TERRAIN GENERATION PROJECT

PHIL ETTER (PETTER@)

CRYSTAL QIAN (CQIAN@)

DESCRIPTION

INTRODUCTION

We made a terrain generation program. Specifically, we used diamond-square algorithm to generate a height-map across a square mesh. Then, we used texture mapping to make the mesh look like a mountainy terrain.

This approach could be useful to make randomly generated terrains in video games (such as in Terraria, etc.) where the user could span in an infinite possibility of locations. Learning how to model realistic land models could also aid in producing geographic simulations for experiments and research.

FEATURES

- When the **"G"** key is pressed, a new terrain is generated.
- When the **"C"** key is pressed, the camera toggles between two modes:
    - o The default is **rotating camera mode**. The mouse wheel controls zoom, and dragging the mouse around moves the terrain about the origin.
    - o In **first-person mode**, pressing the **"W"** button moves the camera forward; **"S"** moves the camera backward, **"A"** moves the camera to the left, and **"D"** moves the camera to the right. Dragging the mouse in this case rotates the camera, although there is a bit of a buffer to the camera rotation range so that the user can't look completely up or completely down.
- The **"~"** (tilde) button toggles wireframe view.
- The **"ESC"** key closes the program.
- Parameters can be switched in the **Config/Config.xml** file, such as the size of a terrain patch, the maximum height of the terrain, fullscreen/windowed mode, the texture scale, the texture type, etc. This is described in more comprehensive detail in the file.

PREVIOUS WORK

There have been many previous attempts at generating random terrain, most of which employ rendering and mesh techniques which are beyond the scope of this class. For example, here is a collection of literature and works on terrain generation.

However, we wanted to see if we could achieve convincing results using a comparatively simple algorithm and rendering implementation. Here are two implementations of random terrain generation that use a similar approach to what we implemented:

1. Game Programmer Fractal Terrain
2. Realistic Terrain Rendering

While these implementations may generate convincing geometry, the meshes generated are not convincingly rendered - which is where our projects differs from these implementations. Furthemore, our terrain generation algorithm is slightly different.

PREVIOUS EXPERIENCE

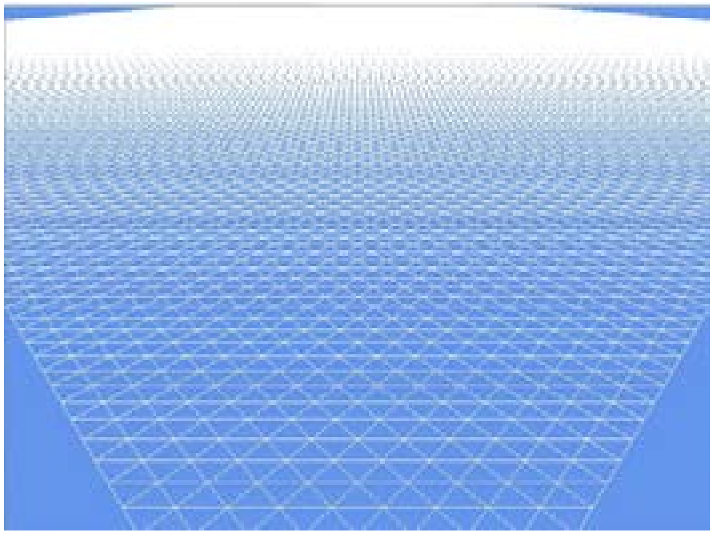
Phil has been making games for a while, so he had a lot of experience which proved useful when writing the basic framework in C# with OpenGL and working with Visual Studio. In fact, I used some of his code to learn the basics of writing a solution and in writing the camera feature. However, the only experience I've had was working with textures in the rasterizer in Assignment 3 and meshes in Assignment 2.

We decided to combine what we learned in those two assignments (as well as in class lectures on cameras, shading, etc.) to make this program.

PROCEDURE

We took Andy Nealen's advice and gave ourselves a fix amount of time to finish the project: in this case, until Friday (5/15/15).

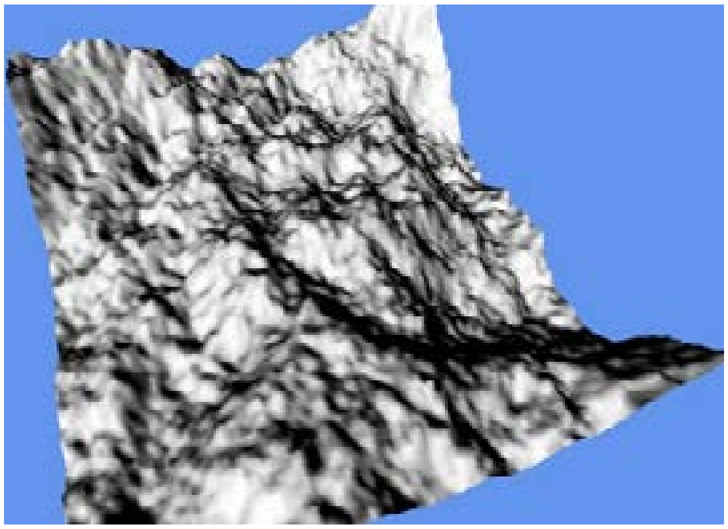
First, Phil wrote a bunch of framework code to get OpenGL to display a mesh which he manually generates from a heightmap, a 2-d array of floating point numbers.



After this was complete, he focused on the actual terrain generation. As we stated before, the approach we use is similar to the diamond-square algorithm. We begin by generating a small 2x2 heightmap of random samples within a specified range, and then continuously refine the heightmap over a certain number of iterations.

The heightmap is conceptualized as a grid of rectangles with samples at the vertices of the rectangles. Each refinement places a heightmap sample in the center of every one of these rectangles and sets its value to the average of the surrounding samples plus a random term which is specified as a parameter and halved each iteration.

Similarly, we place a heightmap sample at all the midpoints of the edges of all the rectangles, and set its value to the average of the adjacent samples and a random term computed as with the rectangle centers. However, after experimentation, Phil decided the algorithm produced a better result if the random term applied the the rectangle edge midpoints was reduced by a factor of 1.5.



To render the terrain realistically, we also required accurate normal data. To compute normals for each vertex of the terrain mesh, we took the cross products of adjacent outgoing edges of each vertex in the clockwise-direction, averaged them together, and normalized the results.

Afterwards, we were able to produce the result to the right.

PROCEDURE (CON'T)

TRIANGLE STRIP APPROACH

At this stage in the project, we were limited to seven iterations of our algorithm to compute our heightmap because the number of vertices on the mesh became so large that we were unable to use 16-bit index buffers. So, Phil modified the program to cut the terrain into 128x128 chunks and render each chunk in a separate draw call.

He also implemented the triangle strip mesh topology (instead of the triangle list topology we used before) to reduce the size of the index buffers generated. To do this, he cut the terrain data into strips and connected the strips using degenerate triangles.

The result of this change was substantial: before, our default settings for seven iterations of the algorithm generated a mesh with 98, 304 indices and around 10,000 vertices. Phil lowered this to 33, 281 indices, reducing the primitive count by a factor of three.

However, the first time he implemented the triangle strips, the triangles had the wrong orientation. He tried flipping them by reversing the index buffer, but discovered that the orientation of a triangle strip with an even number of vertices is stable under reversal. So, another degenerate triangle was added to make the index count odd. Then, the index buffer was reversed to get the triangles to face up.

Thanks to these changes, our final project implementation can use eight or more iterations of the algorithm.

FRAGMENT SHADER

Then, I created new GLSL shaders to give the terrain the correct appearance.

There are two terrain texturing options:

- **"Textured"** gives a uniform grassy hill terrain,
- **"Multitextured"** shows a terrain with snow, rock, dirt, and grass.

The seamless, tileable textures we used in this project were taken from MaxTextures. I looked at a few papers on texture splatting and used a lot of linear interpolation to make the different textures blend seamlessly.

Each location has both a height weight (using its position) and a slope weight (using its normal). Lower slopes suggest plateaus or hilltops, where snow or grass could reasonably fall. Above a certain percentage of the heights, snow would fall. Below a certain percentage of the heights, grass would fall. In between these two height constraints would show an interpolation of grass and snow. Then, as the slopes became steeper, we would see the rock and dirt foundations of the mountains.

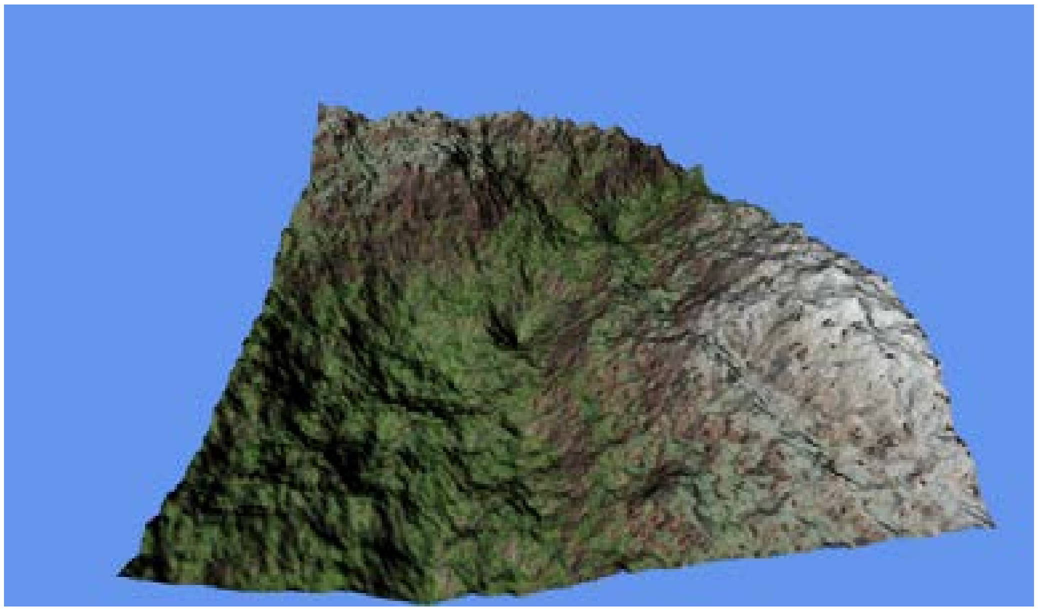
Interpolation was important; if I set an arbitrary cutoff minimum height for snow, for example, without diffusing the snow downwards, there would be awkward and unnatural patches in the mesh.

Additionally, the lighting at each point factors in a Phong calculation without specularity based off its surroundings and the environment light source.

NEXT STEPS

After we implemented a basic terrain, we had time to put in addtiional features (see "Description"). I wrote the camera view, and Phil implemented the one-keypress toggling for wireframe and camera, as well as set up the config file/ **"G"** terrain generation.

Here's what a randomly generated terrain looks like:



ADDITIONAL IMPLEMENTATION

Here are some ideas for what we could do to expand on this project:

- Generate different kinds of terrain. Right now, we just show a hilly, mountainous region. But, by tweaking the parameters, writing different frag shaders, and adding now textures, we could simulate deserts, ocean waves, etc.
- Implement more randomly-generated landscape features, such as trees, rivers, and even moving animals.
- Expand the size of the mesh so that it covers the whole scope of what the user camera can see. In first-person camera mode, the user could explore the terrain endlessly.
- Ability to save terrain data to disk / save a terrain generation seed for a specific terrain

SUMMARY

Considering the implementation time, we think the terrain looks quite good. Overall, this was a fun and interesting project within the scope of the timeframe. We learned how to apply concepts learned in class (specifically, using meshes and lighting), to make our own project, and also learned how to effectively budget time and resources to finish on deadline. This was a completely new experience for me so I had a bit of a learning curve; however, I'm now more familiar with how to implement solutions in C# and work with OpenTK (the C# wrapper for OpenGL we used). Phil learned more about optimizing mesh generation for scalability.