Voxel engine for generation of procedural terrains

Fredrik Johansson
TNM084 Procedural Methods for Images
Linköping University
frejo851@student.liu.se

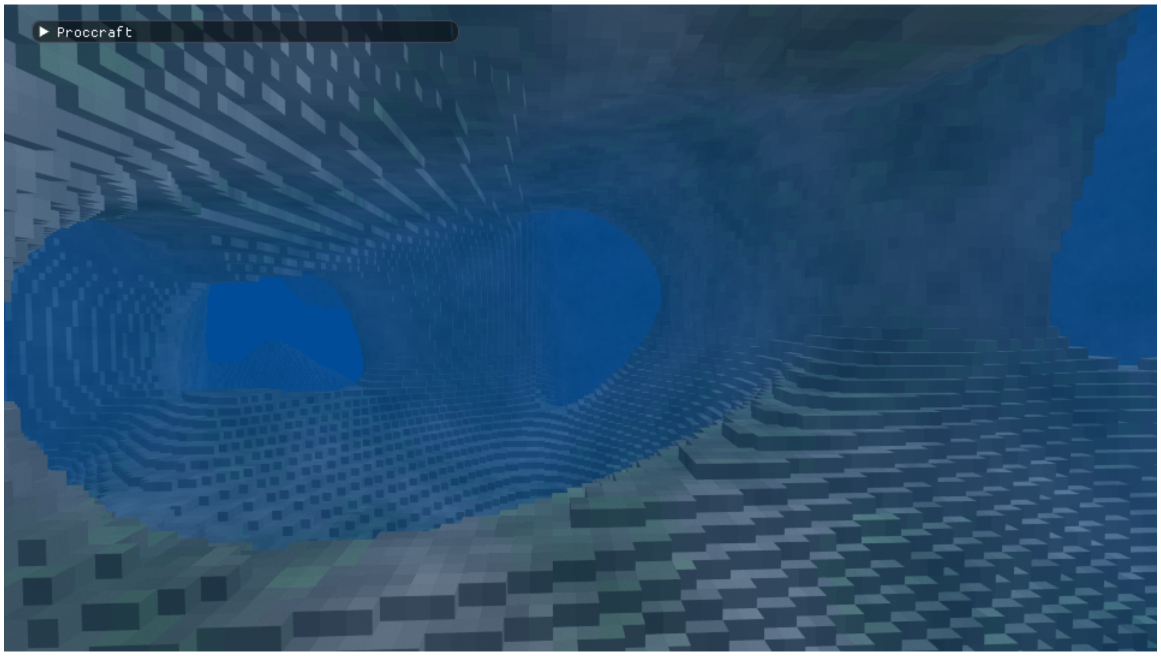

Figure 1: A cave system produced by the voxel engine.

ABSTRACT

This report describes the implementation of a voxel terrain engine from scratch in C++ with the graphics API OpenGL and the shading language GLSL. The engine uses a combination of marching cubes and simplex noise to generate the voxels. In the result, images of some generated terrains are shown as well as a performance analysis of the engine.

1 INTRODUCTION

In video games and simulations, terrains tend to expand over a large surface area. Modelling these terrains can be a tedious task and result in a biased terrain. However, by using procedural methods these terrains can be generated and expand to infinity.

An engine for this purpose were implemented during this project. The engine generates a highly customizable voxel terrain using noise functions.

2 METHOD

The engine was implemented in C++ with the graphics API OpenGL, the shader language GLSL, the OpenGL Extension Wrangler (GLEW), the library GLFW and the mathematics library GLM. GLM was used to generate simplex noise which is based on the paper written by Stefan Gustavson [3].

2.1 Voxel

A voxel is a position in a three-dimensional space. The voxels are used as building blocks for the terrain. Each voxel has a side length of one and is represented by 24 vertices with six values each. The first three values contain the vertex position and the remaining values contain the normal for that vertex. This explains why there are 24 vertices instead of just eight since there are three normals connected to each corner point. By using a combination of three vertices a face can be drawn.

To draw a voxel, its vertices are sent to the GPU via OpenGL in a Vertex Array Object (VAO). The VAO is designed to store the information for a complete rendered object. Therefore, the VAO can contain multiple Vertex Buffer Objects (VBO), in this implementation it contains two. The first VBO contains all the necessary corner points of the voxel. The second VBO contains the normals corresponding to each corner point. Lastly an Index Buffer Object (IBO) is created where the combinations of the vertices are stored to create the voxel faces.

2.2 Noise

3D simplex noise were used to generate the terrain for cave systems. If the noise value is larger than zero, voxels should be drawn.

For mountainlike terrain, multiple 2D simplex noise functions were used to create fractal noise as a heightmap in the y-direction. Fractal noise makes large terrains look more natural and it makes the artifacts less apparent [2].

Multiple simplex noise functions were also used to add color and texture to the voxels. High frequent noise with low amplitude were used to add some texture to the voxels when looking closely. There are also noise functions to add spots of moss to create the illusion of a humid climate. Lastly, there are noise functions to add variations in the color for the stone and the moss.

To achieve the look of each voxel having its own uniform color and shade, the noise is rounded to the largest possible integer value which is less than or equal to the noise value. This was possible since the voxels side length is equal to one as mentioned in Section 2.1.

2.3 Chunk

By simply executing a draw call for every voxel a terrain could be generated. However, for every draw call data is transferred between the CPU and GPU. This is a relative slow process for the computer and leads to poor performance for large terrains. To solve this the number of draw calls had to be reduced. Therefore, chunks were introduced.

Each Chunk consists of 16x16x16 voxels and are batched into a single draw call. This results in a reduced number of draw calls, but a problem remained. Each voxel would render its faces within the chunks which is unnecessary and slows down the engine. Figure 2 illustrates what a chunk looks like without using any noise to displace the voxels within the chunk.

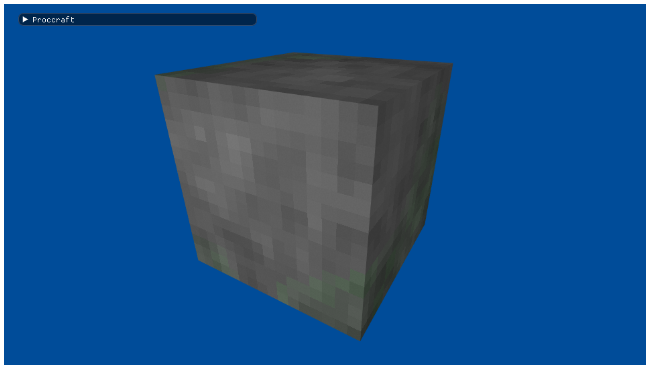

Figure 2: A chunk consisting of 16x16x16 voxels.

2.4 Marching cubes

To solve the problem with the unnecessary faces being drawn, an algorithm called Marching cubes [4] were implemented. The first step of this algorithm is to determine which voxels are visible in each chunk. Therefore, the algorithm iterates through each voxel in each chunk and evaluates the noise value. If the value is above zero the algorithm checks if the current voxels, six nearest neighbours that have already been traversed are of air or stone. If a neighbouring voxel is of air the current voxels vertices corresponding to the direction of the neighbour are inserted into an array.

There is a problem with this approach, the voxel faces at the edges of each chunk becomes invisible. This is solved by iteration through the closest surrounding voxels and thereby also evaluating the surrounding chunks edges.

Lastly, the gathered vertices are sent to the GPU in the same manner as described in Section 2.1. With the help of this algorithm only the necessary vertices are sent to the GPU, resulting in less memory usage per chunk. For example, the size of the chunk illustrated in Figure 2 would decrease from 98304 vertices to 6144 vertices. Figure 3 is an outside view of the optimized chunks in a cave system. Figure 6 illustrates the inside of this particular cave system.

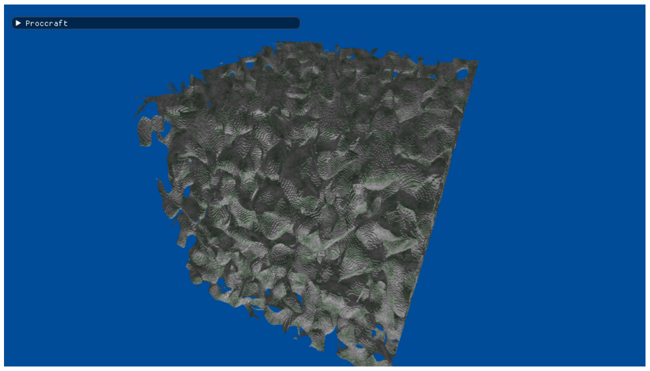

Figure 3: Outside view of a cave system consisting of 1331 chunks, each generated with simplex noise and the marching cubes algorithm.

2.5 Infinite rendering

To allow for an immersive experience while at the same time decreasing the loading time of the engine, infinite rendering were implemented. For every frame the engine evaluates the camera position. If the camera moves between the edge of two chunks, new chunks corresponding to the movement direction are added to a render queue. If the camera position has already passed a specific edge no new chunks are added. In practice, in the default state of the engine this means that for every 16th voxel, new chunks are added to the queue. Every new frame a chunk from the rendering queue is transferred from the queue to the renderer. This makes the terrain continuously generate as the camera is moving until the allocated memory of the renderer is full.

2.6 Light and atmosphere

The engine uses simple directional light and diffuse and specular reflection properties for the voxels.

To add some atmosphere, exponential fog were implemented based on the instructions written by Sergiu Craitoiu [1].

3 RESULTS

The type of terrain the engine generates is highly customizable. Therefore, the result is separated into two different versions. The first version uses 2D noise to generate a heightmap for mountainlike terrains. This version of the engine is illustrated in Figure 4 and 5. The mountainlike terrains have been generated using fractal noise.

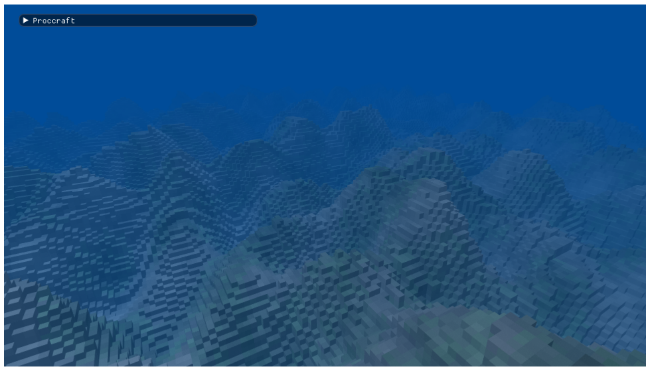

Figure 4: Assigning each voxel a y-value from Fractal noise.

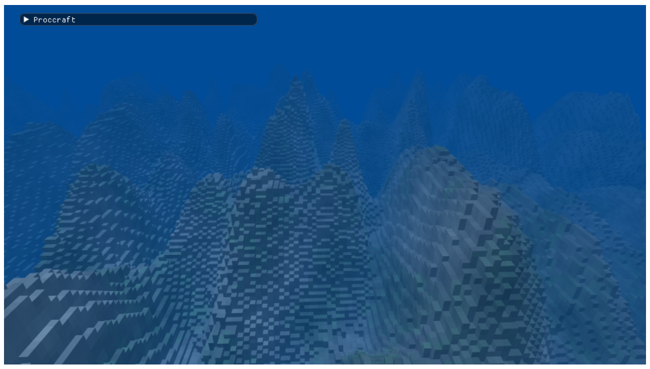

Figure 5: Fractal noise with higher frequency in the x-direction to determine the height of each voxel.

The second version uses 3D noise to generate cave systems. Figure 6, 7 and 8 are some cave systems found while playing around with the noise frequency and amplitude in the engine.

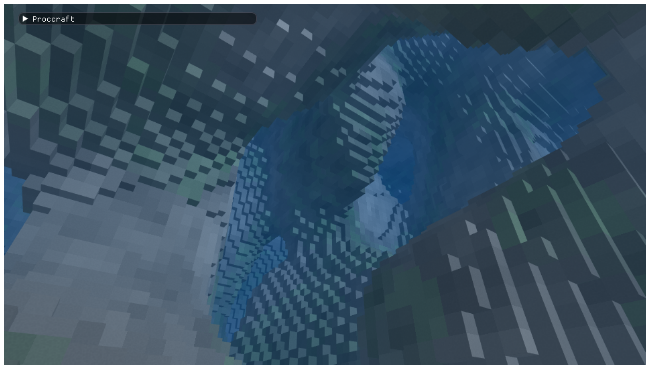

Figure 6: Cave generated with simplex noise using a lower frequency in the y-direction.

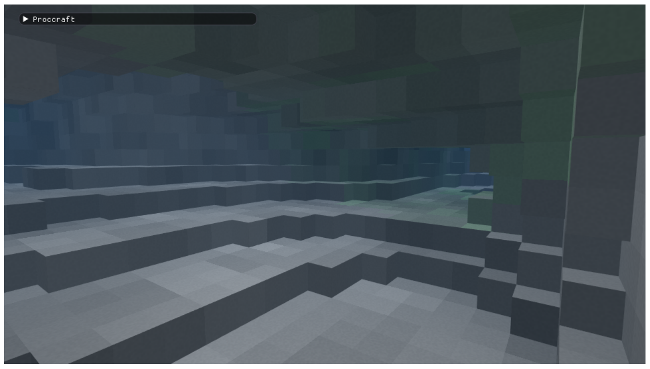

Figure 7: Cave generated with simplex noise using high frequencies.

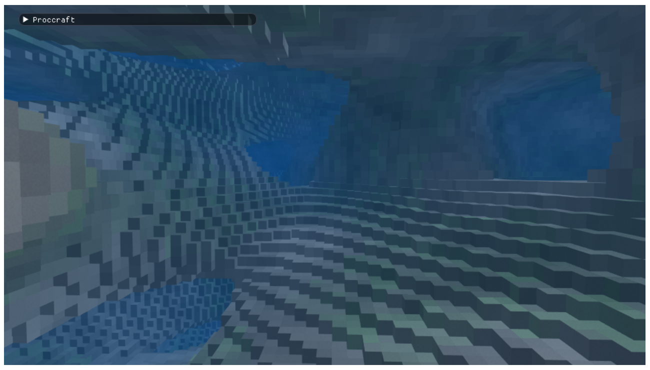

Figure 8: Cave generated with simplex noise using lower frequencies in the xz-direction.

3.1 Performance analysis

The total file size of the engine executable with the including shader files is 640 kB. Table 1 and 2 shows the performance of the engine using different hardware. The tests were executed using the same noise frequencies and having the camera in a static direction. Information about the computer specifications can be seen in the table's caption.

Table 1: Intel Core i7-4790K CPU @ 4.00GHz and AMD Radeon R9 200 Series

num. of chunks	avg. time per frame	avg. frame per second
125	1.26 ms	790
343	1.96 ms	508
729	3.95 ms	250
1331	8.6 ms	116

4 CONCLUSIONS AND FUTURE WORK

In conclusion the engine does what it was implemented and designed for. It is highly customizable and the performance is acceptable for large terrains. What follows are some suggested improvements that could be implemented.

Table 2: Intel Core i7-6500U CPU @ 2.50GHz and Intel HD Graphics 520

num. of chunks	avg. time per frame	avg. frame per second
125	8.9 ms	112
343	11.9 ms	84
729	18.6 ms	54
1331	29.2 ms	34

As mentioned in Section 2.1 each voxel contains 24 vertices, which is unnecessary. This could be reduced by computing the normal with the cross product between two vertices. This would lead to fewer values per vertex and result in a total of eight vertices per voxel.

Currently, once the user updates the noise parameters, it affects the not yet rendered chunks. This leads to sharp edges between the newly generated chunks, and the previously generated chunks. A solution would be to interpolate between the chunks or by recalculating every active chunk.

To increase the performance further frustum culling could be implemented. Frustum culling would probably increase the performance of the engine significantly.

In the current state of the engine no chunks are removed from the renderer. This results in the engine performance dropping linearly to the amount of active chunks. To solve this an upgrade to the rendering queue and rendering array would have to be implemented.

ACKNOWLEDGMENTS

Thanks to Stefan Gustavson for providing me with GLSL code for simplex noise.

REFERENCES

[1] Sergiu Craitoiu. Create a fog shader. *in2gpu*, 07 2014.

[2] Darwyn Peachey Ken Perlin David S. Ebert, F. Kenton Musgrave and Steven Worley. *Texturing & Modeling, Third Edition: A Procedural Approach.* Morgan Kaufmann Publishers Inc., 340 Pine Street, Sixth Floor, San Francisco, CA, United States, 2002.

[3] Stefan Gustavson. Simplex noise demystified. *Linköping University*, 05 2015.

[4] William Lorensen and Harvey Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21:163–, 08 1987.