Voxel Surface - Exam Project
Computer Graphics Programming

Roman Karaba

January 2022

1 Introduction

This report is part of the final exam project for the Computer Graphics Programming course at the IT University of Copenhagen. The goal of this project was to implement a voxel-based surface generator. The surface is generated using a Perlin-like noise function of which the output is then used to set the height value for individual voxels.

2 How to run the project

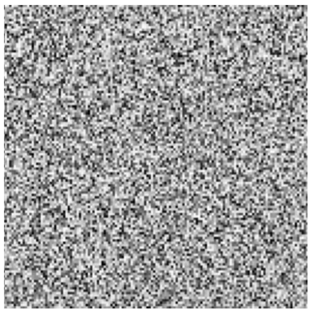
Extract the .zip file into the Graphics Programming Exercises root directory after cloning Henrique Galvan Debarba's github repository[2]. Add the project directory entry into the root CMakeLists.txt file and run CMake to configure and generate project build files.

3 Project Base

The solution for exercise 5. of Graphics Programming Exercises has been used as a base for implementing this project. At first, the codebase was stripped to a barebones state, after which we proceeded to reintroduce some of the stripped functionality in a altered way. For example, the Camera class that can be found on the learnopengl.com website was used, however to be exact we copied the class file from the exercise 12 subdirectory.

4 Voxel Surface using Perlin-Like noise

The voxel surface is being generated using a Perlin-like noise implementation from a tutorial by OneLoneCoder[1]. This implementation is an approximation of the original Perlin Noise implementation by Ken Perlin. While its not exactly the same, it is sufficient for what it was needed to be used for.


(a) Static noise

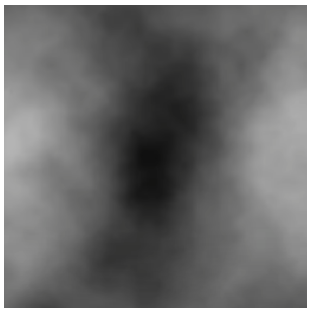

(b) Perlin-like noise

Figure 1: Static vs Perlin-like noise

Generating noise using a simple random function generator produces values that are too random. If it were to be plotted on a 2d texture in grayscale, it resembles more what you can see on a mis-tuned TV with a static grainy image see figure 1a. In order to generate something aking to a landscape surface we need the consecutively generated noise values to be only slightly different from each other while changing in value gain or value loss, which creates areas with groups of higher and lower values see figure 1b.

The included Perlin-Like noise implementation contains methods for generating 1D and 2D noise vectors. The 2D noise vector has been used as a height map for the generated surface. Color of the generated voxels is dependant on the final height position as defined in the fragment shader to illustrate regions of water, grass and rocky peaks. One of the features of Perlin noise is that the resulting vectors have relatively symmetrical values along the edges. This enables us to then to use the output as a tile that we can duplicate combine along its edges to give a perceivably continuous surface. In the case of this project, we use the 2D noise vector and generate a tiled 3x3 surface see 2.

5 Instanced Rendering

Initially in this project we called a draw call for every voxel in the scene. This resulted in extremely poor performance as a single tile of our surface has a size of 256x256 voxels which resulted in over 65 000 draw calls and an average performance of 10 FPS. Trying to create a 3x3 tiled surface with this approach would result then in around 590 000 draw calls which is just technically unfeasible.

OpenGL has an instanced rendering feature[6] that enables us to draw multiple instances of a mesh in a single draw call. We do this by generating a vector of vertex position offsets, generate a VAO and a VBO for it and specify an attribute divisor for the VAO so the pointer on our GPU advances per instance by calling *glVertexAttribDivisor(offsetAttributeLocation, 1).* With this approach we are able to render the 3x3 tiled version with around 590 000 voxels and run at around 1000 FPS on an NVidia RTX2070 Super on a laptop. In order to

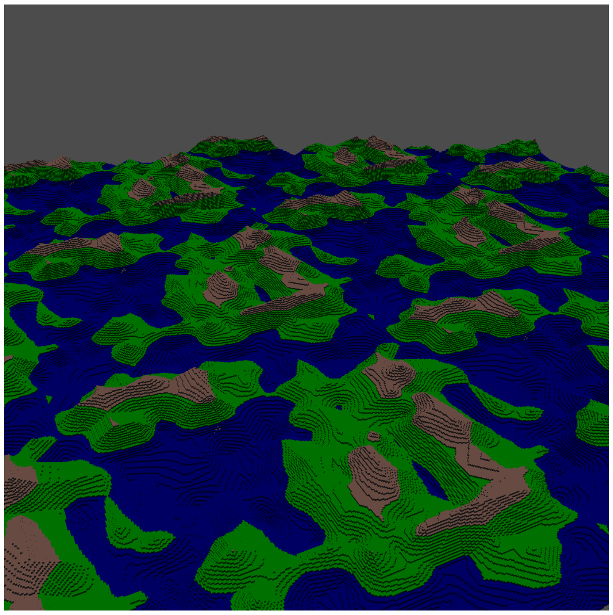


Figure 2: Tiled Voxel Surface

further optimize we enabled backface culling, since that tends eliminate over 50% of fragment shader runs[5] as well as it not being necessary to render the inside of the voxels.

6 Lighting

The lighting in the scene consists of an ambient light component, diffuse light component and a specular light component together creating a Phong lighting model[3] which gets calculated our fragment shader.

6.1 Ambient Lighting

Ambient lighting is a computationally cheap way to simulate light scattering across all of the meshes in the scene. We use it to simulate a sun that illuminates our scene. Based on the rotation of our sun's direction we also simulate a day and night cycle. Calculating ambient lighting consists of multiplying a light color value by a light intensity value and then adding the result of that calculation to the final color value of the computed fragment. This by itself does not produce very interesting results since the resulting color value is added to all of the surfaces in our scene, thus changing the hue of all of the surface colors in our scene.

6.2 Diffuse Lighting

To add to our world more interesting lighting effects, we use calculate a diffuse value. Diffuse lighting is calculated based on the angle between our light hitting our surfaces and their normal vectors. The smaller the angle between the light direction and the normal vector, the brighter the surface appears. To calculate the angle (diffuse value) we can simply calculate the dot product of the light direction vector and the normal vector. We then multiply our diffuse color value with our diffuse value to get a diffuse contribution that later gets added to the final fragment color value.

6.3 Specular Lighting

Specular lighting enables us to add highlights making surfaces appear shinier. Similarly to diffuse lighting, we calculate the specular based on the lights direction and the surface normal however we also add the view direction into the equation. To calculate the specular component, we need to multiply the light color with the light intensity which we then further multiply with the dot product of the lights reflection direction from from the surface and the viewing direction. We also set a dampening factor and raise the dot product to its power. The higher value the dampening factor is set to, the tighter / sharper the final specular highlight on the surface.

7 Scene Controls

We added some keyboard controls into the program that enable control over various parameters of the Perlin-like noise like surface generation.

By pressing the 1 key increment the octave count of the generation algorithm until a max value of 8 after which its reset to 1. This affects the amount of detail that can be seen on the surface, more specifically how many sampling passes get calculated and summed in the end for the noise value. An octave count of 1 is basically a flat plane whereas an octave count of 8 in our implementation often tends ends up being too high and making the surface more noisy and looking similar to the static noise as shown in figure 1a.

Key 2 increments the sampling bias, the lower the sampling bias, the more aggressive the slopes of the generated surface become. Inversely the higher the bias the flatter the surface becomes. We set it to incrementally cycle between 0.5 and 3, anything higher or lower just affects the surface too much or too little.

Key 3 switches the height scalar of the surface in increments of power of 2. We use the height scalar to calculate the final Y position of every voxel, thus the higher the scalar the larger the difference between the lowest voxel Y in the scene and the highest. The height scalar has a range is set to be between 2 and 256. Key 4 executes a re-seed of the Perlin-like noise generator, instantiating a newly generated surface.

Keys 5 and 6 are just simple toggles. Key 5 toggles a skybox in the scene which is disabled by default because in the end the scene looks more visually pleasing without it. It was implemented using the approach from exercise 11, however the cubemap texture is from learnopengl.com website[4].

Lastly key 6 toggles the rotation of our sun light direction that affects the position of the specular highlight on the surface. It is toggled in the render loop, thus it enables and disables a simple day night cycle.

8 Conclusion

We were able to successfully implement a procedurally generated voxel surface using a Perlin-like noise algorithm. The user can use fly around the scene and view the generated surface. It is also possible change the surface generation parameters in real time and observe how it affects the generated surface.

Further possible development of this project could include generating and using a 3D noise map to try to generate cave systems, add the possibility to interact with the environment, use proper texture for the different height levels of the generated mesh.

References

[1] javidx9 / OneLoneCoder. *Programming Perlin-like Noise (C++)*. 2022. URL: `https://www.youtube.com/watch?v=6-0UaeJBumA`.

[2] Henrique Galvan Debarba. *Graphics Programming Exercises github repository*. 2022. URL: `https://github.com/hgdebarba/graphics-programming-2021`.

[3] Joey de Vries. *Basic Lighting*. 2022. URL: `https://learnopengl.com/Lighting/Basic-Lighting`.

[4] Joey de Vries. *Cubemaps*. 2022. URL: `https://learnopengl.com/Advanced-OpenGL/Cubemaps`.

[5] Joey de Vries. *Face Culling*. 2022. URL: `https://learnopengl.com/Advanced-OpenGL/Face-culling`.

[6] Joey de Vries. *Instanced rendering*. 2022. URL: `https://learnopengl.com/Advanced-OpenGL/Instancing`.