Procedural Generation of Biome with Crystals

TNM084 - Procedural Images

Måns Aronsson - manar189

January 16, 2022

Contents

1 Introduction

The concept of noise is well known, but few think about its properties and uses. Noise can be thought of as a collection of random values, coupled to some input parameter(s). Unintuitively, creating random numbers is not an easy task for digital computers due to their deterministic architecture. Therefore, unless input parameters are random (created by some outside factor), such functions are classified as *pseudo-random*. However, pseudo-randomness is not necessarily a bug but a feature for certain applications. A random but repeatable result is often what is desirable when procedurally generating images, objects and terrains. Noise can have many different properties - variance, frequency, amplitude, gain etc. - that when adjusted properly is useful for different tasks. This project aims to use noise to produce a deterministic procedurally generated terrain with different biomes. One of these biomes should also contain deterministic procedurally generated crystals. The biome containing crystals should also include caves in which these crystals are generated on the floor, walls and ceiling.

The aim was formalized with the following goals:

- Procedurally generated crystals where properties such as shape, size and number varies
- Generated crystals should be deterministic

These extra goals were also defined and should be implemented if time allows:

- Cave generation using the Perlin worm technique
- Creating shaders that makes the crystals look good by e.g. making the refract light and lighting their surrounding
- Different biomes, e.g. desert and savanna

Code version control was done with Git and Github. The source code can be found under the branch *crystal* and the following link: `https://github.com/engbergandreas/ProceduralGeneratedTerrain/tree/crystal`.

2 Related Work

The following project is a continuation the project "Real Time Rendering of Procedural Terrain" that was implemented for the course TSBK03 - Advanced Game Programming this same semester. That project was a collaboration with Andreas Engberg, while this addition was solely implemented by the author. The source code and report for the previous project can be found under the *main* branch in the given Github project linked in Section 1. Therefore, this report is focused on concepts and implementation that were not covered in the previous report. It is advisable to read the report for the previous project before this one, to get a complete understanding of the program; the previous report handles key topics such as coding environment, fractional Brownian motion (FBM), frustum culling, discrete level of detail (LOD) and CPU multi-threading. One change to point out is that all uses of Perlin noise have been changed to Simplex noise due to its benefits in computational complexity and removal of directional artifacts [3]. Another problem that was improved upon was the CPU-threading architecture that sometimes produces lag spikes when all available CPU-threads were used by the program.

2.1 Perlin Worms

Perlin worms is a techniques that utilizes the Perlin noise function to create worm-like structures. This is done by taking small steps in directions decided by the noise [1]. This technique can be used to create caves in terrain by carving out the worm area. Smooth caves can be created by choosing parameters that creates a low frequency noise.

3 Implementation

Since this project is a direct continuation of a previous project, a notable amount of work had to be done to improve already implemented functions, and to change code to fit new functionality. This will however not be described in detail. This section focuses on new concepts that were implemented to achieve the project goals, with the exception of the improved CPU multi-threading architecture.

3.1 Crystal Generation

Crystals were implemented to have random features which could generate a wide variety of looks. However, it was important that the basic recognisable crystal structure stayed consistent throughout all variations. A crystal object was created with its world coordinates (spawn position) and direction as input parameters. It consisted of three sections - base, middle and top - where each section contained a number of vertices that all were the same distance from the spawn position along the given direction. The position and direction was then multiplied and used to generate five random values with simplex noise of different frequencies. These were then used to randomize five crystal features as described in Figure 1.

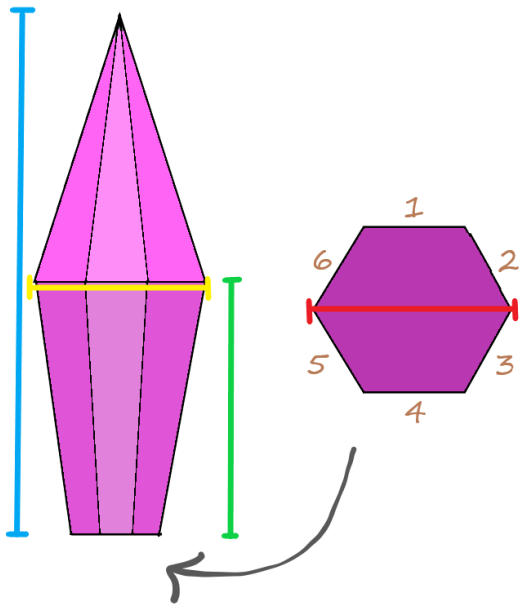

Figure 1: Crystal features where the brown numbers are the sides (six in total), red interval is the base width, yellow interval is the middle width, green interval is the middle length and the blue interval is the top length.

Table 1: Crystal feature value intervals.

Feature	Number of sides	Base width	Middle width	Middle length	Top length
Min. value	3	0.01	0.02	0.02	0.1
Max. value	8	0.04	0.07	0.1	0.2

Each feature was created within a predetermined interval seen in Table 1 to get consistent crystal shapes. Each side of the crystal was implemented as a flat surface with hard edges, thus needing flat shading to look realistic. For flat shading to work properly, multiple normals were needed for vertices that were used by different sides of the crystal. Therefore, multiple vertices with different normal direction were defined in accordance with Figure 2.

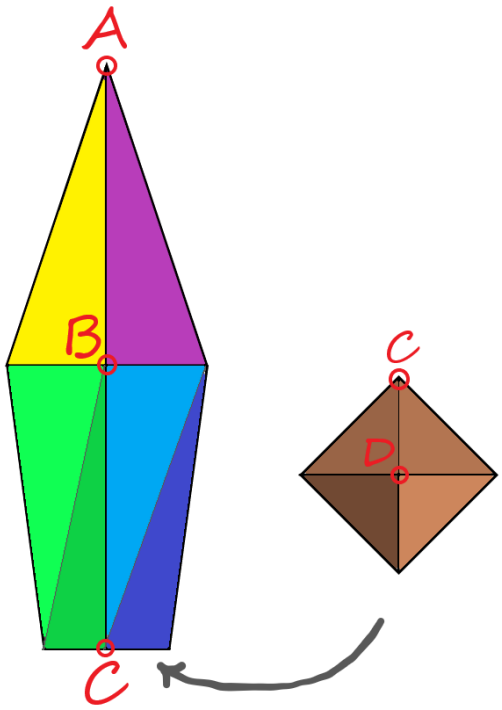

Figure 2: Four-sided crystal with visualized triangle structure. The triangle colors yellow, purple, green, blue and brown each represents a normal direction. Each point consists of as many vertices as there are adjacent normal direction, i.e. point A has the same number of vertices as there are sides on the crystal (four in this case), point B has four vertices, point C has three vertices and point D has one vertex.

Crystal generation was made in groups of one to five crystals in a chunk. Crystal angles were calculated based on the number of crystals in the chunk. Crystal chunks were only generated within a specific biome and within a defined height interval. If these criteria were met, a crystal had a chance to be generated at each vertex point of the terrain chunk with highest LOD. The chance was deterministically evaluated based on simplex noise with the vertex position as input parameter. This way, no calculations were needed for crystal chunk placements and directions as vertex positions (with a small offset to place the crystal into the ground) and vertex normals could be used. To improve performance, crystals were only rendered if the terrains chunk that contained them was shown at the highest LOD. Crystal shaders were implemented making the crystals somewhat transparent and to spawn shadows based on light direction.

3.2 Biomes

Two different methods for generating biomes were considered, as proposed by Douglas Gregory in his post on Stackexchange, zoned and emergent [2]. Zoned biomes works by creating areas of desired size and shape and then randomly assign a biome to each area. Areas could for example be created by Voronoi diagrams as in Figure 3. Emergent biomes works by generating the underlying driving factors that might affect climate. In its simplest form, and what was implemented in this project, biomes are chosen based on heat and humidity of world coordinates. By using simplex noise to generate these values, large coherent areas with biomes can be formed and controlled by changing the frequency of the noise.

Three biomes - desert, plains and crystal mountains - were implemented. Biome parameter thresholds were chosen strictly to create an interesting biome distribution and not based on real world data, see Table 2. Biome selection affected the terrain in two ways, the FBM parameters displacing the y-coordinate of terrain vertices and the color selection in the fragment shader. The desert and plains biomes had identical FBM parameters but different colors while the crystal mountains biome both had an amplified

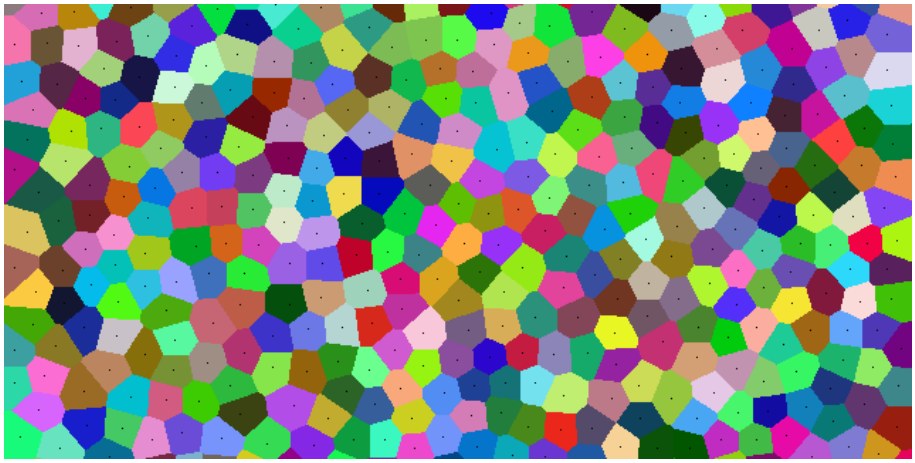

Figure 3: Voronoi diagram that can be used to divide an area into biomes. Image source: [2].

terrain with more details (more noise octaves) as well as a different color height map. All three biomes had the same minimum y-coordinate. For plains and crystal mountains, all values under the minimum y-value were moved to the minimum value, creating a flat area depicting water. The desert biome instead displaced values lower than the minimum value the same amount but in the opposite direction, which avoided flat areas and water and instead created consistent dunes.

Table 2: Simplex noise intervals used to determine terrain biomes.

Biome	Desert	Plains	Crystal mountains
Temperature	$]0.2, \infty]$	$[-0.3, 0.2]$	$[-\infty, -0.3[$
Humidity	$[-\infty, -0.2[$	$[-0.2, 0.3]$	$]0.3, \infty]$

Interpolations between biomes was done by checking the surrounding area around each vertex and counting how much of each biome that area contained. This was first implemented by calculating the biome for many points around every vertex. Since each biome calculation was relatively costly (two simplex noise calculations), this solution was too slow for practical use. Instead, a biome map was created that held all performed biome calculations for fast lookup. The biome container consisted of a sorted associative key-value map, where each key was an x-coordinate and each value was another map. The second map had the z-coordinate as its key and the biome as its value. This made it possible to find a biome value with $\mathcal{O}(\log n)$ time complexity. Biome data points were spatially evenly distributed, with eight samples per chunk. This method allowed for fewer checkups to cover the same area of the terrain. The surrounding area could then quickly be evaluated using iterators. The biome map was updated every time new terrain chunks were generated to always have a complete coverage over the terrain. Interpolation was done by adding each biome's displacement multiplied with a weight based on biomes in the surrounding area. Biome information was also added to the vertex array object passed to the shaders. This information was used to interpolate biome colors with smoothstep functions in the fragment shader.

3.3 Improved CPU Multi-Threading

To be able to generate new terrain on the CPU while still rendering frames, multi-threading was needed. In the previous project, this was implemented by generating every new terrain chunk on a new CPU thread. When no threads were available, the task was simply put in a queue and executed as soon as possible by the compiler. However, it turned out that occupying every available CPU thread, while still generating the terrain faster, could result in lag spikes. Instead, it is advisable to create a thread pool that limits which CPU threads that are used. A thread pool might hold a reference to a predetermined number of threads as well as a task queue. Whenever a task is added to the queue it gets executed by a thread as soon as one is available. Although the implementation of a thread pool was started, it was sadly too much of a challenge to be properly finished. Instead, a simpler but still effective solution

was created. The number of available threads were counted with the goal of using half of them for the program. When the camera moved outside the current chunk, a new thread was called to update the biomes map described in Section 3.2. When the biomes map had been updated, new threads generating a new terrain chunks were added while incrementing a thread counter. This was done as long as the thread counter did not exceed the number of available threads. Then, whenever the render queue updated the terrain chunks grid, the thread counter was decremented allowing for another chunk to be generated. This worked because the loop, waiting to generate terrain chunks on new threads, also was on a different thread than the main thread which allowed the program to generate frames while the process still was going on.

4 Results

This section includes screenshots of selected crystal chunks that showcases the variety of possibilities. It also includes screenshots from the program showcasing all implemented biomes and interpolations between them.

Figure 4: Example crystal chunks with one, three and five crystals.

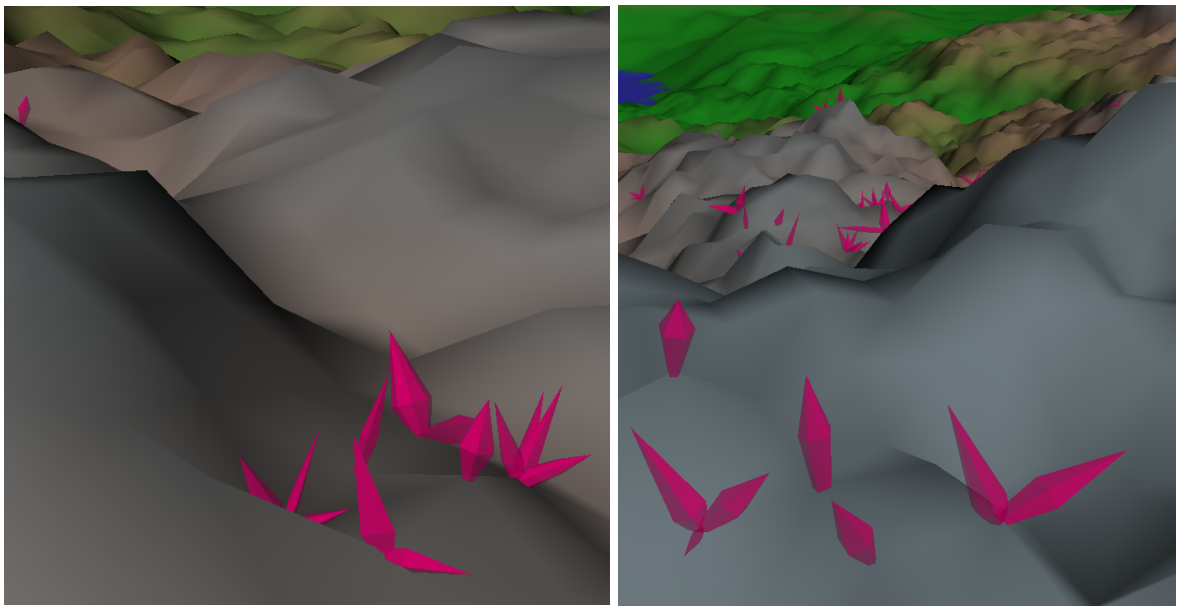

Figure 5: Crystals generated in the crystal mountains biome.

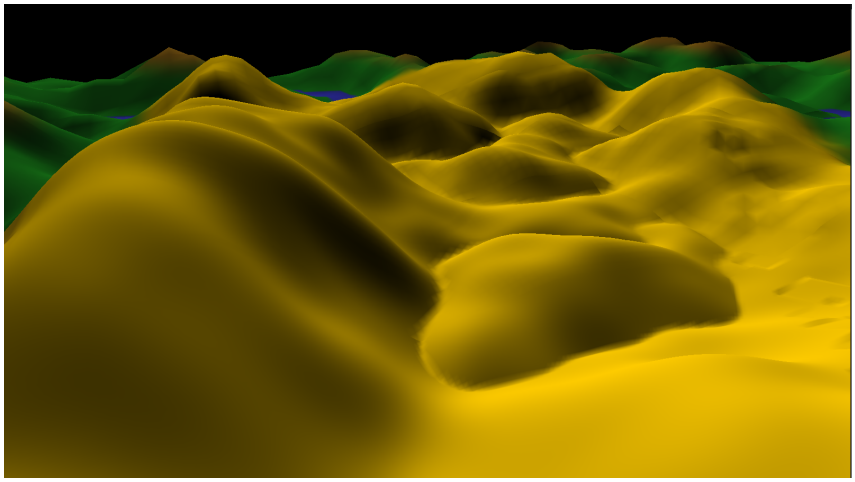

Figure 6: The desert biome.

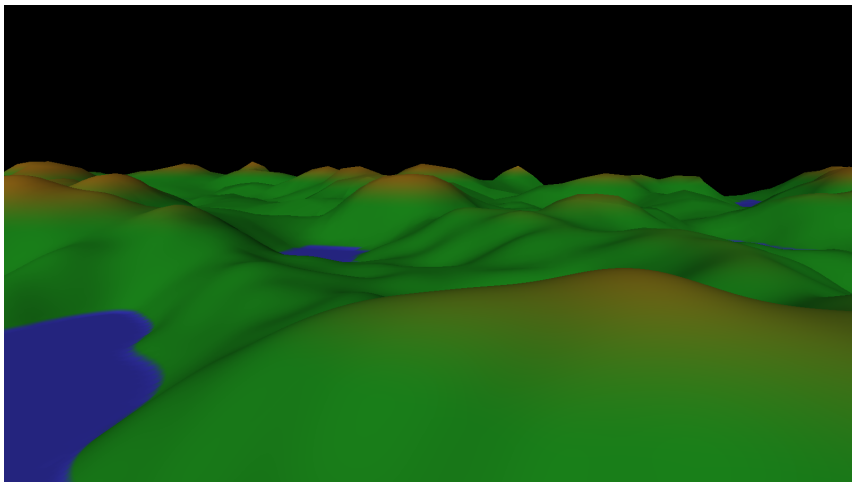

Figure 7: The plains biome.

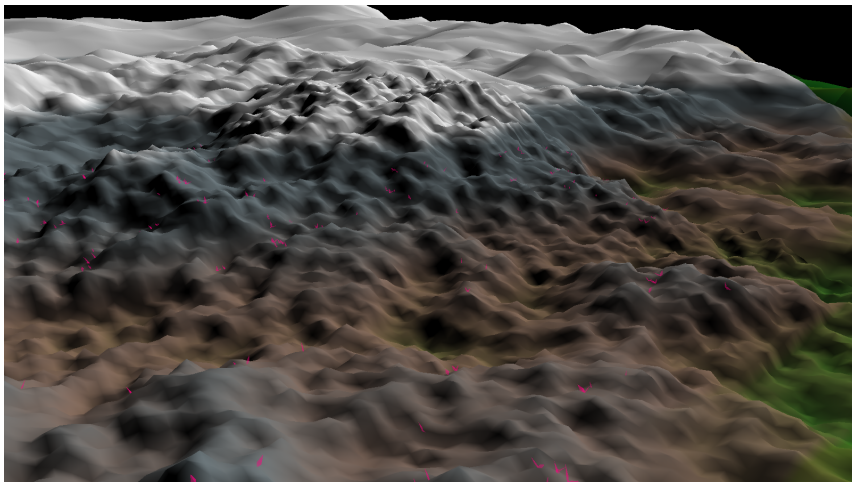

Figure 8: The crystal mountains biome.

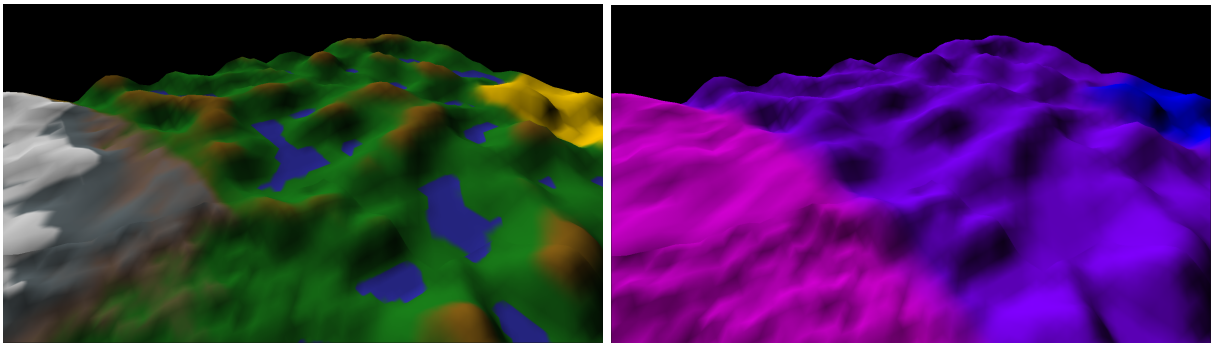

Figure 9: All three biomes with visualized interpolation. Crystal mountains in pink, plains in purple and desert in blue.

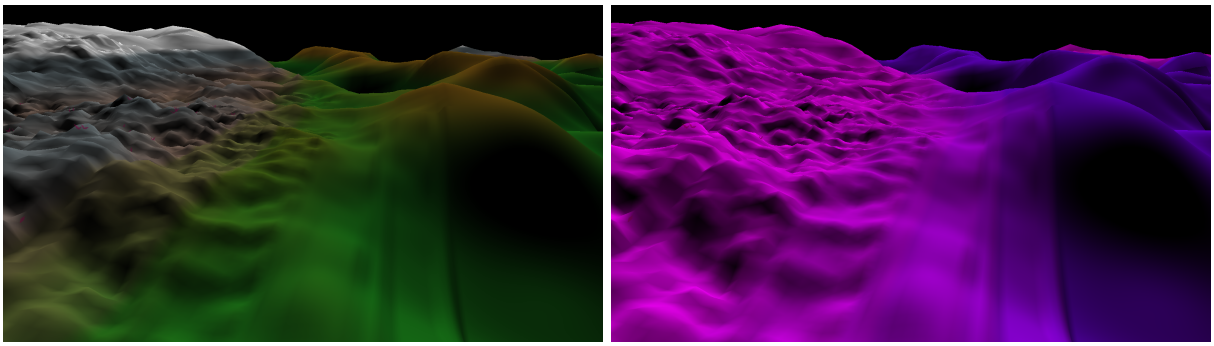

Figure 10: Crystal mountains (left) and plains (right) with visualized interpolation.

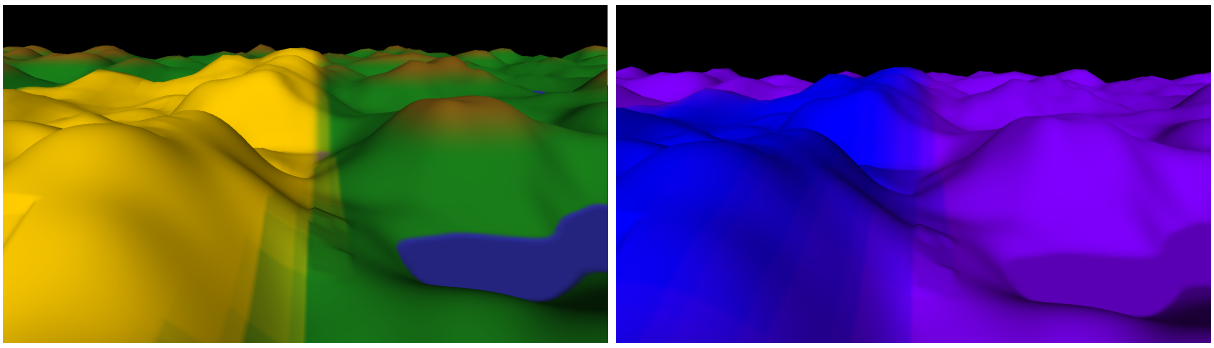

Figure 11: Desert (left) and plains (right) with visualized interpolation.

5 Discussion

The main project goals to deterministically and procedurally generate crystals, together with the extra goal to generate terrain biomes, was achieved. The goal to generate caves containing the crystals with Perlin worms turned out to be a more complex problem than anticipated. There are plenty of examples of this technique but these are all using voxel graphics with some sort of marching cubes generation. The task changes completely when vertex positions and normal calculations has to be performed.

Crystal generation was a success as shapes varied but still resembles the same crystal structure seen in Figure 4, and also was placed in the world in a consistent manner as seen in Figure 5. Improved crystal shaders were however needed as the solution to make crystals transparent with shadows did not create realistic optical properties. Specular lighting, light emittance, refraction and reflection of a environment map would have been the next implemented feature if the project timeline was extended.

Three different biomes were implemented, all with some unique twist. The positive displacement of values

lower than the minimum for the desert biome while plains and crystal mountains had water can be seen in Figure 6 and Figure 7. Crystals were generated in the brown and gray areas of the crystal mountains as seen in Figure 8.

Performance issues has become more and more apparent as more complexity has been added to the project. Limiting the number of CPU threads as discussed in Section 3.3 removed most lag spikes but it still occurs from time to time. In the previous project, there were several if-statements creating a height map used to color the terrain in the fragment shader. When switching all if-statements to multiplications with smoothstep-functions, the frame rate of the program almost doubled. Looking at this, and the implementation of the biome map data structure in Section 3.2, which improved the performance of chunk generation by many orders of magnitude in comparison to the initial implementation, hints that there are optimizations that can be done everywhere in the program. Utilizing the parallelism of the GPU more would be a good way to start.

Biomes can be visualized in real time as seen in Figure 9. Interpolation between biomes produces visible artifacts, both structural as seen in Figure 10 and in color as seen in Figure 11. The biomes were initially implemented with larger differences in their respective FBM parameters, but this created more extreme artifacts. Much time was spent trying to create better interpolation methods to be able to have more diverse biomes, but FBM parameters were made more similar in the end. Again, voxel graphics seems to be better suited to hide imperfections as frequencies shift between biomes.

References

[1] Jason Bevins. *libnoise - Example: Perlin worms*. http://libnoise.sourceforge.net/examples/worms/. [Online; accessed 06-01-2022]. 2005.

[2] Douglas Gregory. *Answer to the thread: How to randomly generate biome with perlin noise?* https://gamedev.stackexchange.com/questions/186194/how-to-randomly-generate-biome-with-perlin-noise. [Online; accessed 08-01-2022]. 2020.

[3] Stefan Gustavson. *Simplex noise demystified*. https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf. [Online; accessed 06-01-2022]. 2005.