COMP 4490 Project Report

Procedural Terrain Generation and Terrain Editor using the Marching Cubes algorithm on CPU

Name: Akshay Sharma
Student Number: 7859678
Instructor: John Braico

Table of Contents

Table of Contents **2**

Introduction **3**

Marching Cubes Algorithm **3**

Calculating Triangles In a Subcube 3

Marching Cubes for Procedural Terrain Generation **5**

Density function **5**

Understanding Noise Functions 6

2D Noise vs 3D Noise 6

Playing with noise 7

Implementations **7**

Procedural Terrain Generation 7

Terrain Editor 8

Future Work **8**

References **10**

Introduction

Terrain generation is a task traditionally done using heightmaps to calculate elevation and rendering mesh using the elevation. In this project, we use the marching cubes algorithm and different density functions to generate terrains procedurally. We start by introducing the marching cubes algorithm, its density function, and how it can be used to render 3D meshes. We then explore some different density functions and how they can be layered and manipulated to generate different features. We also compare 2D Perlin noise to 3D Perlin noise, observing the features that can be introduced by 3D noise in our terrain. We then go through a broad overview of the key parts of the implementation of the terrain generation and terrain editor programs. We finally conclude the report by discussing future work that can be done for this algorithm including a broad overview of how this program can be generated on the GPU.

Marching Cubes Algorithm

The Marching Cubes is an algorithm used to create a triangle mesh from an implicit “density” function defining the surface of our geometry. The density function is given a point in a 2D or 3D space and the function returns a single floating-point value. These values that are returned by the density function vary over space and can be positive, negative, or zero at any given point. These values indicate the state of the supplied point with respect to our surface that is defined by the density function. The point is inside the surface if the value is positive, outside the surface if the value is negative, and on the surface, if the value is zero. Our goal is to construct a polygon mesh along the surface of our geometry, where the value returned by our density function is zero.

In the Marching cubes algorithm, we render the surface one cube at a time. Each cube is subdivided into n subcubes. We then calculate the density value for the vertices of each of these subcubes and use these density values to render triangle primitives within each subcube. These primitives are calculated one subcube at a time as we “march” through the space of our cube which is where the algorithm gets its name. When these triangle primitives are connected, we get the triangle mesh for the surface defined by our density function. To understand how the algorithm calculates primitives, let’s take a look at how triangles are calculated within one subcube.

Calculating Triangles In a Subcube

For each subcube, we have the density values that we calculated for each of its vertices. We use these density values and their signs to see which vertices of the current subcube are inside or outside our geometry. The signs of each of our vertices and either be positive or negative (a point can be either inside or on/outside the surface). This means that we have a total of 2^8 = 256 configurations or cases for a cube. We use a triangulation table referenced by the case number to retrieve information about which edges will be used to calculate our triangles and

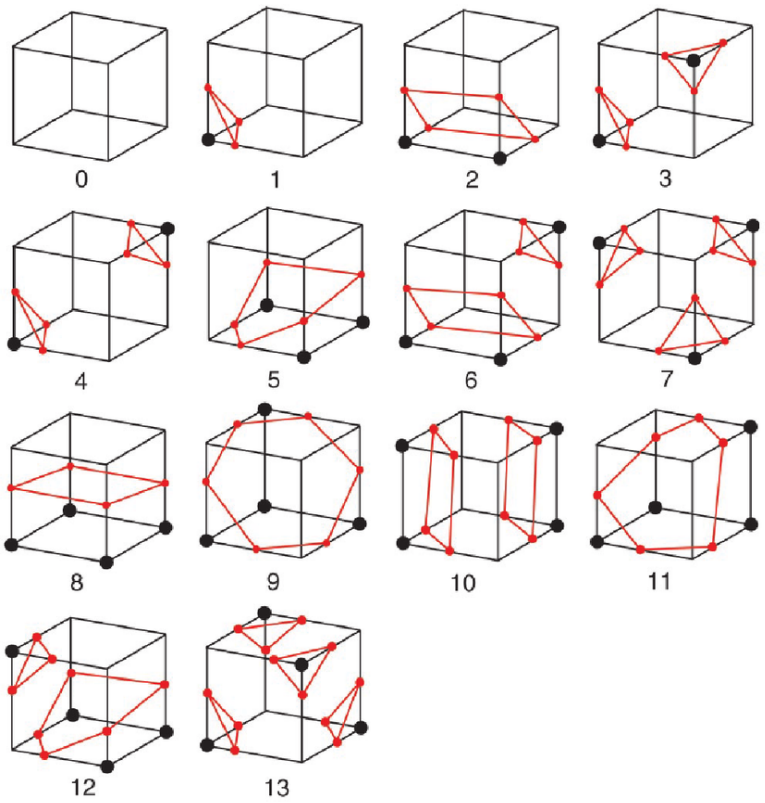
how many triangles will be created within the current subcube. The two most trivial cases are as follows:

If the signs of all the values are the same, then it can mean either of the following two cases:

1. All points are positive: All the points are inside our surface.
2. All points are negative: All the points are outside our surface.

In either of the two aforementioned cases, we don't make any triangle primitives inside our cube as we are only interested in rendering the surface of our density function (density value=0).

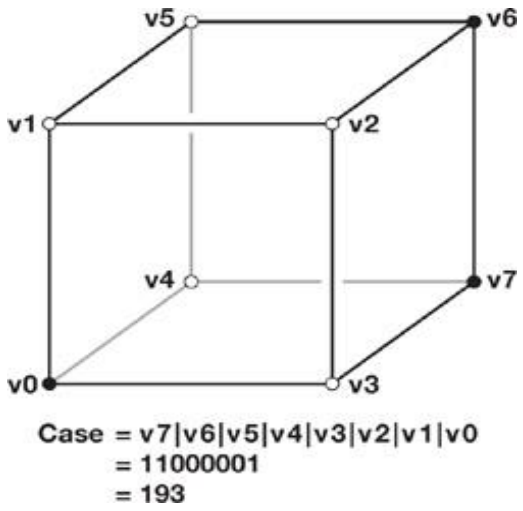
For the rest of the cases, it is observed that there are only 14 unique configurations for the geometries in the subcubes and the rest of them are just variations of them. The 14 unique cases are displayed in the following image:



As discussed previously, the algorithm uses a triangulation table with the case numbers to check which edges will be used to calculate the triangles. But how do we determine the case numbers and how do we calculate the exact point that is to be used on the resulting edge? The solution to this question is as follows:

Firstly, to calculate the case number, we take all the vertices of our subcube and assign a bit to each of them. A bit is set to 1 if the corresponding vertex has a positive density value and 0 if it had a negative density value. We then perform a bitwise OR operation to concatenate these bits

and then convert the resulting binary value to get an integer case number. An example with vertices 0, 6, and 7 (resulting in case 193) set to one is as follows:



Finally, to calculate the exact point on the edge that is to be used for the triangle, we have one of two options:

1. Use the midpoint of the edge: This is a crude approximation of the exact point on our defined surface and hence results in blocky-looking textures.
2. Approximate where the density on the edge will be zero: Since we have the density values at both of the endpoints of an edge, we take a weighted average of the two points to get the position on the edge where the density values are zero and use that point. This leads to smoother-looking surfaces.

This triangle calculation process is repeated for each subcube and the resulting triangle primitives join together to form our terrain. But how do we make terrains using this?

Marching Cubes for Procedural Terrain Generation

As saw in the previous section, we can render a surface geometry defined by a density function using the marching cubes algorithm. To use this for Procedural Terrain Generation, we sample an interesting density function to create features like mountains, trenches, flatlands, overhangs, etc. We take a look at interesting density functions in detail in the next section.

Density function

Now that we know how to create meshes using the marching cubes algorithm, we create interesting density functions to be sampled for generating our terrain from our world space

coordinates. The density function we use in this project is created by layering simplex noise over a flat surface to create elevations and tranches.

Understanding Noise Functions

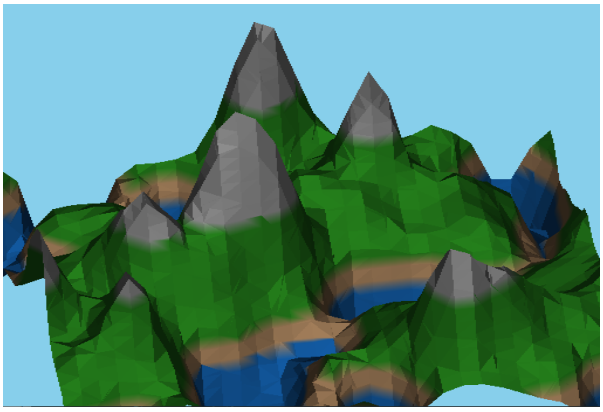
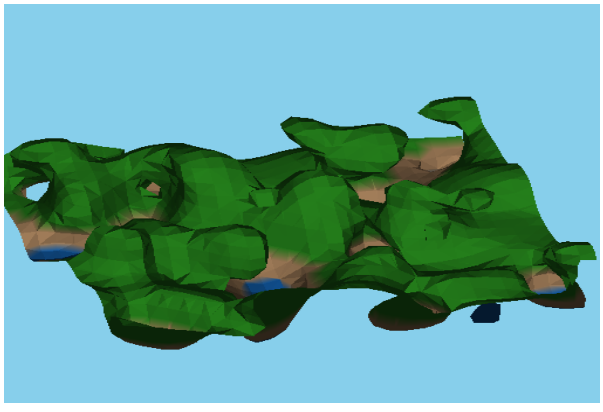
We use a noise function to displace our initially flat terrain creating a terrain rich in features such as elevation. However, to do this, we cannot use a random noise function as the terrain is supposed to be uniform instead of completely random. **Perlin noise** is a noise function popularly used for terrains as this function outputs a pseudo-random value that is not completely unrelated to the previous value, but randomly increases or decreases over time which gives us more natural-looking terrains. We input our world space coordinates instead of time in Perlin noise and get a continuous output that we then layer over a flat surface to create our features.

Simplex Noise is a noise function comparable to the Perlin noise but with fewer artifacts and a lower computational load, and hence it is used instead of the Perlin noise. We sample simplex noise at different frequencies and layer different octaves of the noise on top of each other to create interesting features. This is discussed in detail in the further section of the report.

Simplex noise is a noise function that can be used in multiple dimensions. The one-dimensional case is not of interest to us in this project as it only creates a line graph. We take a look at 2D and 3D Perlin noise in the next section for generating terrains.

2D Noise vs 3D Noise

When 2D noise is sampled, it can be used to displace a flat surface along the XY plane to create landscapes. 2D noise however is limited to creating peaks and tranches. This is where 3D noise comes in. Using 3D noise, we can access another dimension to further displace our flat terrain. In physical terms, this leads to interesting features like cliffs, overhangs, and caves, which better mimic naturally occurring landscapes. 3D noise however also leads to features like floating rocks and other terrains which are not always physically possible. A comparison of terrains generated using 2D and 3D noise can be seen in the following table (NOTE: These images are rendered by layering different octaves of noise on top of each other):

	
2D Noise	**3D noise**

Playing with noise

So far, the noise/density function that is used to create our terrains gives us uneven elevations. Intriguing as it may be, it is still very limited, and hence, in this section, we explore how we can modify and layer this noise to make more interesting features for our terrains.

1. Frequency: This value indicates the frequency at which we sample our values. In the context of timesteps, these would be the intervals at which we sample our signals. For our use case, since we use the world coordinates of a point to sample Simplex noise in different dimensions, this can be intuitively thought of as increasing the area that we are sampling for our terrain. This is useful because if only a small area is sampled, we do not see any interesting features emerging from our terrain.
2. Octaves: When we sample our noise from the function, to make our elevations more interesting, we layer noise from different frequencies on top of each other. We half the amplitude of each of these layers so that we don't end up superimposing many intensities. When the intensity is halved, the resulting terrain looks a lot more natural with smaller hills emerging from larger hills.
3. Redistribution: using frequency and octaves, we can make natural-looking hills but not flat surfaces/valleys, to do this, we raise our elevation (the result of noise) to a variable factor. Flatter valleys start to appear when this factor is small and high peaks emerge when this factor is large.

Combining these features can give us a variety of terrains like the ones in the figures we previously looked at. We can generate a variety of noise functions in this way and create different biomes for our terrains based on elevation using texture maps to make stunning terrains. Now that we understand how terrain generation works with the Marching cubes algorithm and noise functions, in the next section, we take a look at a brief overview of the implementations.

Implementations

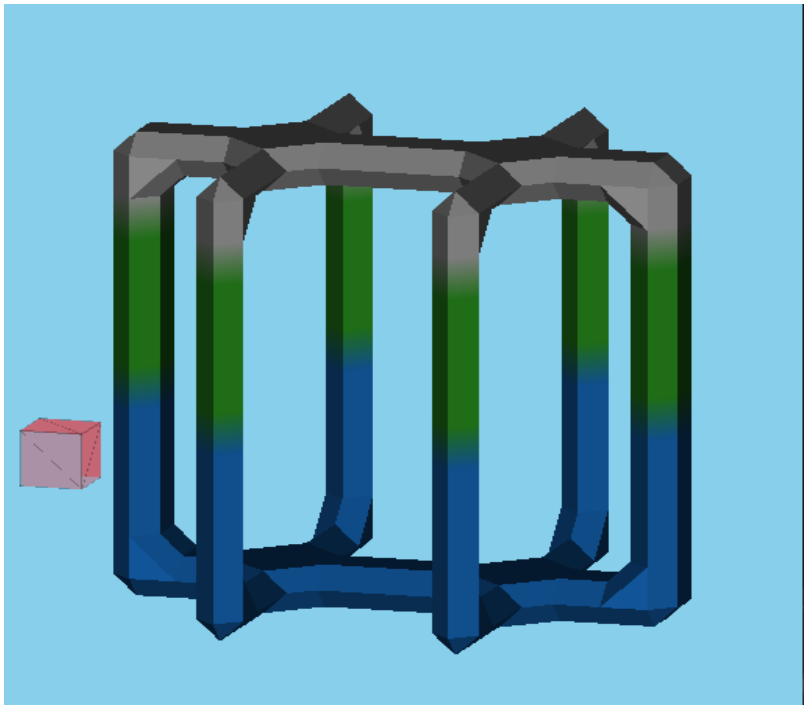
This section provides a brief overview of the implementations done for this technique in OpenGL C++. This project contains two scripts: ProceduralTerrain.cpp and terainEditor.cpp. Brief descriptions of these scripts are as follows:

Procedural Terrain Generation

This script contains a CPU implementation for the procedural terrain generation described in this report and provides various controls to navigate the procedurally generated world and tweak the parameters like the number of octaves, frequency, and intensity. It also provides functionality to switch between both 2D and 3D noises to observe both of those terrains.

Terrain Editor

The terrain editor script is a drawing program implemented using the marching cubes algorithm that allows the users to navigate a cube-shaped brush in a 3D space using keyboard controls to create and erase 3D terrains. This tool implements collision testing between the brush and the vertices of our 3D subdivided cube. It sets the vertices it collides with as on or off and renders the terrain accordingly. The user can turn off the drawing mode to then inspect the created terrain and move the brush around the space without altering the current drawing. An example of a structure drawn using this tool is given below.



Future Work

This project is a CPU implementation. The CPU implementation however is too slow to navigate terrains in high quality in real-time. The user must wait for the terrain to render (less than a second for each step) to explore the procedurally generated world. This implementation can be sped up considerably using a GPU as the calculation for each subcube can be done in parallel using a GPU.

A future task I am planning to implement is the GPU version for this project. I spent some time studying how the marching cubes algorithm can be implemented on the GPU. I wasn't able to do it within the project deadline but the key notes for the GPU implementation are as follows (in case anyone reading this report is interested).:

NOTE: this is a two-pass implementation

FIRST PASS

1. The CPU implementation uses structs and vectors to hold the vertex positions and densities for the sub-cubes. To do this with a GPU, we use the concept of a **3D texture**. A 3D texture is just an array of size n containing textures of size nxn, where n is the number of subdivisions. We use this texture to store the density value of each corresponding grid point. This is populated with the density values by using the fragment shader and render to texture (using frame buffer objects).

SECOND PASS

1. In the second pass, we pass the vertices and density values to a geometry shader. The geometry shaders are capable of taking in one geometry and outputting in another. This is a useful property as we supply the origin point of our subcube to the shader and the shader outputs anywhere between one to five primitive triangles for our mesh.

Another future task is to add texture mapping to the terrains based on elevation. This will make the terrain look more natural than using plain colors.

References

1) https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu
2) https://www.redblobgames.com/maps/terrain-from-noise/
3) https://github.com/SRombauts/SimplexNoise
4) http://paulbourke.net/geometry/polygonise/

END