worlds [9]. The focus of the latter was on achieving a smooth transition between heightmap blocks while optimally storing data for a large-scale world. The continents map was created using an external tool [10] and information about terrain types was stored in a macro-matrix (MM). To optimize memory utilization, an MM's entry maps to multiple vertices in the virtual world. According to the terrain type, height values are applied for each vertex in the 3D world. The result is rendered as a triangles mesh that is texturized according to the height value of each vertex in the mesh. Coastline algorithms were applied when the vertex is mapped to an offshore entry in the MM for a smooth transition between water and land and mini islands were spawned to disturb the sea.

On top of fractal noise, a technique called "level of detail" (LOD) is used to optimize the GPU's utilization in large-scale worlds [11]. With LOD, the number of triangles is reduced in meshes that are out of the player's field of vision.

Other authors used deep convolutional generative adversarial networks to generate maps [12], but it's yet to result in landscapes as visually appealing as the ones obtained using Perlin noise. However, such machine learning methods proved efficient in 2D level generation [13].

4 Design & Implementations

4.1 Noise Functions

Noise functions in computer graphics are sequence generators of random or pseudo-random values between 0 and 1. It was initially developed in the mid-80s when computers had limited memory and images were inefficient for texturing. This section describes white noise, value noise, Perlin noise, Simplex noise, and fractal noise. For visualization purposes, the noises are displayed in 2D.

4.1.1 White Noise

White noise functions use random number generators to obtain noise values between 0 and 1. As a result, spatially close points can have values of high variance which is uncommon in patterns in nature. In the latter, points that are close to each other are similar while points that are far apart tend to differ. Reproducing similar patterns with different function calls can also be problematic with white noise due to its unpredictability. Therefore, it is unsuitable for texturing and height maps.

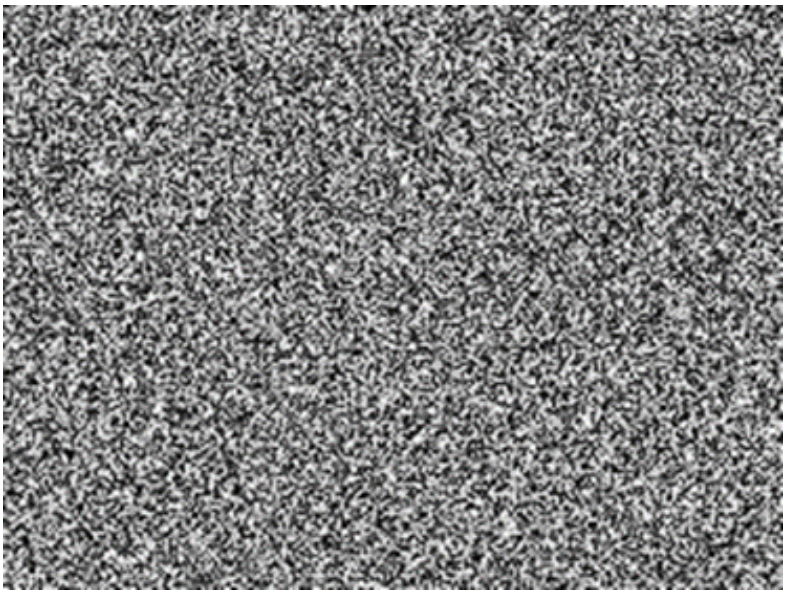

Figure 4.1: White noise

4.1.2 Value Noise

Value noise creates random values spaced at regular intervals. In 2D, we start by generating a grid. We then generate random floats in the interval $[0, 1]$ at the vertex positions of the grid as shown in Figure 4.2.

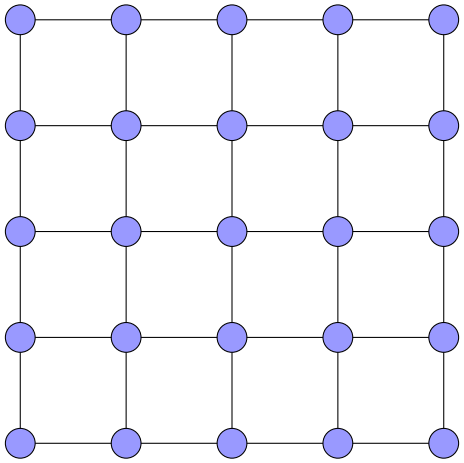

Figure 4.2: The random values in 2D value noise at vertex positions

To obtain the remaining points' values inside each cell we will bilinearly interpolate using the surrounding cell vertices. The result is a noise full of visual artifacts as seen in Figure 4.3.

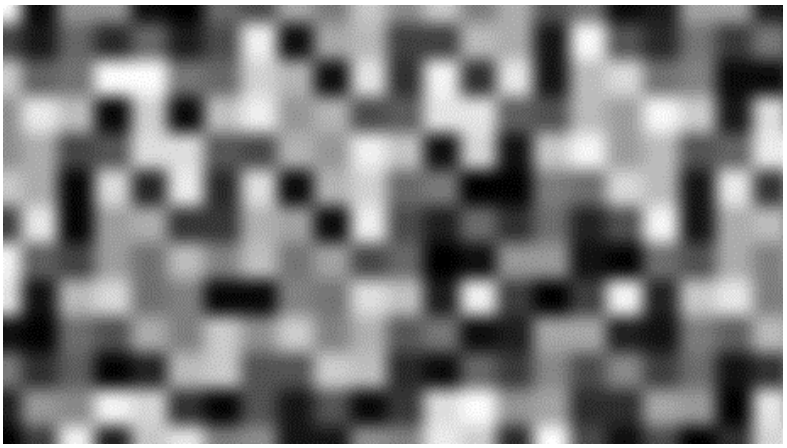

Figure 4.3: One interation of 2D value noise

4.1.3 Perlin Noise

Perlin noise [14] is a gradient noise developed by Ken Perlin in 1983 and improved in 2001. To generate the noise we create a $(width \times height)$ grid. The algorithm is then called for each (x, y) grid vertex coordinate and returns a float value in the range $[0, 1]$.

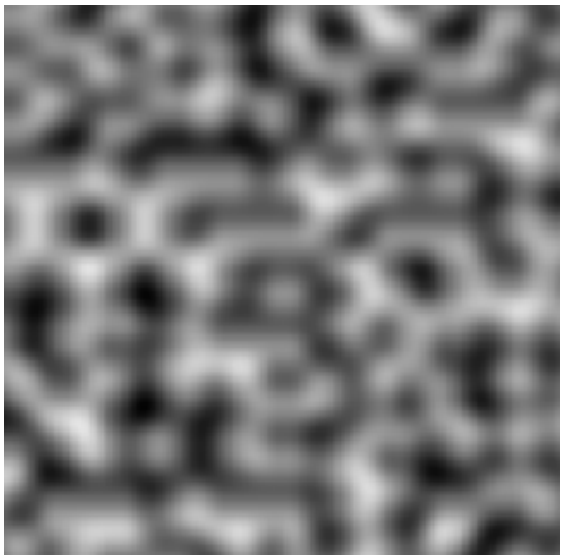

Figure 4.4: Perlin noise

For each input coordinate, we take modulus 1 to obtain the unit square. We then generate a pseudo-random gradient vector for the corners of the unit square.

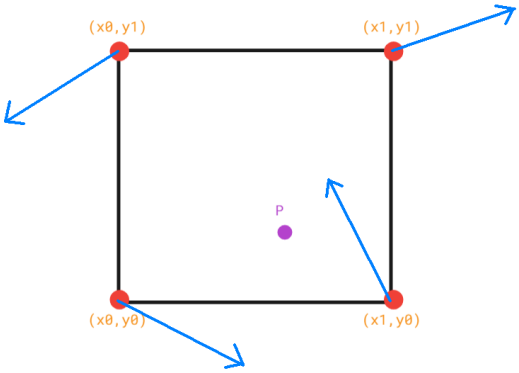

Figure 4.5: Unit square with 4 pseudo-random gradient vectors

To remove directional bias, those gradients are picked from the following pool of vectors to avoid the direction of coordinate axes and long diagonals:

$$\begin{cases} (1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0) \\ (1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1) \\ (0,1,1),(0,-1,1),(0,1,-1),(0,-1,-1) \end{cases} \tag{4.1}$$

For each point P inside the unit square, we compute the dot product between the gradient vector in each corner and the distance vector between P and that corner. We then linearly interpolate the four resulting products.

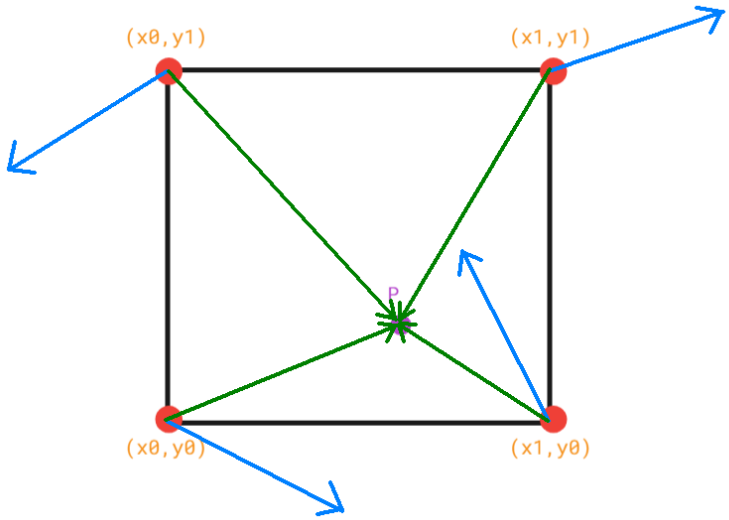

Figure 4.6: Gradient vectors in blue and distance vectors in green for point P

The output of the previous step is a noise full of interpolation artifacts. To make the transition between grid cells smoother, we apply the fade function:

$$6t^5 - 15t^4 + 10t^3 \tag{4.2}$$

Figure 4.7: Fade Function 4.2

By using gradients instead of random values, the distribution of frequencies is more regular than that of value noise.

4.1.4 Simplex Noise

Simplex noise [15] is also a gradient noise developed by Ken Perlin. Unlike Perlin noise which relies on a hypercubic grid, this noise implementation is based on a simplex grid. A simplex is the simplest polyhedron in geometry. A simplex is a triangle in 2D, a tetrahedron in 3D, etc.

With a grid of equilateral triangles, we wish to obtain the relative coordinates of our map points. We do that by matching our noise map to a square region. Afterwards, we recover the original coordinates by skewing each cell from square to rhombus. Since the skewing is performed by translating the points along the XY diagonal, we have to subtract some skew value **s**.

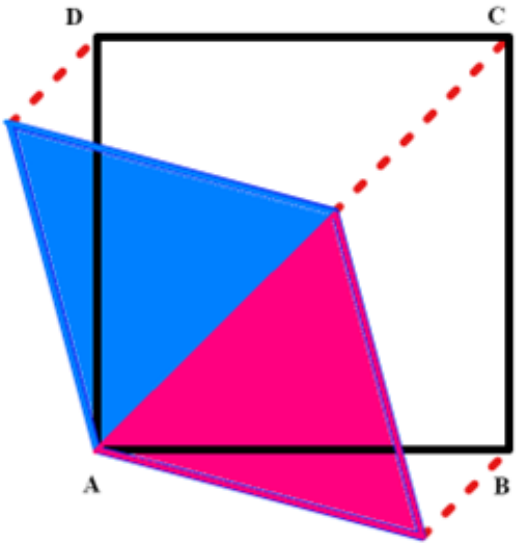

Figure 4.8: Skewing from square to rhombus

First, notice that in Figure 4.8 A with initial coordinates (0,0) remains the same, while B and C with initial coordinates (1,0) and (1,1) respectively are skewed. Therefore, the skewing depends on the coordinates in addition to the skewing factor **s**. We obtain:

$$B = \begin{bmatrix} 1 - s(x_B + y_B) \\ 0 - s(x_B + y_B) \end{bmatrix} = \begin{bmatrix} 1 - s(1+0) \\ 0 - s(1+0) \end{bmatrix} = \begin{bmatrix} 1 - s \\ -s \end{bmatrix} \tag{4.3}$$

and

$$C = \begin{bmatrix} 1 - s(x_C + y_C) \\ 1 - s(x_C + y_C) \end{bmatrix} = \begin{bmatrix} 1 - s(1+1) \\ 1 - s(1+1) \end{bmatrix} = \begin{bmatrix} 1 - 2s \\ 1 - 2s \end{bmatrix} \tag{4.4}$$

with **s** being the skewing factor that depends on the coordinates. To find the skewing factor **s**, we solve for $||AB||^2 = ||AC||^2$, this yields $s = \frac{3-\sqrt{3}}{6}$. Similarly, to unskew and convert triangles to squares, we solve for the opposite direction and get the factor $u = \frac{\sqrt{3}-1}{2}$.

Using the triangular lattice as our starting point, we floor our skewed coordinates, to determine which cell contains our point and the relative coordinate. Then we compare the coordinates' magnitudes to determine whether our point lies in the

upper or the lower triangle.

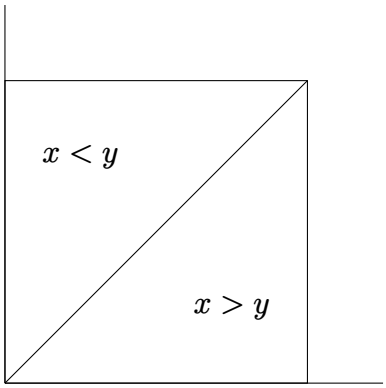

Figure 4.9: Finding the simplex where a point lies

Instead of linear interpolation, each vertex of the simplex is hashed into a pseudo-random gradient direction after unskewing and the corners contributions are summed, normalized, and returned.

Each corner contribution is computed as follows:

$$(0.5 - d^2)^4 \cdot (< \Delta x, \Delta y > \cdot < \nabla x, \nabla y >) \tag{4.5}$$

with d the distance to the point, and Δx and Δy the relative coordinates of the point in the simplex.

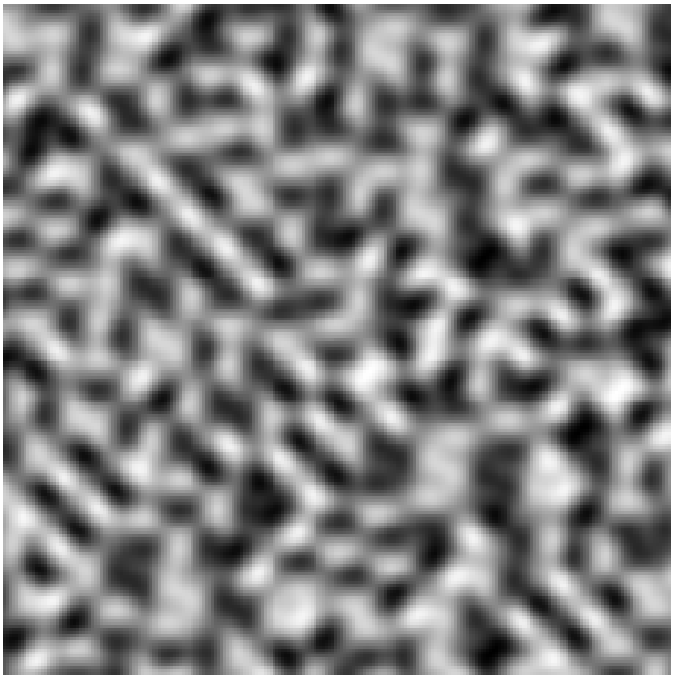

Figure 4.10: One iteration of 2D Simplex noise

4.1.5 Noise Algorithms Benchmark

The following is a summary of the previously mentioned noise algorithms and their efficiencies [16]:

Table 1: Efficiency Benchmark of Noise Algorithms

Algorithm	**Speed**	**Memory Requirements**	**Quality of results**
White Noise	Fast	Very Low	Very Low
Value Noise	Fast	Very Low	Low - Moderate
Perlin Noise	Moderate	Low	High
Simplex Noise	Moderate but Scalable	Low	Very High

To produce the best-looking results while taking into consideration memory utilization and fast real-time generation, Perlin Noise with a time complexity of $O(N \cdot 2^N)$ and Simplex Noise with a complexity of $O(N^2)$ for N dimensions are ideal for this project.

4.1.6 Fractal Brownian Motion

Fractal Brownian Motion (FBM) noise is obtained by layering similar or different noise maps. The number of iterations is referred to as **octaves**. In each octave, we can increase the frequency of noise thus increasing the level of details by setting a factor called **lacunarity**. A lacunarity of 2 means that each octave will have double the number of details of the previous one. Additionally, we can also decrease how much each octave contributes to the overall shape by setting a factor called **persistence**. A persistence of 0.5 means that each octave will have half the amplitude of the previous one.

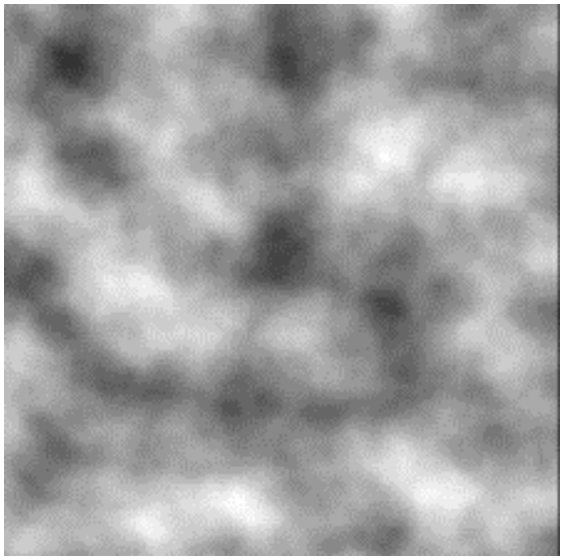

Figure 4.11: 7 Octaves of Perlin Noise with a lacunarity of 1.7 and a persistence of 0.5

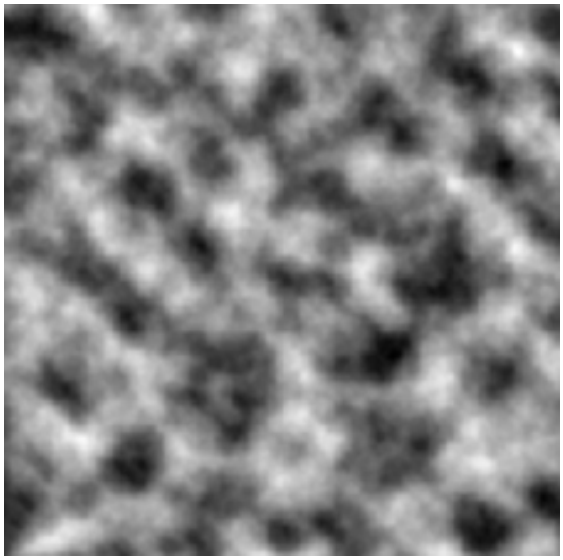

Figure 4.12: 7 Octaves of Simplex Noise with a lacunarity of 1.7 and a persistence of 0.5

4.2 Colour Map

To make the noise maps look like an ecosystem, I treat the pixel values as height levels in my environment and assign different colors to different height intervals.

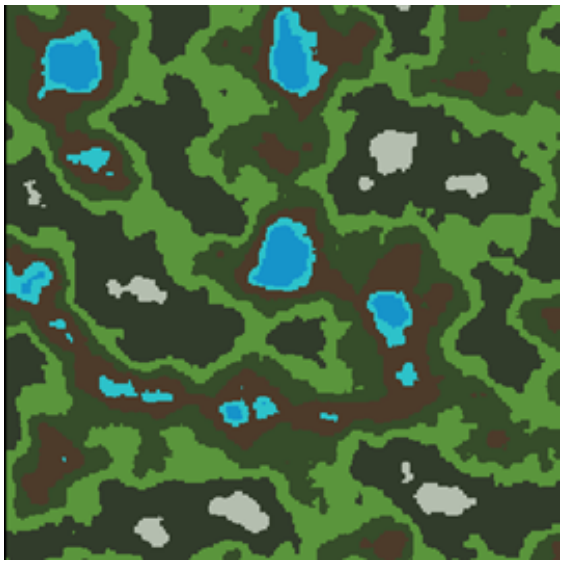

Figure 4.13: Colour Map of 2D Perlin Noise

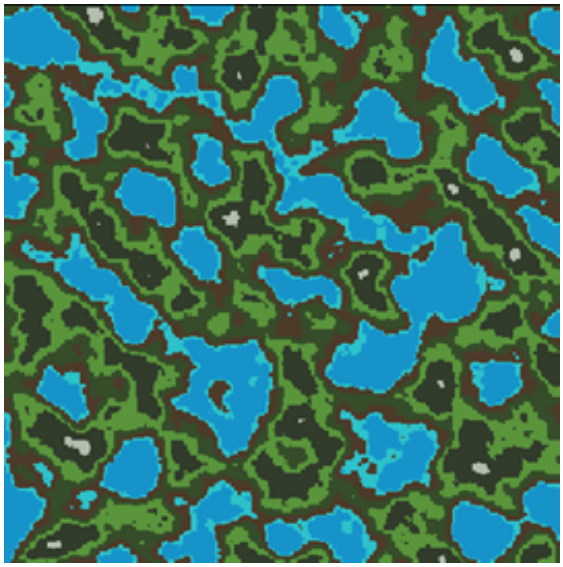

Figure 4.14: Colour Map of 2D Simplex Noise

One can notice from Figures 4.13 and 4.14 that the water-to-land ratio is not proportional. In nature, water is less frequent than what we notice with 2D Simplex noise and more frequent than the occasional pools of water obtained with 2D Perlin noise. To achieve a more realistic water to land ratio, I settled for 3 octaves of Perlin noise and 4 octaves of Simplex noise with a lacunarity of 1.7 and persistence of 0.4. The result can be shown in Figure 4.15.

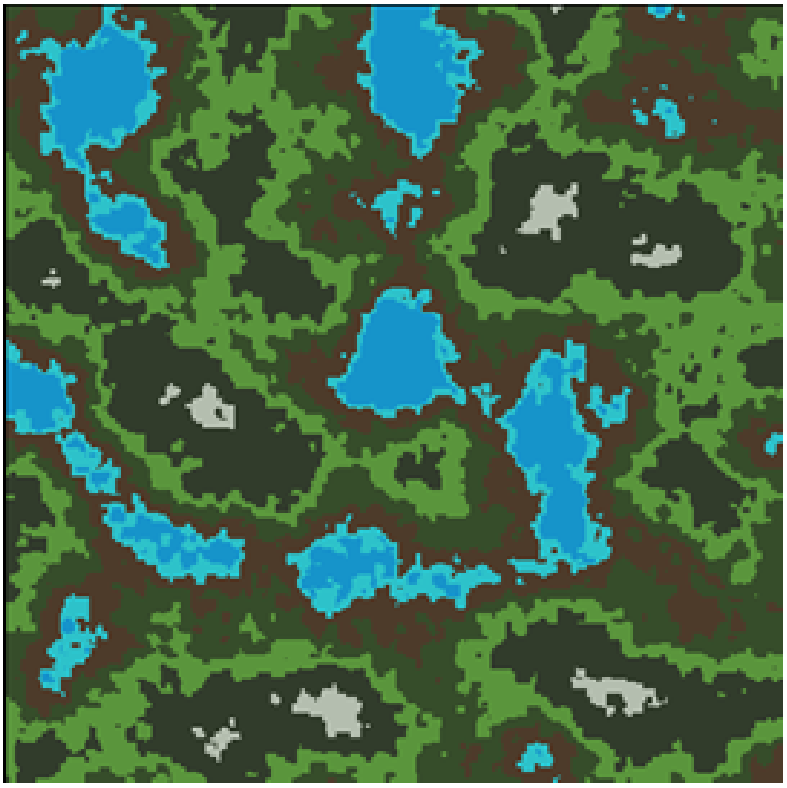

Figure 4.15: 3 iterations of Perlin noise and 4 iterations of Simplex noise

4.3 Mesh Generation

The next step is converting our 2D noise to a 3D mesh. In game engines, meshes are triangulated because every shape can be made out of triangles. The latter are always guaranteed to be flat. We do not have to worry about our points being co-planar which is computationally expensive to check nor how to apply textures to our mesh if the vertices are not co-planar.

To create a mesh with parameters width w and height h, the number of vertices is:

$$\text{Number}_{\text{Vertices}} = (w + 1) \cdot (h + 1) \tag{4.6}$$

Additionally, the number of triangle vertices making up the mesh is:

$$\text{Number}_{\text{Triangles}} = 6 \cdot w \cdot h \tag{4.7}$$

Given the previous parameters, we procedurally generate a sequence of lower and upper triangles to obtain our final mesh.

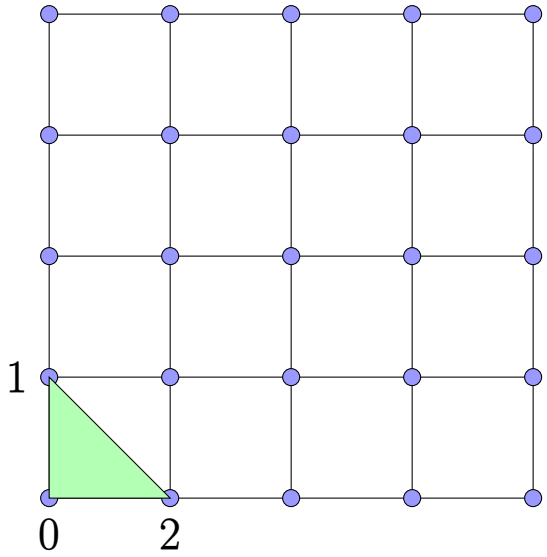

Figure 4.16: Mesh triangulation in game engines

Finally, we create an array that holds UV data, assign texture coordinates to it, recalculate normals, and create our mesh. To create a mesh in UNITY3D, one should pass the vertices, UVs, and triangles to it [17].

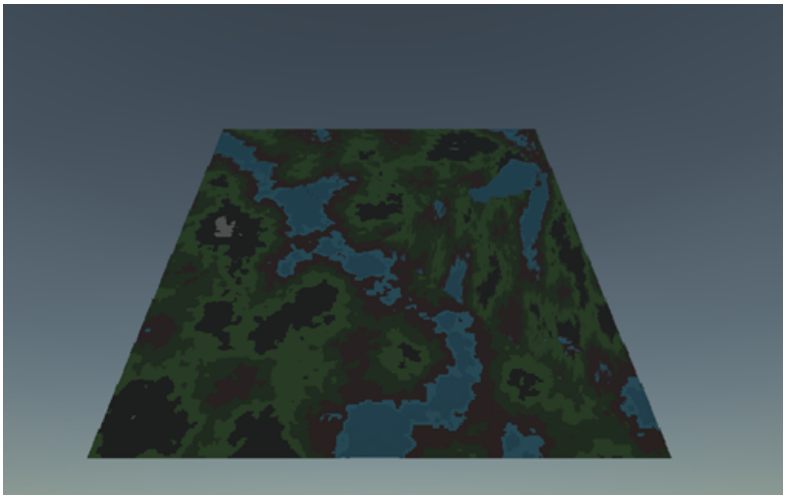

Figure 4.17: Converting the 2D map to a 3D mesh

The current mesh is flat as the heights are float values between 0.0 and 1.0. We multiply by a height multiplier and flatten vertices at sea level to obtain a realistic terrain.

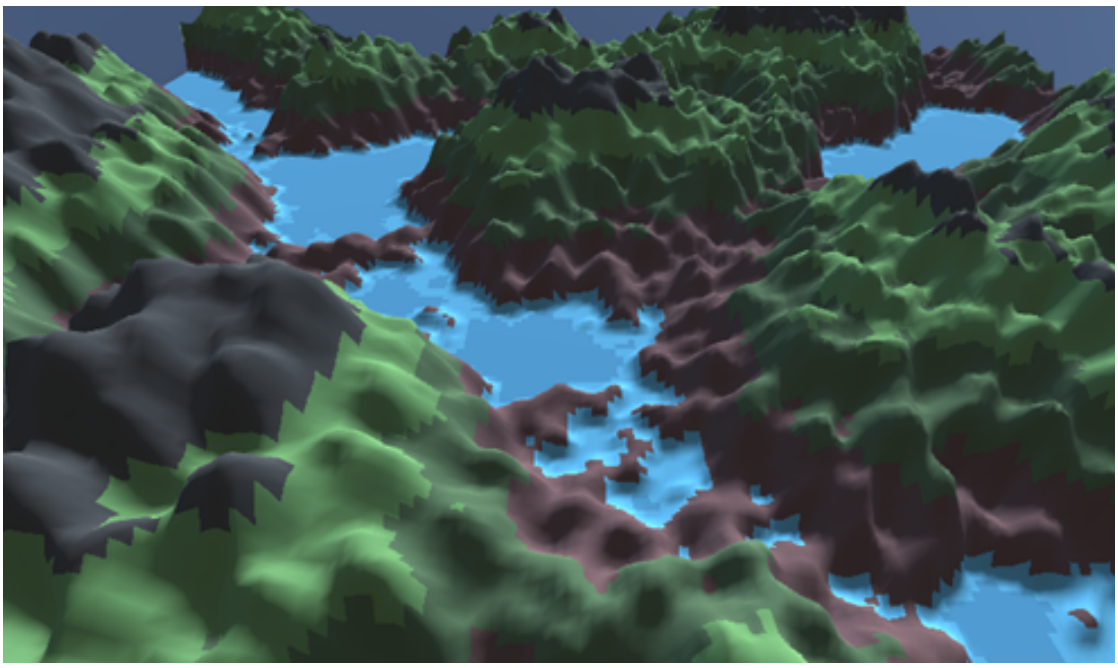

Figure 4.18: 3D mesh with a height multiplier

4.4 Erosion

Fractal noise on its own creates interesting landscapes. However, valleys and mountains appear to have similar characteristics whereas they are clearly distinguishable in the real world. Additionally, lower areas are not flat enough to navigate on. In order to add realism for a more immersive experience, erosion algorithms are applied [18]. Two of the most studied types of erosion are hydraulic and thermal erosion.

4.4.1 Hydraulic Erosion

Hydraulic erosion models water starting somewhere on the terrain and picking up soil as it flows downhill, deposits its carry as sediment, and eventually evaporates.

Over the years, two approaches were established to mimic fluid movement. A grid-based approach, also referred to as the Eulerian approach [19], and a particle-based approach, also referred to as the Lagrangian approach [20]. In the former, each cell tracks how much fluid is inside it and computes how water flows between cell points. This method produces better results but it can lead to ravines and has a higher computational time since the states of all cells have to be computed at each step. With the Lagrangian technique, the movement of water particles is simulated. Droplet information such as velocity, carry capacity, and position is stored and the particle is moved according to those properties. It is less accurate than the Eulerian approach but is computationally faster and more suitable for real-time terrain generation. Therefore, I chose to implement particle-based erosion following Hans Theobald Beyer's algorithm [21].

4.4.1.1 Droplet Parameters

Every water particle stores the following information:

- The particle's position as 2D vector.
- The flow direction as a 2D normalized vector.
- The speed of flow as a float and initially zero.
- The amount of water in the droplet as float.
- The amount of sediment carried as a float and initially zero.

At each step, we wish to move the drop to a local minimum by computing the gradient g of the current droplet position p_{old}, we obtain g by bilinearly interpolating the gradients of the surrounding cell points.

Let:

$$p_{\text{old}} = (x + u, y + v) \tag{4.8}$$

with u and v the relative coordinates of the droplet inside the cell. The gradients of the surrounding points $p_1(x, y)$, $p_2(x + 1, y)$, $p_3(x, y + 1)$, and $p_4(x + 1, y + 1)$ are the following:

$$\begin{cases} g(p_1) & = (p_2 - p_1, p_3 - p_1) \\ g(p_2) & = (p_2 - p_1, p_4 - p_2) \\ g(p_3) & = (p_4 - p_3, p_3 - p_1) \\ g(p_4) & = (p_4 - p_3, p_4 - p_2) \end{cases} \tag{4.9}$$

Using bilinear interpolation:

$$g(p_{\text{old}}) = \begin{pmatrix} (p_2 - p_1) \cdot (1 - v) + (p_4 - p_3) \cdot v \\ (p_3 - p_1) \cdot (1 - u) + (p_4 - p_2) \cdot u \end{pmatrix} \tag{4.10}$$

The resulting gradient g is then used to determine the new flow direction of the droplet along with an inertia value between 0 and 1. The latter decides how much the old direction and the gradient contribute to he new direction vector. A random direction is generated if the gradient is zero.

$$\text{direction}_{\text{new}} = \text{direction}_{\text{old}} \cdot p_{\text{inertia}} + g \cdot (1 - p_{\text{inertia}}) \tag{4.11}$$

Consequently, the updated position of the water droplet is computed using the new direction vector:

$$\text{p}_{\text{new}} = \text{p}_{\text{old}} + \text{direction}_{\text{new}} \tag{4.12}$$

If the new position falls outside the terrain borders, the droplet stops flowing.

4.4.1.2 Erosion and Deposition

We also compute the height difference between the old and new positions. If it is positive, the drop moved uphill and the carried sediment is deposited at the previous position. The deposited sediment should not exceed the height difference to avoid creating spikes.

$$\text{deposited} = \min(\text{sediment}, \text{height difference}) \tag{4.13}$$

The sediment is distributed along the four cell points surrounding the current position using bilinear interpolation.

If the height difference is negative, the drop moved downhill and could either erode or deposit sediment depending on the droplet carry capacity c. The carry capacity is high when the droplet goes down fast and carries a lot of water. To prevent the carry capacity from falling too close to 0 when the height difference is small, a minimum slope $p_{minSlope}$ value is selected. Additionally, the parameter p_{capacity} determines the amount of sediment a drop can carry.

$$c = max(-\text{height difference} \cdot p_{\text{minSlope}}) \cdot vel \cdot water \cdot p_{\text{capacity}} \tag{4.14}$$

If the droplet is carrying more sediment than the set capacity, a percentage of the sediment surplus is deposited at the old position. Dropping more than the surplus can result in spikes.

$$(\text{sediment} - c) \cdot p_{\text{deposition}} \tag{4.15}$$

If the carried sediment does not exceed the allowed capacity, we erode a percentage of the remaining capacity without exceeding the height difference to avoid pits.

$$min((c - \text{sediment}) \cdot p_{\text{Erode}}, -\text{height difference}) \tag{4.16}$$

Similarly to deposition, I erode evenly the corners of the grid cell using bilinear interpolation.

After each step of the droplet flow, the speed is updated with more emphasis on speed inheritance and the water gradually evaporates.

$$\text{vel}_{\text{new}} = \sqrt{\text{vel}_{\text{old}}^2 + h_{\text{diff}} \cdot p_{\text{gravity}}} \tag{4.17}$$

$$\text{water}_{\text{new}} = \text{water}_{\text{old}} \cdot (1 - p_{\text{evaporation}}) \tag{4.18}$$

To ensure that the droplet does not flow endlessly, a maximum number of steps is assigned per drop.

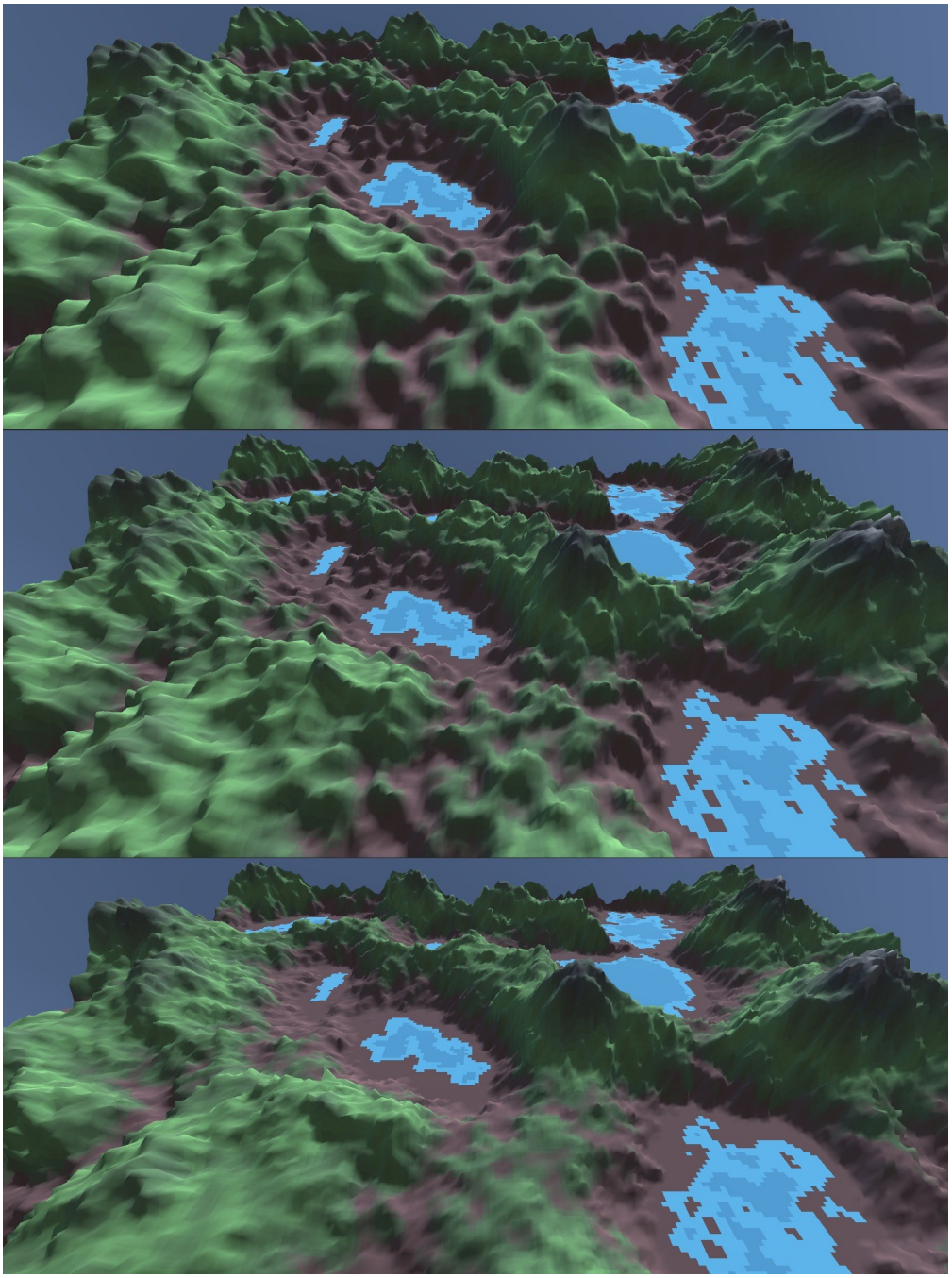

Figure 4.19: No erosion, 30K droplet with a maximum path of 5, and 30K droplet with a maximum path of 20

4.4.2 Thermal Erosion

Thermal erosion models the erosion of steep cliffs that crack up due to temperature changes and fall down [22]. If the height difference between the current position and neighboring grid points exceeds a set value T, the soil is eroded from the current position and deposited into the lower points. While the algorithm is quick and simple, the results are not realistic. Therefore, only hydraulic erosion is implemented in this project.

4.5 L-Systems

In 1968, Hungarian biologist and botanist Lindenmayer proposed a context-free grammar (CFG) called L-systems [23]. It describes the development of simple cellular organisms such as algae. It was originally inspired by the self-similarity that exists in nature or what Mandelbrot called Fractals. A fractal object can be produced by repeating a pattern recursively. With the recursive branching in trees, L-systems are suitable to model them.

Our L-systems formal grammar consists of:

- **Terminals:** The language's alphabet.
- **Nonterminals:** Other symbols used in the grammar.
- **Axiom:** The initial state or start symbol.
- **Production rules:** Rules used to generate sentences.
- **Number of iterations**

For example, given the following symbolic alphabet:

- **'F'**: Draw a line forward of length d.
- **'+'**: Rotate right by δ degrees.
- **'-'**: Rotate left by δ degrees.

- '[': Store the current state by pushing into a log stack.
- ']': Recover the last stored state by popping from the log stack.

The axiom is set to be X (Nonterminal) and the production rules are defined as follows:

$$\begin{cases} X \to [FF[+XF - F + FX] - -F + F - FX] \\ F \to FF \end{cases} \tag{4.19}$$

Additionally, with $\delta = 30°$ we get the following results with 3 iterations:

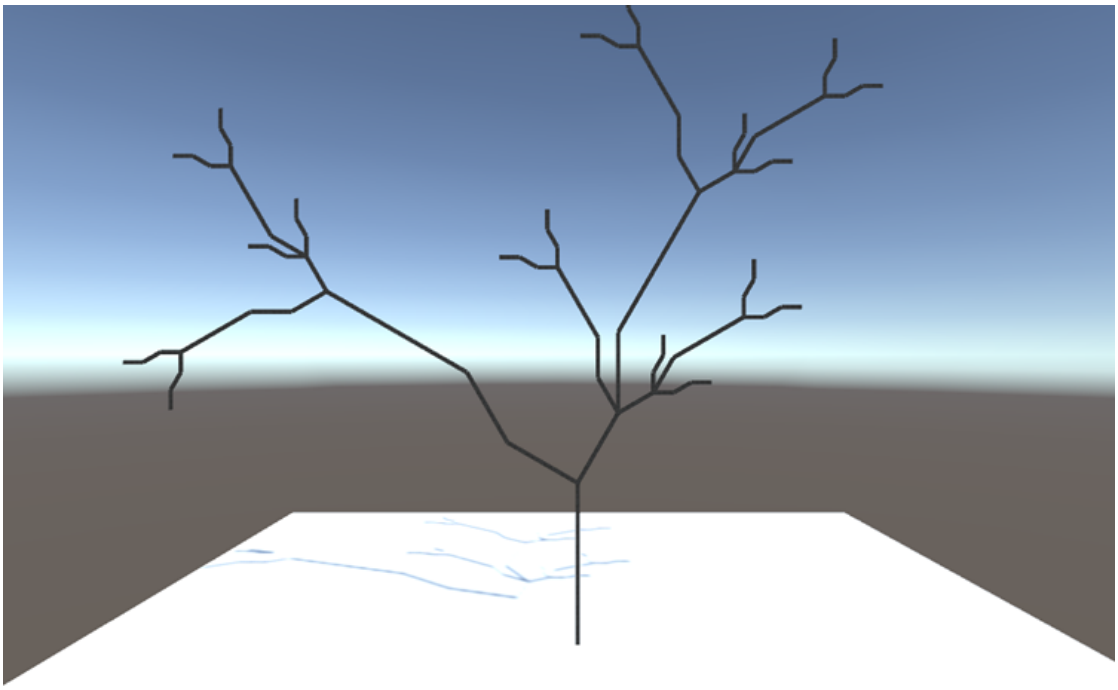

Figure 4.20: Sample tree based on rules 4.19

By applying different rules we can spawn different types of trees.
For instance, while keeping the same angle and changing the production rules to:

$$\begin{cases} X \to F - [[X] + X] + F[+FX] - X \\ F \to FF \end{cases} \tag{4.20}$$

We get the following results with 4 iterations:

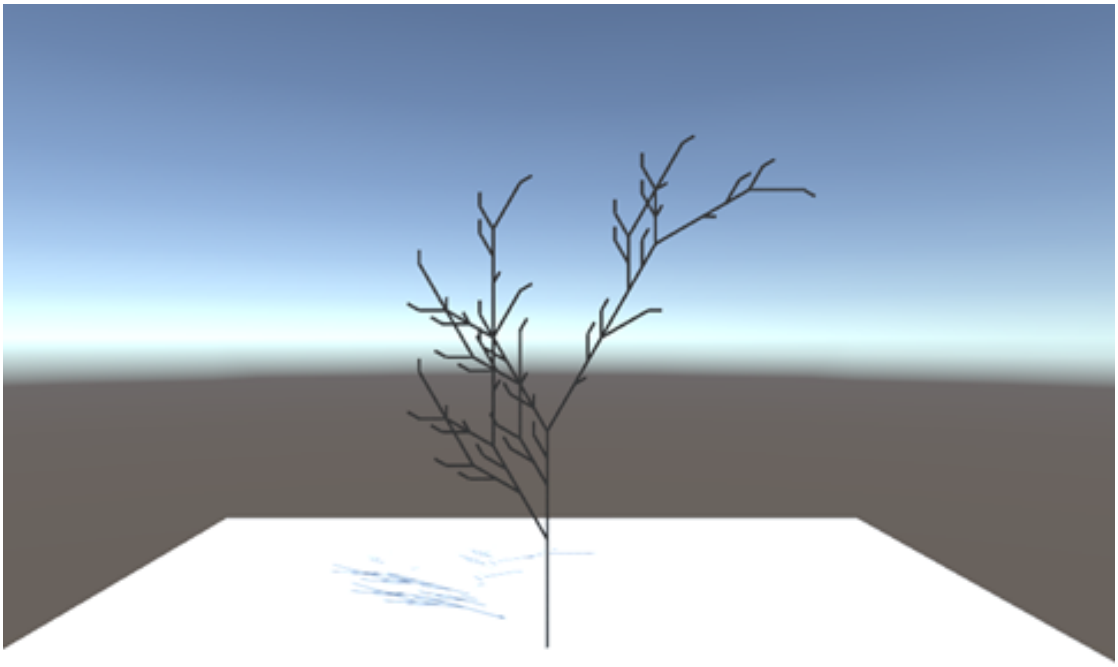

Figure 4.21: Sample tree based on rules 4.20

Given the time restrictions, adding more details such as leaves, angling branches in more than 3 directions, and placement algorithms were not explored.

5 STEEPLE Analysis

The STEEPLE analysis tool assesses this project's implications on society as a whole. The following is the STEEPLE analysis for PCG in ecosystems:

5.1 Social Impact

This project on its own has no social impact but can be extended to create serious games. For instance, in healthcare, access to rehabilitation facilities is a challenge for low-income households and rural areas [6]. With the increase in disabilities caused by autoimmune diseases and injury [7], patients displayed more motivation and engagement when rehabilitation exercises were performed in a game environment [8]. PCG is now being used in research for physical rehabilitation programs and allows for the generation of new interesting and personalized content that keeps the patient engaged [9].

5.2 Technological Impact

The technological impact lies in improving the replayability of media while minimizing CPU and memory utilization. This work will allow game studios and independent developers to introduce variance to their projects while maintaining some control over the output.

5.3 Economical Impact

This project will be open-source nonprofit software. It can be used for educational purposes or for other individuals to use as a building block in their own projects.

5.4 Environmental Impact

This project on its own has no environmental impact on its own but can be extended to create educational serious games about issues such as climate change.

5.5 Political Impact

This project does not have any political implications.

5.6 Legal Impact

This project does not have any legal implications.

5.7 Ethical Impact

This project will abide by the morals and legal code when it comes to intellectual property.

6 Engineering Standards

No relevant engineering standards were found for game engines, scene generation, and game design. Additionally, since the purpose of the project is to explore and implement algorithms related to scene generation, engineering standards for commercially-released games are not relevant. The only standard we are concerned with is ISO/IEC 23270:2003 Information technology — C# Language Specification.

7 Final Remarks

7.1 Challenges

The main challenge I faced in the course of my capstone is getting familiar with shader programming. One of the goals I had to give up was texture mapping. While I managed to assign different colors to different heights as seen in 7.1, I was not successful with texture interpolation.

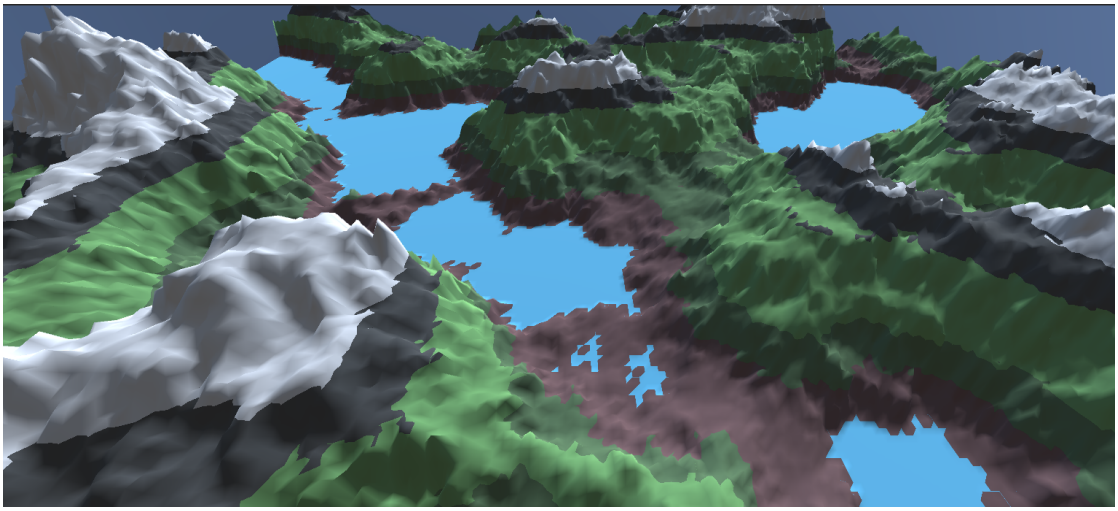

Figure 7.1: Colouring with shaders

Another challenge was the lack of resources needed to understand complex algorithms such as the Simplex Noise. Most resources I stumbled upon did not explain kernel summation and skewing. It took me numerous iterations to fully grasp how the algorithm works.

Finally, although my work was inspired by [21] and [6]. The proposed approaches sometimes did not suit my terrain design. Therefore, it was through trial and error that I obtained the final game world. Given the lack of community support for bugs and errors, the development phase ended up taking longer than anticipated.

7.2 Future Work

More work has to be done to make the environment navigable from a first-person perspective. Future work will focus on the following:

- **Shader Programming:** To ensure a more immersive experience, texture mapping, and water animations should be added.
- **3D L-systems:** Expanding the branching angles and adding terminals for leaves and flowers. Algorithms to procedurally place trees around should also be explored (e.g a variation of Poisson disk sampling with an increased density near water sources).
- **Pseudo-infinite generation:** Inspired by Charack [9], I wish to optimally store data for a large-scale world. Since we are only storing the seed for each mesh, the smooth transition between different meshes is the main concern for this section.

7.3 Conclusion

This project explored the creation of a full-fledged procedurally generated natural 3D game world. To achieve this, I generated my 2D height maps by using fractal noise that combines layers of Perlin noise and Simplex noise. I then convert my 2D map to a triangulated mesh by assigning vertices, UVs, and triangles, and recalculating the normals. The landscape is refined further using particle-based hydraulic erosion and L-systems grammar is used to procedurally generate trees. This project shows that PCG is a valuable tool for creating game content. It cuts down the development time and cost for indie game developers and allows artists and designers to focus on game mechanics, stories, and other game components.

Overall, this was a wonderful learning experience where I got to explore the field of computer graphics. I am also eager to further develop this project and explore more biotic and abiotic components in my ecosystem, such as animals and atmosphere.

References

[1] J. Togelius, E. Kastbjerg, D. Schedl, and G. Yannakakis, "What is procedural content generation? mario on the borderline," Jan. 2011. DOI: 10.1145/2000919.2000922.

[2] I. Kotsia, S. Zafeiriou, and S. Fotopoulos, "Affective gaming: A comprehensive survey," *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2013. DOI: 10.1109/cvprw.2013.100.

[3] G. Smith, E. Gan, A. Othenin-Girard, and J. Whitehead, "Pcg-based game design," *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games - PCGames '11*, 2011. DOI: 10.1145/2000919.2000926.

[4] D. Braben and I. Bell, *Elite*, 1984.

[5] *No man's sky (ps4 game)*, 2015.

[6] T. Gao and J. Zhu, "A survey of procedural content generation of natural objects in games," *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, 2022. DOI: 10.1109/icaiic54071.2022.9722677.

[7] F. K. Musgrave, C. E. Kolb, and R. S. Mace, "The synthesis and rendering of eroded fractal terrains," vol. 23, no. 3, 1989, ISSN: 0097-8930. DOI: 10.1145/74334.74337.

[8] E. W. Hidayat, I. K. Putra, I. A. Giriantari, and M. Sudarma, "Visualization of a two-dimensional tree modeling using fractal based on l-system," *IOP Conference Series: Materials Science and Engineering*, vol. 550, no. 1, p. 012 027, 2019. DOI: 10.1088/1757-899x/550/1/012027.

[9] F. Bevilacqua, C. T. Pozzer, and M. C. d'Ornellas, "Charack: Tool for real-time generation of pseudo-infinite virtual worlds for 3d games," *2009 VIII Brazilian Symposium on Games and Digital Entertainment*, 2009. DOI: 10.1109/sbgames.2009.21.

[10] T. Mogensen, *Instant planet generator*, 2009. [Online]. Available: http://www.eldritch.org/erskin/roleplaying/planet.php.

[11] M. Kahoun, "Realtime library for procedural generation and rendering of terrains," *Master's thesis*, 2013.

[12] A. Wulff-Jensen, N. N. Rant, T. N. Møller, and J. A. Billeskov, "Deep convolutional generative adversarial network for procedural 3d landscape generation based on dem," *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 85–94, 2018. DOI: 10.1007/978-3-319-76908-0_9.

[13] A. Summerville, S. Snodgrass, M. Guzdial, *et al.*, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018. DOI: 10.1109/tg.2018.2846639.

[14] K. Perlin, "Improving noise," vol. 21, no. 3, 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566636. [Online]. Available: https://doi.org/10.1145/566654.566636.

[15] S. Gustavson, "Simplex noise demystified," Aug. 2015. DOI: 10.13140/RG.2.1.3369.6488.

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

[17] U. Technologies, *Example: Creating a quad.* [Online]. Available: https://docs.unity3d.com/Manual/Example-CreatingaBillboardPlane.html.

[18] B. Jákó, "Fast hydraulic and thermal erosion on the gpu," *Proceedings of the CESCG 2011: The 15th Central European Seminar on Computer Graphics*, 2011.

[19] B. Beneš, V. Těšínský, J. Hornyš, and S. K. Bhatia, "Hydraulic erosion," *Computer Animation and Virtual Worlds*, vol. 17, no. 2, pp. 99–108, 2006. DOI: 10.1002/cav.77.

[20] A. Volynskov. [Online]. Available: http://ranmantaru.com/blog/2011/10/08/water-erosion-on-heightmap-terrain/.

[21] H. T. Beyer, "Implementation of a method for hydraulic erosion. bachelor's thesis," *Technische Universität München*, 2015.

[22] A. Travis, "Procedurally generating terrain," *44th annual midwest instruction and computing symposium*, 2011.

[23] R. Sun, J. Jia, and M. Jaeger, "Intelligent tree modeling based on l-system," Nov. 2009. DOI: 10.1109/CAIDCD.2009.5375256.

[24] L. Magnusson, I. Kebbie, and V. Jerwanska, "Access to health and rehabilitation services for persons with disabilities in sierra leone – focus group discussions with stakeholders," *BMC Health Services Research*, vol. 22, no. 1, 2022. DOI: 10.1186/s12913-022-08366-8.

[25] K. A. Theis, A. Steinweg, C. G. Helmick, E. Courtney-Long, J. A. Bolen, and R. Lee, "Which one? what kind? how many? types, causes, and prevalence of disability among u.s. adults," *Disability and Health Journal*, vol. 12, no. 3, pp. 411–421, 2019. DOI: 10.1016/j.dhjo.2019.03.001.

[26] J. E. Deutsch, M. Borbely, J. Filler, K. Huhn, and P. Guarrera-Bowlby, "Use of a low-cost, commercially available gaming console (wii) for rehabilitation of an adolescent with cerebral palsy," *Physical Therapy*, vol. 88, no. 10, pp. 1196–1207, 2008. DOI: 10.2522/ptj.20080062.

[27] D. Dimovska, P. Jarnfelt, S. Selvig, and G. N. Yannakakis, "Towards procedural level generation for rehabilitation," *Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCGames '10*, 2010. DOI: 10.1145/1814256.1814263.