EDAN35 Project - Voxel Terrain

Mattias SImonsson
e-mail: dat11msi@student.lu.se

***Abstract*—In this paper, we describe a method for procedurally generating a terrain mesh using voxels. We create chunks of voxels using a 3D noise function and apply the Marching Cubes algorithm to turn the voxels into meshes. We then texture this mesh using triplanar texturing and render it using RenderChimp.**

I. Introduction

Procedurally creating terrain is used in several areas of computer graphics, a notable example being video games. A common way to do this is to represent the terrain as a 2d heightmap, a texture where each pixel represents the height of the terrain at that point. This has the limitation of only allowing a single height at each point which means that generating caves, cliffs and overhangs become much harder. Another way to represent the terrain is by using a 3d grid of voxels. This approach allows us to have multiple heights on a point on the terrain and in general allows for much more complex shapes, but it also requires much more memory. Still, we are going to use a voxel representation for the terrain in this project.

II. Application

The first thing we need to generate a nice looking seemingly random terrain is a noise function. A noise function can be thought of as a seeded pseudorandom number generator. You give it a value and it returns another value. If you pass it the same value twice, it returns identical values each time. The most well known example is perlin noise, which is a very popular noise function. We are going to use a refined version of perlin noise, called simplex noise. Simplex noise was created by Ken Perlin[1], who also created perlin noise, to overcome the limitations of perlin noise. Simplex noise is also faster in higher dimensions. We are going to use a c++ implementation[2] based of simplex noise based on a paper Stefan Gustavson[3]. To allow for caves and cliffs in the terrain we are going to use the 3D version of Simplex Noise with 9 octaves (noise function applied 9 times for each point at different frequencies) of noise.

After we create our voxels we need to turn them into a mesh to be able to render them. To do this, we are going to use another popular algorithm, the Marching Cubes algorithm as described by Paul Bourke[4]. The Marching Cubes algorithm basically turns a density function into a mesh. We give the algorithm eight voxels at a time and it uses lookup tables (which means it's very fast) to represent the voxels as 0-5 triangles.

Since the Marching Cubes algorithm requires a density function to create a mesh, we need to think about what values we set our voxels to. We should think of each voxel value as the density of the terrain at that point, anything above density 0 is air, and anything below 0 is ground. We populate the voxel grid using 3D Simplex Noise and subtract the height of the voxel from the noise value to get a solid bottom layer and non-solid top layer. Then we iterate through the voxels and collect the triangles generated by the Marching Cubes algorithm. We remove duplicated vertices and generate vertex normals for each vertex by averaging the face normals of all triangles the vertex is a part of.

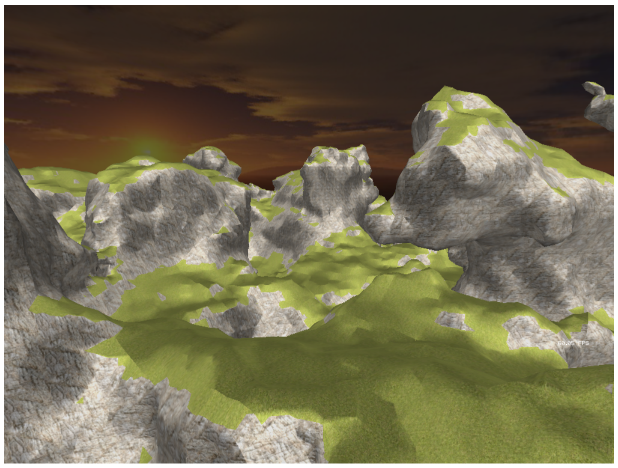

Figure 1. End result

Now we have a procedurally generated terrain mesh without color, texturing is needed. Creating proper texture coordinates while generating the mesh is very very hard, if not impossible, so we are going to use a technique called Triplanar Texturing[5]. At each pixel we do three texture samples per texture using the world coordinate of the pixel, one sample using x and y, one using x and z and one sample using y and z. We then weight these samples using the normal vector of the pixel so that pixels facing straight up (normal (0,1,0)) only use the x-z sample and so on. This gets us nice texturing without stretching the texture at the cost of needing three samples.

We would also like to use more than one texture, a terrain doesn't just exist of rock or grass. Using the normal vector of each pixel, we can decide what texture to use. We calculate the dot product of the normal vector and the up vector and compare it to a threshold. If the dot product is above the threshhold, a grass texture is applied, otherwise a rock texture.

III. Results

I am overaly happy with how the terrain looks, the algorithms worked as I expected. The mesh generation produces a noticable framerate drop but that is partly mitigated by spreading out the generation over multiple frames. There is a memory leak somewhere that I have not been able to find which leads to the program crashing after walking too far away from the start point. I don't think I'm leaking memory anywhere but I'm not used to c++ so I could very well be. The leak finding methods I tried gave no results.

IV. Discussion

There are several things you could do to improve the look of the terrain. Bumpmapping could be incorporated using triplanar texturing, ambient occlusion could also be done to make it more realistic, but most of my time was spent trying to optimize the terrain generation and the memory leak. I did try to reuse all vertex buffers, voxel grids are also reused, but to no avail. I am happy with how fast the Marching Cubes algorithm is though, I expected it to be much slower.

References

[1] http://en.wikipedia.org/wiki/Simplex_noise
[2] http://www.6by9.net/simplex-noise-for-c-and-python/
[3] http://webstaff.itn.liu.se/ stegu/simplexnoise/simplexnoise.pdf
[4] http://paulbourke.net/geometry/polygonise/
[5] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html