Real time rendering of procedural terrain

TSBK03 Advanced Game Programming

Måns Aronsson - manar189
Anderas Engberg - anden561

December 29, 2021

Contents

1 Introduction

The concept of procedurally generated terrains have been around since the 70s and the game *Beneath Apple Manor* by Don D. Worth. It has since then been used in a number of titles, but was popularized in the hugely successful game Minecraft by Mojang. One revolutionary feature of Minecraft is the effectively infinite world generation made possible by the use of Perlin noise [4]. This project aims implement - like Minecraft - a program with infinite terrain generation. It also aims to implement techniques for frame rate improvement by decreasing the number of vertices drawn at each frame. Project requirement were specified as follows:

- A procedurally generated terrain with mountains and valleys.
- Working frustum culling, so that terrain outside of the camera view wont be rendered.
- Different level of detail on rendered terrain

Beside these requirements, additional goals were specified in case of completion of project requirements:

- Some form of parallelism, either on GPU or CPU, to improve the performance of the program.
- Realistic shadows by implementing advanced lighting techniques such as shadow volumes.

Version handling was done using Git and Github. The source code can be found under the *main* branch under the following link: `https://github.com/engbergandreas/ProceduralGeneratedTerrain`

1.1 Environment

The project was implemented with OpenGL 3.3 with the additional libraries GLM, GLFW and GLAD. GLM is used for mathematical vector and matrix operation. GLFW is used to create an OpenGL context, enabling the creation of a window that outputs the graphics, as well as handling user inputs. GLAD generates a loader for OpenGL functions in your system, making the use of OpenGL easier and more streamlined.

2 Theory

Although many computer graphics concepts and techniques are used in the implementation of this projects, this section focuses on the underlying theory needed to implement the three main requirements, namely fractional Brownian motion, frustum culling and level of detail.

2.1 Fractional Brownian Motion

Gradient noise is generated by (pseudo) randomizing gradient directions and interpolating between them. One such noise is the Perlin noise, first presented by Ken Perlin in 1984 and later improved with the introduction of Simplex noise in 2002 [7] [8]. One property of these noise function is the deterministic result based on input parameters. This makes it possible to generate random terrains that are deterministic, i.e. the terrain looks identical if it is deleted and generated again.

Another property of these noise functions is the possibility the change the frequency of the noise. It is this property that is utilized to generate fractional Brownian motion (FBM). FBM is created by adding multiple octaves of noise together to combine high and low frequency patterns, thus creating a more detailed and complex noise. An octave in this case refers to a multiple of a base frequency. In this project we use the built in Perlin noise function in GLM, based on the work by Stefan Gustavsson [6].

2.2 Frustum Culling

What is visible on the screen can be thought of as a volume in front of an camera. The volumes takes the shape of square pyramid and spans from the camera to defined max distances called the far plane. It can be thought of containing everything that potentially (there can be occlusion) is shown on the screen. Frustum culling is a technique of only rendering objects that are withing the view frustum volume.

2.3 Dynamic Level of Detail

Objects far away will occupy less area than close objects when projected on to the view plane. Therefore, details becomes less visible the farther away from the camera an object is located. This allows for reduction of detail (number of vertices) still producing similar results, making the rendering of the scene faster. Rendering and updating different number of vertices for objects in a scene is called dynamic "level of detail" (LOD).

The methods explored in this project are discrete, continuous and hierarchical LOD. Discrete LOD is arguably the simplest of the three, and works by creating multiple instances of each model with different resolution. The program switches between these models based on threshold values of the error returned from an error function. Continuous LOD changes the resolution of a model arbitrarily by splitting and collapsing edges using the geometry and/or tessellation shaders. Hierarchical LOD is similar to discrete LOD in the sense that models with different resolutions are generated and switched between based on an error. Further, the model is split into predefined smaller chunks that are saved in a tree structure for fast switching [2].

One problem that arises when working with LOD is discontinuity between neighbouring chunks with different resolutions. Skirts aims to solve this by adding additional vertices to each chunk, creating a vertical "skirt" around its border. This is an effective way to camouflage eventual discrepancies in the geometry. Another problem that can arise, if there are large enough differences between resolutions of a model, is called "popping". It is the noticeable geometry change when switching between resolutions.

3 Implementation

This section describes the most interesting problems that occured when developing the program and what specific design decisions were made. Although concepts are written in the general order they were implemented, each concept has continuously been worked on throughout the project, meaning parts of this section is not in chronological order.

After setup of the environment, the basic render loop, basic shaders and Mesh-class was created following the online tutorial by Joey de Vries [9]. Then, two cameras were created with controls to move around in the world. The view frustum of the first camera was visualized and the second camera was placed so that it overviews this visualization of the first camera.

3.1 Terrain Generation

The endless terrain was created as an unevenly sized grid of equally large chunks. Each chunk consisted of a mesh of vertices with equal spacing containing the terrain data. The edge vertices of each chunk contained the skirt. The x and z component of a vertex was given by the corresponding position in world coordinates while the height, i.e. the y component, was computed using a FBM with six octaves of Perlin noise. Each octave doubled the frequency and halved the amplitude of the previous frequency. All chunks were placed next to the other chunks in the grid without any gaps in between, see Figure 1.

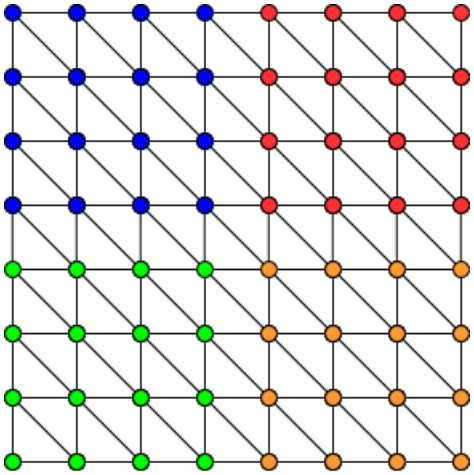

Figure 1: Simplified version displaying the underlying structure of the terrain grid, each color represent a chunk.

To improve the look of the terrain, a minimum vertex height, colors and a simple lighting model were implemented. The minimum vertex height gave the impression of water around mountains. Colors were implemented as a simple height map in the fragment shader. The lowest terrain was blue (water), then green (grass), gray (rock) and finally the top was white (snow). The lighting model consisted of the diffuse part of the Phong lighting, see Equation 1.

$$C_{out} = max(\vec{N} \cdot \vec{L}, 0) \cdot C_{in} \tag{1}$$

where C_{out} is the fragment output color, $\vec{N}$ is the normal, $\vec{L}$ is the static light direction $(1, 1, 0)$ and C_{in} is the incoming color from the vertex shader.

The generation of new - and deletion of old - terrain was triggered when the camera moved outside the middle chunk of the terrain grid. The movement direction was used to get the correct starting position of the new chunks. It was also used to correctly update the chunk pointers in the terrain grid, making

the chunk currently spanning the camera position into the middle chunk of the grid. A custom chunk destructor was implemented to correctly deallocate OpenGL buffers.

3.1.1 Computing the Normals

There are several ways to compute the normal for a given vertex, one can find or create a triangle enclosing the vertex and compute the normal from the cross product of two of its edges. Another method is the cross method where we compute the four vertices north, south, east and west of the vertex, calculating the cross product between the edges north-south and east-west. The former method gives a rough representation of the true normal and can work well in areas with low variation. The latter gives decent result, especially if the four vertices are axis aligned. Due to the predictable structure of a chunk's vertex grid, a third and more complex method was instead implemented. It considered all triangles connected to the vertex, making it the most computational heavy method but also the most accurate of the three. Every neighbouring vertex is know in the grid making it possible to calculate the normal of all six connected triangles. At the edges however, up to three vertices may not exist in the current chunk grid, but rather in a neighbor chunk. Instead of trying to find the correct vertices in the neighboring chunk, temporary fake vertices were generate at such positions as shown in Figure 2. This was done to create smooth surface normals along the edges of neighboring chunks and avoid artifacts, see Figure 3. As seen in the left image, there are visible line artifacts between neighboring chunks whereas in the right image, fake vertices are created and no artifacts can be seen. The algorithm used in the project computes an average weighted sum of all connected triangle normals where the area of the triangle acts as its weight. The result is then normalized and uploaded to the vertex data.

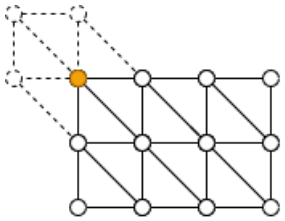

Figure 2: Generate fake vertices (dotted) to compute correct normal (yellow vertex)

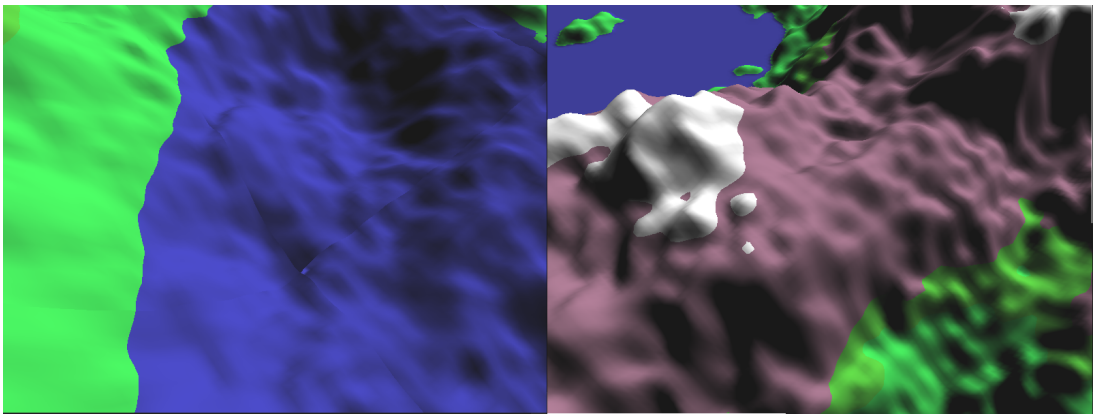

Figure 3: Left image: line artifacts between neighboring chunks. Righht image: smooth normals between chunks.

3.2 Axis Aligned Bounding Box Frustum Culling

An axis aligned bounding box (AABB) encloses a terrain chunk, creating a minimum rectangular box. It iwas created by taking the maximum and minimum points in all directions of a chunk. The x and z components of the bounding box was given by the position and size of the grid. And the y component

was given by the maximum and minimum height of a chunk. The bounding box was used to determine if the chunk object is visible by the camera, which then flagged the object for rendering. It simplified frustum culling, as the chunk can be described by eight vertices in the bounding box instead of thousands of vertices in the mesh. The bounding boxes are visualized in green lines shown in Figure 4.

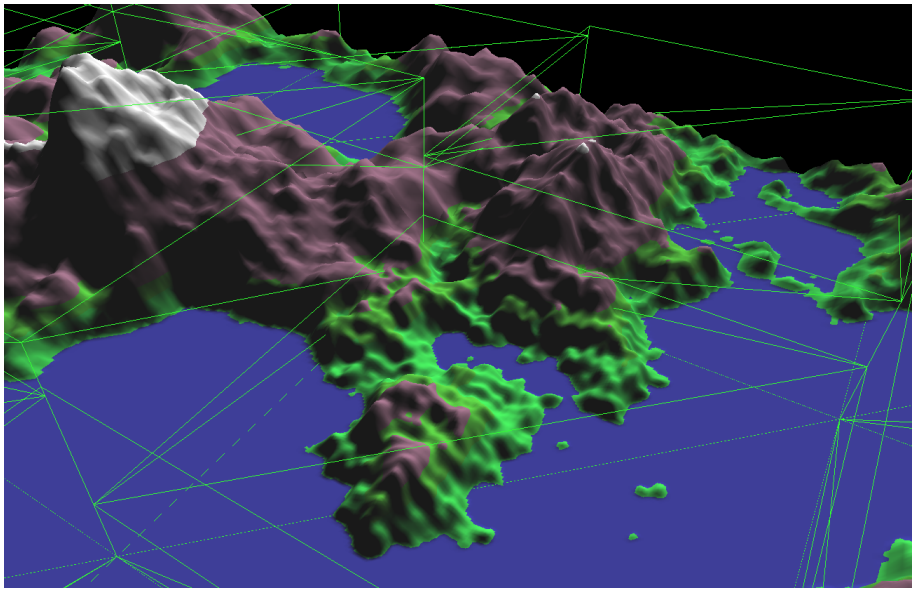

Figure 4: Bounding boxes are visualized as green lines, given by the min and max extreme vertices of a chunk.

3.2.1 Defining the Frustum

OpenGL requires all vertices that are visible to be in the normalized device coordinates (NDC) in the range $[-1, 1]$. The perspective projection used created a skewed camera frustum that maped world coordinates inside the frustum, to the projected view coordinates in the range $[-1, 1]$. However, culling needed to be performed on objects before they are sent to the shader, this meant that operations needed to be done in the world coordinate system. The first step was to find the camera frustum in world coordinates, this can be done by going from NDC to world via the inverse view-projection matrix.

Eight frustum corner points $f_i = (x, y, z, w)^t$ were defined as $f_i = (\pm 1, \pm 1, \pm 1, 1)^t$ where the near plane had $z = -1$ and the far plane $z = 1$. The inverse view perspective matrix was given by $(VP)^{-1}$. The world frustum corner points were given by the equation

$$f_{world,i} = f_i \cdot (VP)^{-1}/w \tag{2}$$

where w is the homogeneous coordinate and the division was done component-wise. The resulting frustum was drawn and visualized, see Figure 5. The frustum was made up of the six planes top, bottom, left, right, near and far. The frustum plane π_i was defined by a point $P_{0,i}$ and a vector perpendicular to the plane as

$$\pi_i = \overrightarrow{P_{0,i}P_i} \cdot \vec{n_i} \tag{3}$$

where P_i is any point in the plane, $\vec{n_i}$ is the normal pointing inwards in the frustum and $P_{0,i}$ is taken as one of the eight corner points f_{world}.

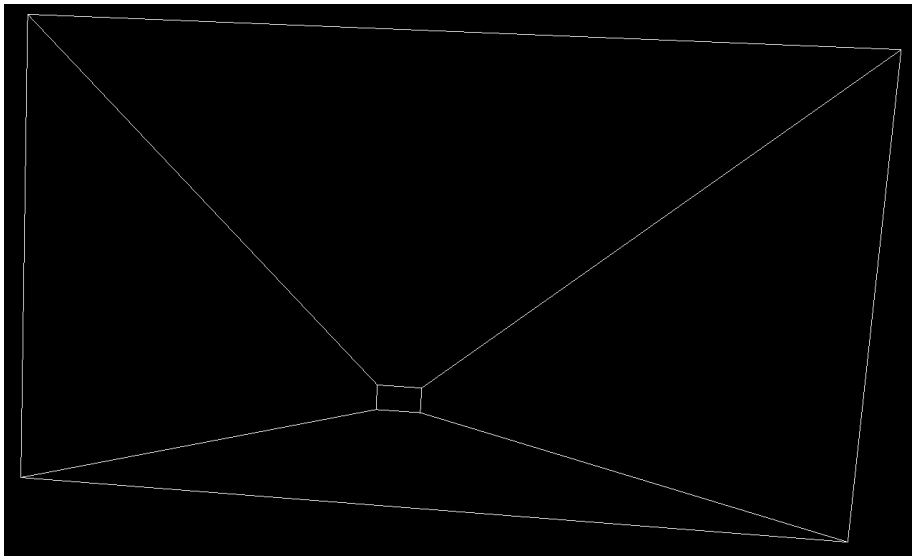

Figure 5: Camera frustum visualization

3.2.2 Culling

Intersection tests were done in order to determine if an object should be culled. An variation of the basic intersection method proposed by Assarsson and Möller [1] was implemented, without the extra optimization steps. The intersection tested the bounding box against each camera plane to find if the object was inside or outside the plane.

Only two of the eight vertices on the AABB needed to be tested. These vertices were called *p-vertex* and *n-vertex*. The *p-vertex* was given by the point furthest in the positive direction i.e. the direction of the normal $\vec{n_i}$. The *n-vertex* was given by the point furthest in the negative direction; an example is illustrated in Figure 6. Because both the bounding box and frustum planes π_i were given in world coordinates, the *p-* and *n-vertex* could efficiently be retrieve using the signs of the x-, y- and z-components of the plane normal in a look-up table, see Table 1 [5]. For every frustum plane, the vertices using equation (3) were evaluated with P_i as input parameter. The $n-vertex$ was first evaluated and if the result was positive, the bounding box was reported as outside. Otherwise, the $p-vertex$ was evaluated and if the result was negative, the bounding box was completely inside. Otherwise, it was intersecting the plane. All chunks that were reported as inside for all six planes were drawn. If the AABB was intersecting the chunk it was still fully drawn. However, optionally one can subdivide the chunk and test each part individually to cull more of the terrain.

Table 1: Look up table to determine *p-* and *n-vertex*

n_x	n_y	n_z	*p-vertex*	*n-vertex*
+	+	+	$\{x_{max}, y_{max}, z_{max}\}$	$\{x_{min}, y_{min}, z_{min}\}$
+	+	-	$\{x_{max}, y_{max}, z_{min}\}$	$\{x_{min}, y_{min}, z_{max}\}$
+	-	+	$\{x_{max}, y_{min}, z_{max}\}$	$\{x_{min}, y_{max}, z_{min}\}$
+	-	-	$\{x_{max}, y_{min}, z_{min}\}$	$\{x_{min}, y_{max}, z_{max}\}$
-	+	+	$\{x_{min}, y_{max}, z_{max}\}$	$\{x_{max}, y_{min}, z_{min}\}$
-	+	-	$\{x_{min}, y_{max}, z_{min}\}$	$\{x_{max}, y_{min}, z_{max}\}$
-	-	+	$\{x_{min}, y_{min}, z_{max}\}$	$\{x_{max}, y_{max}, z_{min}\}$
-	-	-	$\{x_{min}, y_{min}, z_{min}\}$	$\{x_{max}, y_{max}, z_{max}\}$

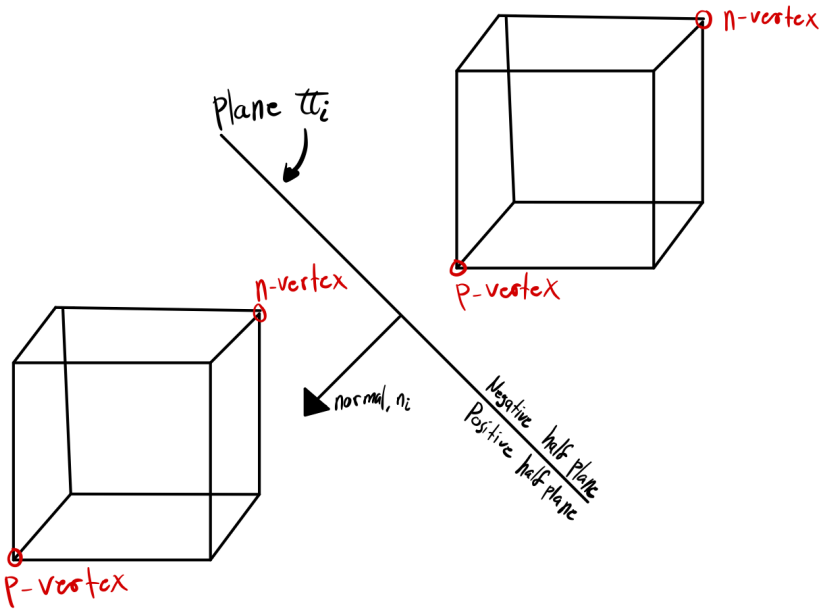

Figure 6: $p-vertex$ is given by the furthest AABB point in the positive direction and $n-vertex$ is given by the point furthest in the negative direction

3.3 Discrete Level of Detail

The implemented discrete LOD created several copies of a chunk with fewer vertices per level. Every level used half the amount of vertices from the previous level and vertices are "reused" between levels. Figure 7 illustrates the LOD-structure in one dimension for three levels. Level one visited every vertex in the grid while level two and three visited every second and every fourth vertex respectively. The number of vertices in the grid was $n_{lod} = (n-1)/lod + 1$ where n is the base number of vertices in the grid and $lod = 1, 2, ..., k$ is the number of levels a chunk had. For k levels, n must be chosen so that it fulfils two criterion; $n-1$ must be divisible by 2^{k-1} and chosen such that $n = 2^{k-1} \cdot b + 1$ with factor $b = [1, \infty[$ determining the number of columns in the lowest level.

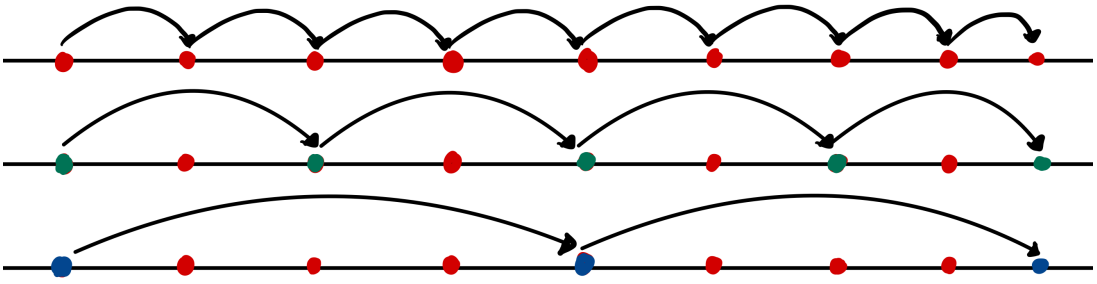

Figure 7: Discrete level of detail showing three levels in one dimension.

The lower LOD chunks were stored as a pointer inside the chunk class, which makes it easy and fast to switch between different LODs. To chose which LOD-chunk to render, the distance between the chunk and camera were used. To illustrate LOD in action, a color was given for each LOD from green (most detailed) to red (least detailed), as shown in Figure 8.

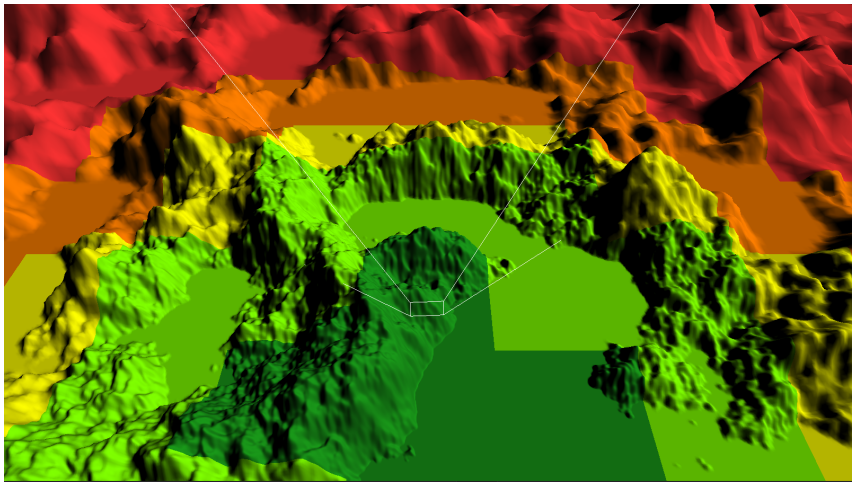

Figure 8: LOD visualized by color, green most detailed to red least detailed with respect to the distance to the camera.

3.4 CPU Multi-threading

Since vertex positions of new chunks were generated on the CPU, every new chunk would freeze up the program until it was completed. To circumvent this problem, CPU multi-threading was implemented. Generating every chunk on a new thread makes it faster, but joining threads at the end of the render loop would still make the program freeze up. Therefore, chunks were generated on detached threads. Detached threads operates independently of the main thread and are deleted as soon as they are finished with their assigned task. After a new chunk had been generated on a detached thread, it was added to the render queue. That way, the main thread was able to update the chunk grid with the newly generated chunk as soon as the render queue was not empty. To correctly update the chunk grid, the movement that triggered new chunks to be generated was also added to the render queue with each new chunk.

This implementation had three major problems that had to be solved. Firstly, generating chunks on different threads made OpenGL incorrectly reference the vertex array object (VAO), vertex buffer object (VBO) and element buffer object (EBO). This was solved by only generating the geometry information on a different thread and then creating all references on the main thread with a ”bake” function. Secondly, quick successive movements outside of chunks, e.g. crossing a chunk corner or leaving and quickly returning to a chunk, could trigger chunks to be generated in the wrong order, which made the chunk grid update incorrectly. This was solved by adding a movement queue and only starting to generate chunks of a movement when all chunks from the previous movement had finished. Lastly, accessing the movement- and render queues while new items were added to them would occasionally result in a reference error. This was solved by adding a mutex to critical parts of the code. A mutex is used to protect shared data from being accessed at the same time.

4 Results

The project resulted in a program with endless terrain generation. Object culling was effectively implemented which increases the frame rate of up to 60% and LOD which decrease rendering time by up to 5.5 times. We built a 17 x 17 grid of chunks, each with roughly 26,000 vertices and 5 levels for the LOD. The program was tested on a laptop with an Intel i5 8th gen processor with integrated graphics card Intel UHD 620 and 8 GB of ram. Several settings were tested, with both LOD and frustum culling turned on roughly 600-1100 frames per seconds (FPS) was achieved. With LOD on and culling off we attain between 200-400 FPS. The opposite settings (LOD off, culling on) get around 150-250 FPS and with both settings off barely 30 FPS was kept, approximately 95% slower than with both settings on.

The resulting culling can be seen in Figure 9 where chunks are separated to highlight the bounding box surrounding the terrain chunks. Notice how only chunks inside the frustum are rendered, as chunks outside are ignored in the rendering loop. Figure 10 and Figure 11 shows the resulting endless terrain and level of detail visualized in color and drawn in wire-frame mode from the "ground camera" respectively.

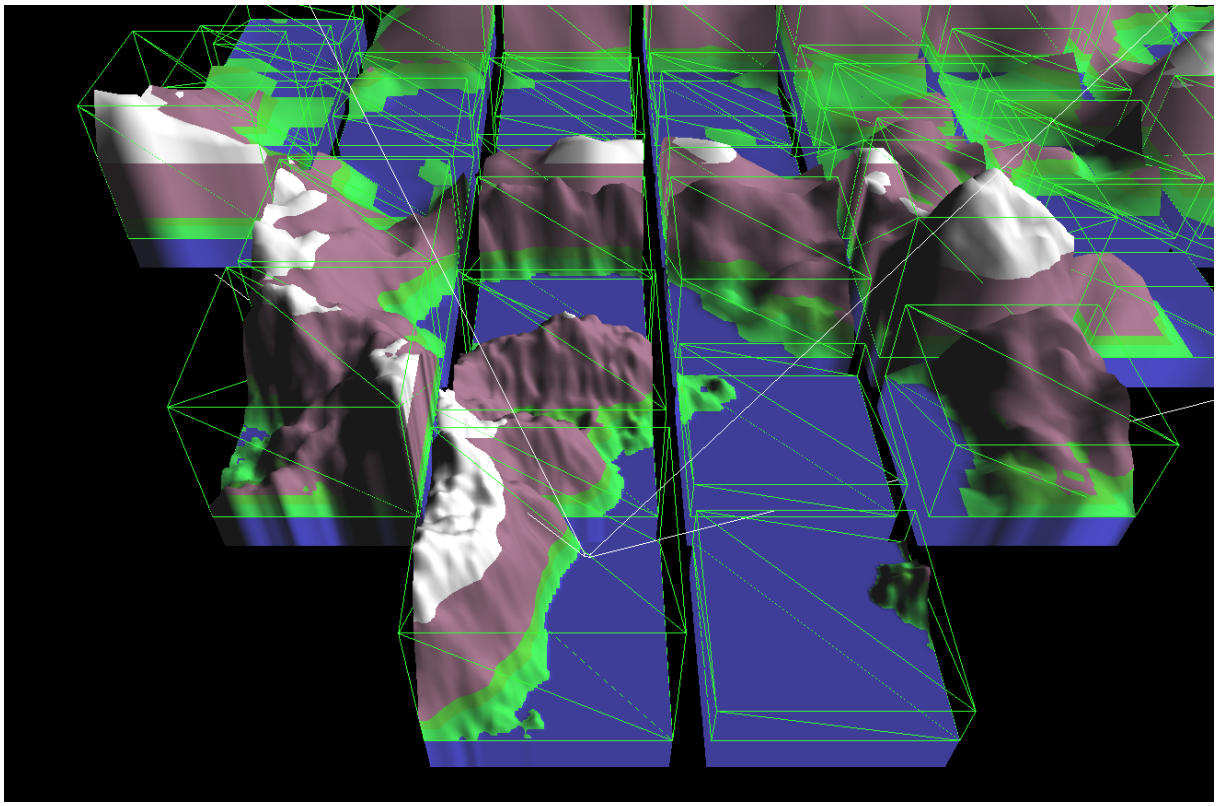

Figure 9: Result from culling terrain, chunks outside the camera frustum are not drawn

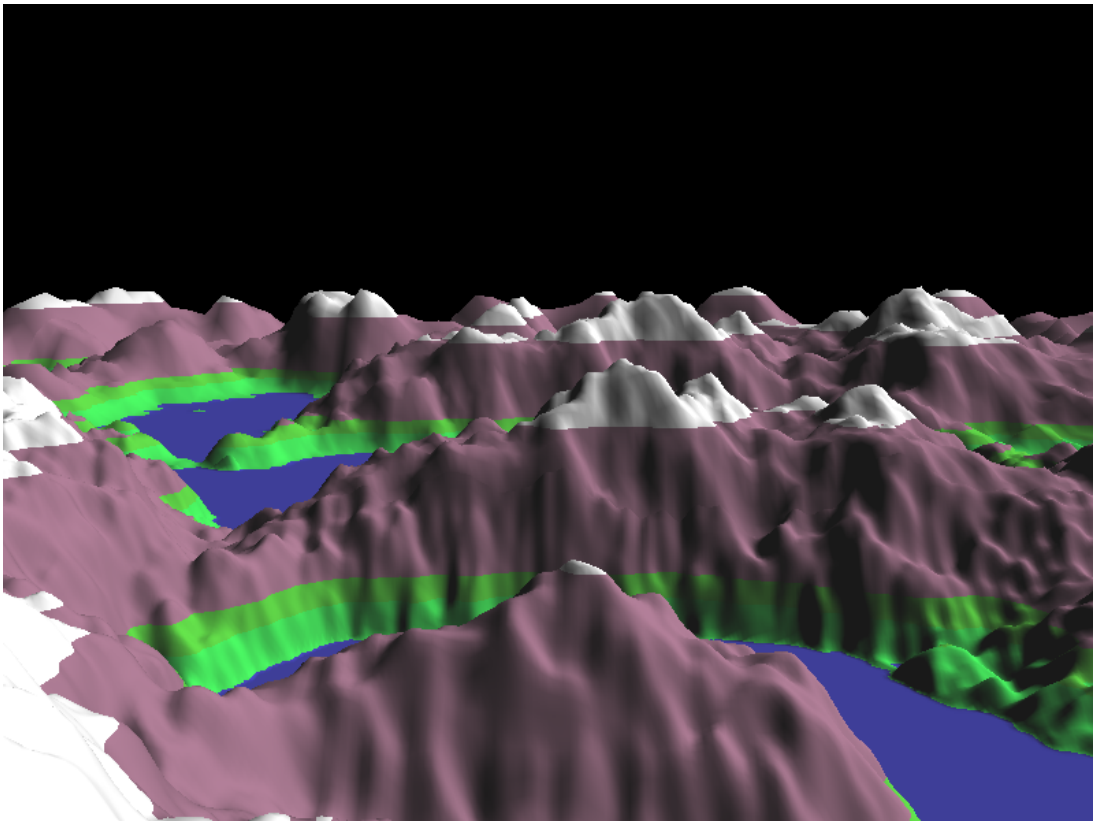

Figure 10: Endless terrain viewed from camera one

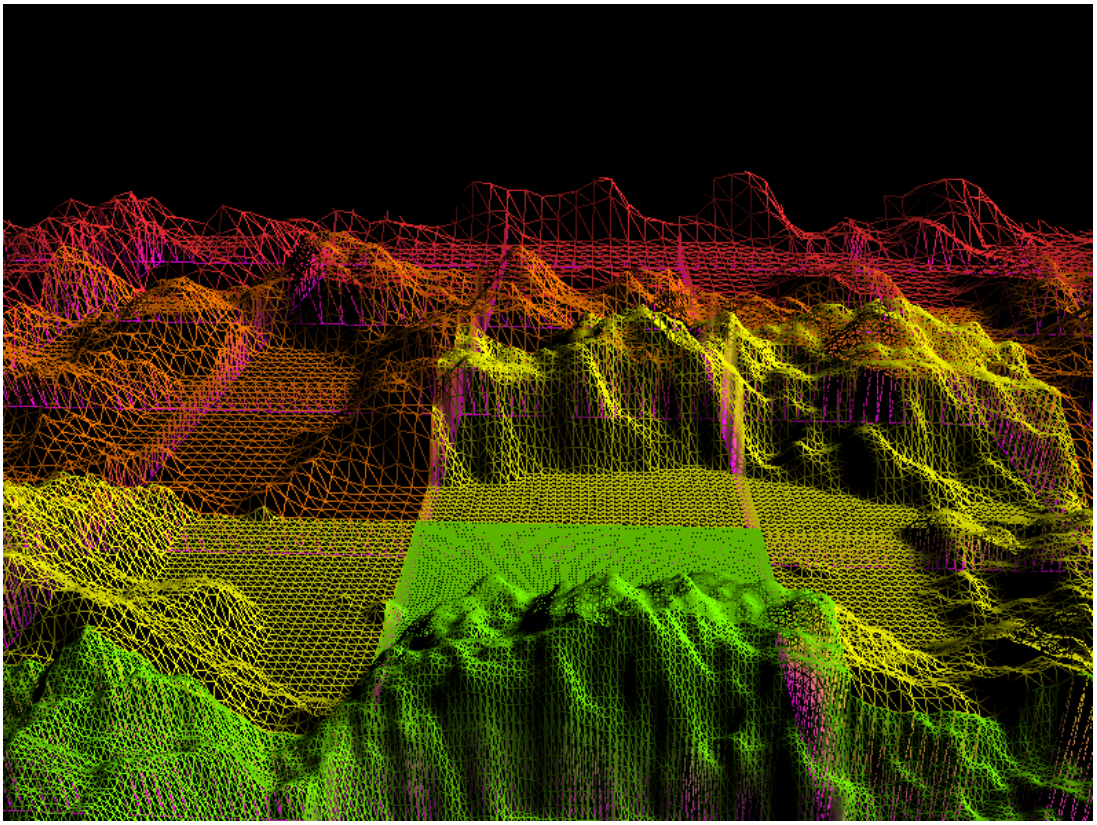

Figure 11: Result showing lower LOD further away from the camera

5 Discussion

This section handles some choices made in the project and alternative methods and solutions. Furthermore, some of the problems with the chosen methods are highlighted. From the test results it can be seen how important it is to keep the number of vertices low in order to attain a better frame rate. Both the implementation of level of detail and frustum culling greatly increase the frame rate.

Terrain generation on GPU An alternative solution to generating terrain is to make use of the GPU and shaders. A single flat grid of fixed size could be generated. The height would then be changed in the vertex shader using the same type of FBM perlin noise as before. Then, new chunks could be instantiated from the same original mesh grid. The benefit of this approach is that we utilise the parallel processing on the GPU when computing the height value instead of a single thread on the CPU. Moreover, instantiating greatly reduces the CPU overhead when generating new chunks as the mesh model can be saved in cache memory. The major drawback however, is the normal calculations, as it is no longer possible to use a form of lookup table to quickly compute the normals. Instead, simpler methods of computing the normal, such as taking small steps in the noise around the vertex to approximate the normal, could be utilized.

Bounding volume choice The rectangular bounding box was chosen, since it align with the terrain grid very well. Another bounding volume that can be used is the bounding sphere. Frustum culling with bounding spheres includes only one check, that is the signed distance between the sphere radius and the camera planes it is therefore potentially faster than using the bounding box. However, one problem that was considered using bounding spheres over bounding boxes was for terrain chunks that are elongated along one axis, for example in Figure 9 there is a chunk that contain mostly water. Applying a bounding sphere to this chunk would result in a sphere that is mostly empty in the bottom and top half. This would lead to the chunk being rendered even if the player looks far above or below the chunk. It would therefore be interesting to see which of the two bounding volume choices is the most efficient for real-time rendering. Spheres being faster to compute but has a larger false positive rate while bounding boxes are more accurate but requires additional computations.

Popping between LOD A problem with discrete level of detail is the popping effect that occurs when the renderer switches between two different levels. This is caused by the fact that lower LODs are missing half the information from previous ones as illustrated in Figure 12. The effect of popping can be reduced somewhat by increasing the number of levels or by decreasing the difference between levels both of which are more memory consuming. Another approach is to use a weighted interpolation between levels [2]. In theory, this should be more efficient and less memory consuming than the former methods.

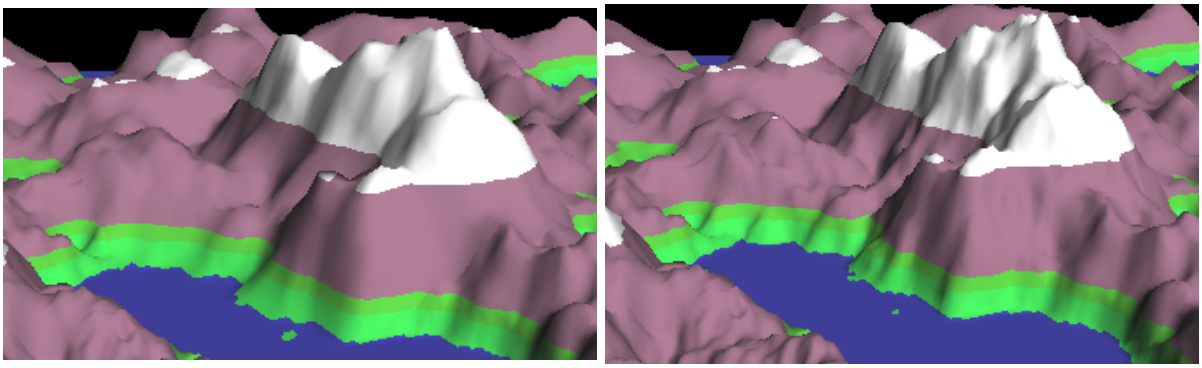

Figure 12: Popping between different LOD

Shading Another problem that occurs is the different shading neighboring chunks with two different LOD receive. The effect can be seen in Figure 13. This is caused by more detailed normals in the higher level than the other. No solution for this was found to this problem other than to increase the distance

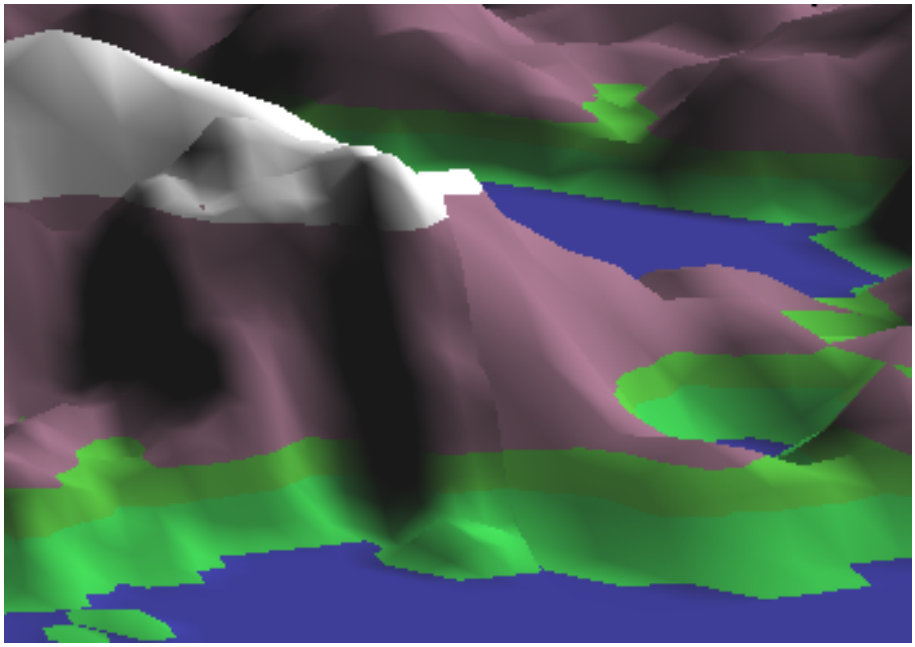

Figure 13: Inaccurate shading between neighboring chunks of different LOD.

a level is rendered at which effectively puts the problem further away from the camera and is therefore less likely to be seen.

Threaded Level of Detail In the final version of the program, every level of detail were generated in the chunk's constructor. A better solution is to generate each resolution individually, lowest to highest resolution, and making them accessible to render as they finish. This functionality was implemented in the Git branch "lod", but not merged with the main branch due to unsolved bugs. It is however possible to find in the Github repository [3].

References

[1] Ulf Assarsson and Tomas Moller. "Optimized view frustum culling algorithms for bounding boxes". In: *Journal of graphics tools* 5.1 (2000), pp. 9–22.

[2] K. Bladin and E. Broberg. "Design and Implementation of an Out-of-Core Globe Rendering System Using Multiple Map Services". Dissertation. Linköping University, 2016. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-137671.

[3] Andreas Engberg and Måns Aronsson. *ProceduralGeneratedTerrain*. Version 1.0.0. Dec. 2021. URL: https://github.com/engbergandreas/ProceduralGeneratedTerrain.

[4] Jonas Freiknecht and Wolfgang Effelsberg. "A Survey on the Procedural Generation of Virtual Worlds". In: *Multimodal Technologies and Interaction* 1.4 (2017). ISSN: 2414-4088. DOI: 10.3390/mti1040027. URL: https://www.mdpi.com/2414-4088/1/4/27.

[5] Ned Greene. "Detecting Intersection of a Rectangular Solid and a Convex". In: *Graphics Gems* (1994), p. 74.

[6] Stefan Gustavson. *Simplex noise demystified*. https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf. [Online; accessed 23-12-2021]. 2005.

[7] K Perlin and EM Hoffert. "Acm siggraph". In: *Course in Advanced Image Synthesis* (1984).

[8] Ken Perlin. "Improving Noise". In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. San Antonio, Texas: Association for Computing Machinery, 2002, pp. 681–682. ISBN: 1581135211. DOI: 10.1145/566570.566636. URL: https://doi-org.e.bibl.liu.se/10.1145/566570.566636.

[9] Joey de Vries. *Learn OpenGL*. https://learnopengl.com/. [Online; accessed 27-12-2021]. 2014.