Using marching cubes with noise to generate complex surfaces

Filip Bergman[*] Mathias Bothén[†]

Lund University
Sweden

Abstract

In this project the marching cubes algorithm was implemented and used in different ways. It was found that the algorithm can be used to create a wide variety of surfaces, from a sphere to a terrain with mountains. Another observation was that Perlin-noise could be used to create real life like terrains. It was also observed that using the processor to perform the marching cubes algorithm is relatively slow, compared to what is possible with the GPU, since it does these calculations much more efficiently. This results in the program being slow to start up, but once the cubes have been generated it is just as fast.

1 Introduction

In 1987 when a healthcare company wanted an efficient way of visualizing data from CT and MRI devices, a research team developed an algorithm that took a 3D discrete scalar field and created a polygonal mesh of an isosurface. This algorithm was named marching cubes and was however not only useful for medical purposes, but also 3D modeling, since almost any structure can be made with it. In later years this algorithm has been used in many different ways, one being terrain generation by combining this algorithm with noise.

In this project the main goal was to implement the marching cube algorithm and later use it to generate complex surfaces, such as terrain. This was done by using Perlin-noise. This subject was explored since the possibilities of the marching cubes algorithm with noise are endless. It is possible to generate endless worlds using these techniques and one example of this is Minecraft's world generating algorithm, which uses 3D noise to create its terrain.

This project was built upon the framework from the first course EDAN80, Computer graphics.

2 Marching cubes algorithm

The first algorithm that was implemented was the Marching Cubes algorithm. This algorithm proved to be quite challenging to implement. A guide with code was found that both implemented the algorithm on the CPU and the GPU. The GPU version was a bit to challenging for the scope of this project since noise and terrain generation also was to be implemented in this project.

How the marching cube algorithm works is that it creates one big cube consisting of an arbitrary amount of voxels, in this case 32x32x32 voxels. These voxels are small cubes themselves. Within each of these voxels the algorithm will check each of the 8 corners to see if they are inside or outside of the input surface, and sets a value at each corner, 0 if outside and 1 if inside of the surface. These values are then concatenated which will give a value from 0 to 255. This means that if the whole voxel is inside or outside of the surface, i.e if the concatenated value is 0 or 255 respectively, nothing has to be done in this voxel. If this value is 1 to 254 however, it means that 1-7 corners are inside the surface and that part the surface should be drawn in this voxel. This is done by covering the corners that are inside the surface with polygons as can be seen in figure 1. Since there are 8 corners, there are 2^8 = 256 different

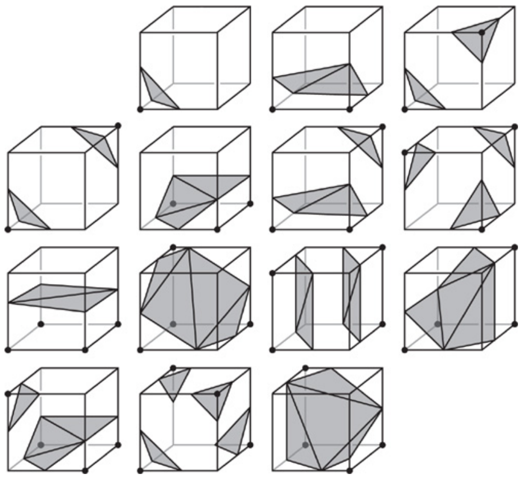
Figure 1: The 14 unique polygon structures.

combinations, but only 14 unique ones, the ones showed in figure 1, the other ones are just rotated versions of these 14. To sum it up, the algorithm iterates over each voxel inside the box(32x32x32 = 32 768 voxels) and generates polygons if the surface intersects with that voxel, otherwise it does not generate anything in that voxel [Nvidia 2020].

The first step taken to implement the marching cubes algorithm was to create the edge and triangle lookup tables. The edge table consists of each unique constellation of edges that have a vertex of a polygon positioned on them. The triangle lookup table consists of all the 256 different voxel constellations. These tables were found online, along with the algorithm that takes the concatenated corner value along with some other parameters and maps it to the right polygons that should be generated [Bourke May 1994].

The second step after this was to implement the code that iterates over all the voxels. The solution to this was a modified version of a solution found online [Halayka 2020].

The result of this first implementation is shown in figure 2 and had random values as input. After applying some diffuse shading and adding a texture to the polygons, as well as using the mathematical formula for distance in 3D from the center of the cube as the surface, a sphere could be generated as seen in figure 3.

3 Terrain generation

Creating terrains using marching cubes has the advantage of being very flexible. Once created, simple tweaks to numbers can change the density of the terrain, creating more/less caves or maybe mountains. The flexibility comes from the fact that the world has a bunch of values scattered evenly across(described in the marching cube algorithm). Depending on one value, this part of the world is either inside or outside the terrain. Outside meaning air. The threshold that decides what values are inside the terrain can be changed in real time and modified to a suitable number.

To create a terrain, each point around the world would simply

[*]e-mail: fi5731be-s@student.lu.se
[†]ma6728bo-s@student.lu.se

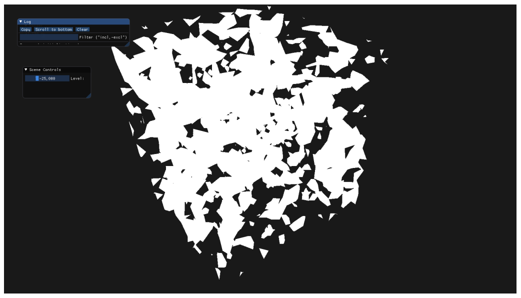

Figure 2: Marching cubes with a simple shader and randomized values.

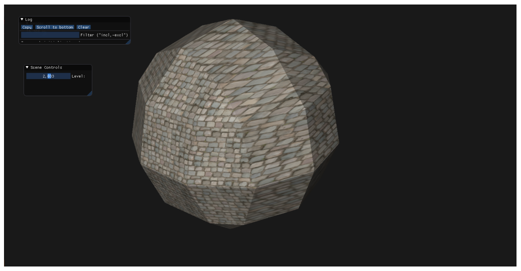

Figure 3: A sphere generated with marching cubes algorithm, with a relatively low resolution.

have to be filled in with values. However these values must have a pattern that mimics the real life terrain. One could go in and manually place the values but this would take ages and is not very practical and since an infinite terrain as well as being able to change it in real time was an aim of this project. Thus another way generate terrain had to be found.

3.1 Noise

Noise is a way to generate psuedo-random values that have a more natural/harmonic succession of numbers. The human eye is very good at picking up on patterns and can easily tell the difference between something that is totally random and something that has a small pattern. This was first created by Ken Perlin and was aptly named Perlin noise. He created this in response to his frustration of how computer generated imagery looked so machine made back in 1983. Noise can come in different structures, mainly in which dimension it is made. A noise can be made in any dimension, but most useful to this terrain generation are Perlin noise in 2d as well as 3d.

Different dimensions of noise have to be interpreted in different ways depending on its application. The application of 2d noise in a terrain is often most simple since it can be used as a height map where higher values mean that part of the terrain is higher up in the air while lower values mean its lower. This could be compared to maps where density of lines mean steeper cliffs and hills. The use of 3d noise can be quite abstract, but as already described, these values can be used to determine whether or not a part of the world is part of the surface/terrain.

3.2 Implementation of terrain

To create the noise, a pre-built library by Ryo Suzuki was used[Suzuki 2020]. This was easy to implement and had functions ready to be used. While 3d Perlin noise is very useful, the result of its values look nothing like a terrain or anything realistic. Because of gravity, there is a higher likelihood that parts further down are part of the surface and parts higher up are part of the air. This

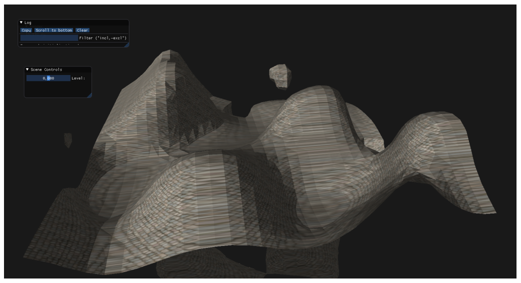

Figure 4: First iteration that started to look like a terrain.

was implemented by having parts below a certain level(an average surface height) get lower values, meaning higher chance of being inside the surface, and parts that are above this level have higher values meaning a bigger chance of being outside the surface. This allowed the surface to be more flat as can be seen in figure 4. This, although much more like a terrain, was way too smooth and round. To make this more realistic, it had to be a bit more jagged/uneven. This was done by sampling from the noise multiple times at different places and then adding the values together, but with every iteration they had less and less impact. This way, the big hill shapes were still preserved, but they had more detail on them. A problem became clear that there was a maximum height of mountains/hills and a lot of them was the exact same height. To combat this, a 2d noise was sampled as a height map so that certain areas would get higher altitude. However this removed some of the caves and overhangs so the impact of the 2d noise had to be tweaked so there was still caves while also having taller mountains.

4 Other additions

After a marching cubes chunk could be generated to look like a terrain, improvements were made to the general gameplay experience. A chunk is relatively tiny and if one were to fly around the world then they would come to the ”edge of the world”. To fix the problem of the map being too small, several chunks were added to build a 4x4 chunk array. Every chunk sampled from the 3d noise using their world coordinates and since the chunks laid next to each other, it created a smooth transition so that the world continued. To stop a player from falling off the edge of the world, the terrain would have to follow them around. Once they got too close to an edge, the chunks would move one step in the direction of the player, making sure that the player was always around the center of the terrain. This also meant that the world was now infinite and the player could explore in any direction.

5 Results

After the marching cube algorithm and noise algorithms had been combined, as well as some tweaking with some parameters in the noise to make the terrain look more realistic or just more aesthetic. Some sliders were also added to the GUI to enable testing different mountain heights, frequency, detail and similar.

The aesthetic additions made were things like changing the texture to a rock texture if the dot product between the y axis and the normal of the triangle was above a certain value, making it look like grass cannot grow on vertical surfaces. Water was also added at a y = 0 and it was made transparent so that terrain below it could be seen. Below this level as well as a tiny bit above, a sand texture was added to create nice looking tropical beaches. Also above a certain y value a snow texture will be used if the dot product of the normal and y axis is above a certain value. The rock texture was also used in case the terrain was above a certain height so that mountains would not have any grass on them. If none of these textures

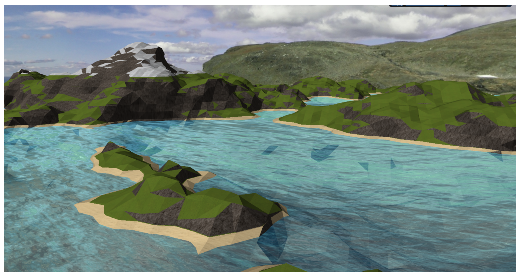

Figure 5: Final terrain.

are used then grass will be used. A skybox was also added to create a nicer atmosphere. The result of these aesthetic additions can be seen in figure 5.

Tweaking the surface level to be lower allowed for the terrain to display more water and gave a look of a tropical place while increasing the surface level added more mountains and snowy hills.

6 Discussion

Even though there were a lot of algorithms and aesthetic additions made, this is a project with infinite possibilities. However before any additions should be made, the most important thing to do would be to change the marching cube algorithm calculations to the GPU instead of the CPU.

6.1 Improving performance

This would enhance the experience greatly since now when the player goes to an edge chunk, all 4x4 = 16 chunks have to be calculated again, which takes some seconds for the CPU, which freezes the program. This transition would be seamless with the GPU. Moving the calculations to the GPU is simple in theory. A compute shader would be used to do the exact same calculations but the difficulty lies in sending the data back and forth between the CPU and GPU. This information can only be passed by using images but figuring out a way for both to read and interpret the data they are given is difficult. Couple days of testing later and progress was given up, simply too big of an undertaking. Implementing this would be easier if it was done from the start. Another addition that would further improve performance would be to only generate the chunks that are new and not the ones that have not changed when the player moves around. There is also the fact that every triangle have their own 3 vertices and even though they overlap with other triangles new vertices are created and many vertices are in the exact same place. A way to solve this would be to use indices to connect triangles to unique vertices, however the math to solve this problem is very confusing because of how the triangles are generated in almost random order and could be connected with any other random triangle.

6.2 Terrain variation

At the current state of the project, every new chunk generated has the same parameters as the other ones, which makes the terrain look very repetitive. What could be added here are biomes/climate zones. For example for some chunks, the terrain will be mostly flat plains with grass and some lakes, and for other chunks, high mountains with snow. In the future it would be interesting to try to apply this solution to a sphere and make whole planets.

One thing that was attempted was to create caves by modifying the 3D noise function, however it proved to be quite difficult. There was always a trade-off between the caves and above the water level. If the caves worked, the mountains looked off, and the opposite. It was decided that the above surface was more important to focus on for this project although there is still a possibility for caves and overhanging cliffs, the numbers just have to be tweaked correctly.

7 Conclusion

In conclusion, this project has been very interesting and fun to develop and something we want to further work on after this course. There are still so many improvements that can be made that we could spend weeks to solve, however we would not mind since the results are so intriguing.

References

BOURKE, P. May 1994. Polygonising a scalar field.

HALAYKA, S., 2020. julia4d3. [Online; accessed 13-December-2020].

NVIDIA, 2020. Gpu gems 3: Chapter 1. generating complex procedural terrains using the gpu. [Online; accessed 13-December-2020].

SUZUKI, R., 2020. Reputeless/perlinnoise. [Online; accessed 13-December-2020].