Real-Time Terrain Generation utilizing Ray Tracing

Austin Stone, as46569, austinstone@utexas.edu

August 26, 2015

1 Introduction

As I was deciding upon a topic on which to focus my project, I was intrigued by the idea of creating fractal-esque three-dimensional outdoor terrain, similar to what might be found in today's video games or used as the backdrop in scenes from movies. Although the exact method which I used to achieve the effect evolved and shifted from my original ideas as the project progressed, the end result was quite similar to what I envisioned: an interesting 3-dimensional space that can be traversed by the user. my project consists of an OpenGL fragment shader and an accompanying C++ program that create a mountainous world through which the user can move and explore via keyboard input. After achieving the basic effect of a mountainous terrain, I then progressed to add in a series of additional features to my generated world, which will be discussed in this paper.

2 Early Work

As I began my project, I used information provided by [4] as a helpful starting point to learn about fractal-based terrain generation. Their initial example involved creating a simple "skyline" that mimicked the effect of looking at a mountain range from the side. This smyce also mentioned utilizing the diamond-square algorithm [**?**] in order to create a two-dimensional mesh of uneven terrain. I first began by trying to implement these two examples.

First, I was able to reproduce the basic idea of drawing a set of connected line segments by using OpenGL to give the aforementioned jagged "skyline" appearance. Because this was mostly just a proof-of-concept, I quickly moved on to attempting to implement the diamond-square algorithm. The diamond-square algorithm works by using a series of averaging steps (with added random perturbations) to methodically fill in a grid with weightings. These weightings can then be used as height values for a mountainous terrain. We were successful in implementing the basic diamond and square passes of this algorithm, and I verified my work by inspection of command-line print outs. The only aspect that required further work was properly wrapping the averaging for the edge cases.

After working to implement this algorithm, a problem began to arise, however. I wanted each instance of the shader program to have full knowledge of the world environment. I desired this because at the beginning of my project, I was unclear exactly on how I should create lighting and color using a GLSL shader. To us at the time, it seemed simplest to have each shader be injected with an entire view of the world geometry, and to operate from there.

I knew that the height map that I would generate would need to be very large in order to create an expansive, smooth-looking environment; however, as I researched the size limits of variables that can be passed into an OpenGL fragment shader, these size limits seemed prohibitive for passing in this height map. Another possible option, having the diamond-square algorithm run within the shader, seemed unworkable since each individual instance of the fragment shader (for each pixel) would need to generate the exact same height map in order for there to be continuity. Given that the diamond-square algorithm revolves around the use of random perturbations on the computed averages, each shader execution would in all likelihood generate a completely different height map. In addition to this issue, I also faced the question of how the program should behave when the user reaches the end of the terrain map, however far out that might be. Was it even possible to generate a partial update to the diamond-square algorithm product such that it would extended beyond its original size?

With all of these questions, I began to research and look for ideas from the works of others. I had earlier found a shader [2] that generated terrain much like I was desiring to do. I took a number of hmys to break down this shader code and understand how it enabled each shader instance to have knowledge of the complete world geometry.

3 General Approach - Implied Geometry

As I studied the fragment shader that I had found, I found much of it to be initially confusing; however, by slowly and methodically working and talking through the code, I began to understand the approach that was taken. A principal difference of this shader from my previous work was in how it attempted to generate the height of its terrain at various (X,Z) positions in three-dimensional space. Rather than pre-computing a height map for the entire world view in C++ or shader code, a deterministic function was used to take an arbitrary two-dimensional coordinate and generate a height for it at runtime; the entire world geometry was implied by this function. This allowed for all individual instances of the fragment shader to have an agreed upon height for a given (X,Z) position, and it allowed this without requiring any height data to be passed into the shader. In the C++ code, only fmy vertices are generated at the corners of the viewing screen, and the shader program handles the creation of all of the geometry in the world.

While I wasn't originally planning to do ray tracing, I later realized that this technique, which allowed each shader program instance to have a shared view of the world geometry, works great for ray tracing. Rays can be traced into the world in parallel in each instance of a shader program running on the GPU. Traditional techniques, such as defining the world geometry in terms of OpenGL primitives such as trimesh, do not allow each shader to know what resides in the world, and therefore make it difficult if not impossible to do ray tracing in parallel on the GPU because each shader instance does not have knowledge of the entire world geometry.

The process of generating an image begins by placing a hypothetical camera object at a given (X,Y,Z) location in the world. For a given pixel, the shader works by stepping along a ray emanating outward from the camera through the pixel's location. As an optimization, the step size is determined by checking how high off the ground the ray currently is. The higher the ray is, the further the ray is from hitting something. If the hash-given height value at the current position on the ray was found to be larger than the Y-value of the current position on the ray, then the ray was declared to have hit the terrain. A more refined-search algorithm (using a binary search) was then employed in their shader to hone in on the exact point at which the ray's Y-value became equal to the height value provided by the hash function. With these intersections being computed, a basic silhouette-like view of the terrain could be generated. This general approach of stepping along rays was also discussed by [7].

This approach of using a hash function to generate the heights and then stepping along rays in order to find intersection points made much sense to us. Consequently, I employed this approach in my own shader. I made use of the same hash function to generate my heights, and I employed the same two-tiered approach of finding intersection points (first with a linear search and next with a binary search). Regarding the search intersection code, while I utilized the same approach, I spent a number of hmys in reconstructing the code such that I could justify the logic being performed at each step. Concerning my camera setup, I made use of a slightly more explicit "viewing window" concept in which each ray's direction is determined by calculating the direction from the pinhole camera to a particular location on the viewing window, which sits at a constant distance in front of the camera. I am able to rotate the viewing window by applying a rotation matrix to the ray starting locations (which correspond to individual pixels on the viewing window). I can translate the viewing window by simply changing the location of the pinhole, from which the viewing window and ray starting locations are determined.

In general, this process of deconstructing existing code, walking through it slowly to grasp it, boiling it down to its essential complexity, and then reconstructing it in a way that was clearer and constructed by principles that I could justify was a great learning experience and truly forced us to understand what was going on at a deep level.

When creating my initial reconstruction, I originally did so in Python for ease-of-debugging purposes. I used MatPlotLib to create visualizations of the height maps and terrains that I generated. Once I had arrived at the point where individual still images were being rendered by (sequentially) shooting out rays corresponding to individual pixels, I ported the code to GLSL. After translating the code and making a few minor modifications, my terrain appeared. I was able to navigate through it and confirm that a variated and continuous terrain had, in fact, been created.

4 Starter Code

As I began my development in GLSL, in order to reduce the complexity of my starting code base, I based my initial work on a small C++ program that contained a vertex shader and fragment shader inlined into the code [9]. The program was originally a demo for a VR headset that displayed a fractal-like object. With the exception of the shader-input variables, the contents of the fragment shader were removed before I began my development. The C++ program also contained functionality for handling keystrokes and injecting data from the C++ program into the shaders.

5 Terrain Variation

In order to generate its terrain, the shader that I studied and broke down utilized a grid-like concept in conjunction with its hash function in order to generate the terrain. Essentially, at each intersection point on the grid, a hash value was obtained, and if the exact point for which the program requested a height value was not exactly one of the these grid intersection points, a height value was obtained for that point by an interpolation of the height values at the surrounding grid intersection points. This helped give a nice flowing continuity across the entirety of the terrain.

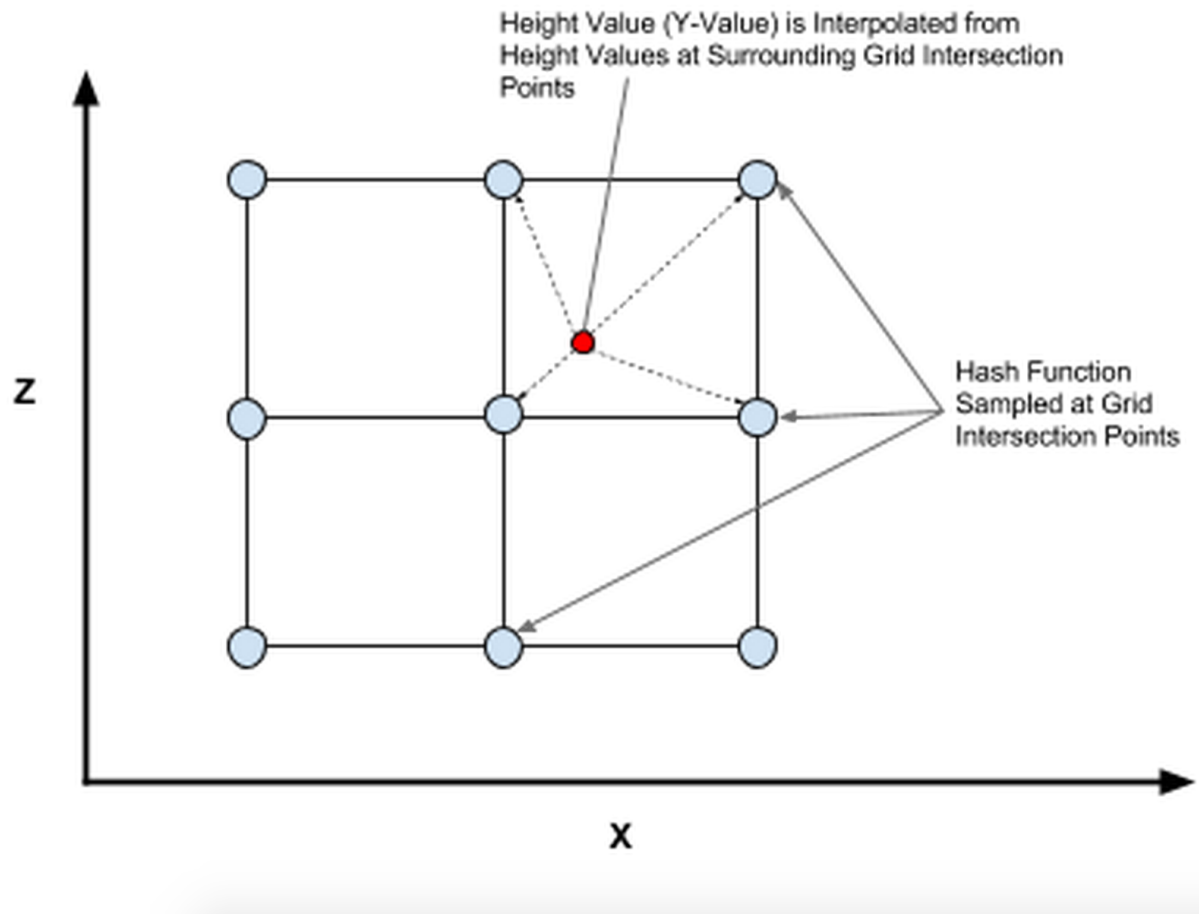

Figure 1: Diagram of the grid-like structure I used to generate my height map

I used this same concept, and after significant effort in trying to design my own interpolation scheme, I ended up using a similar process. In my shader, I was able to achieve a high degree of bumpiness in the terrain by re-applying this same grid-based process of terrain generation at successively smaller grid sizes and summing these individual contributions. In this way, the general height of a spot on the terrain was determined by the application of the hash function using the first and largest grid size, and then local variation was obtained by making use of smaller grid sizes whose weighting was also decreased (so that they affected the overall height less).

6 Basic Surface Features

In order to give my mountainous terrain a more realistic appearance than their smooth, uniformly colored initial one, I made use of a number of concepts that have been discussed in class.

6.1 Shadows

I implemented shadows by, after finding the intersection point of the original ray with the terrain, tracing an additional shadow ray from the intersection location to a directional light (at an assumed infinite distance, like in the ray tracing project). If an obstruction was found in the shadow ray's path, the magnitude of the pixel's color was reduced.

6.2 Diffuse and Specular Contributions

Using the Phong Shading Model, I also calculated the diffuse and specular contributions at each point of intersection. The diffuse contribution gave the terrain much of its rocky appearance.

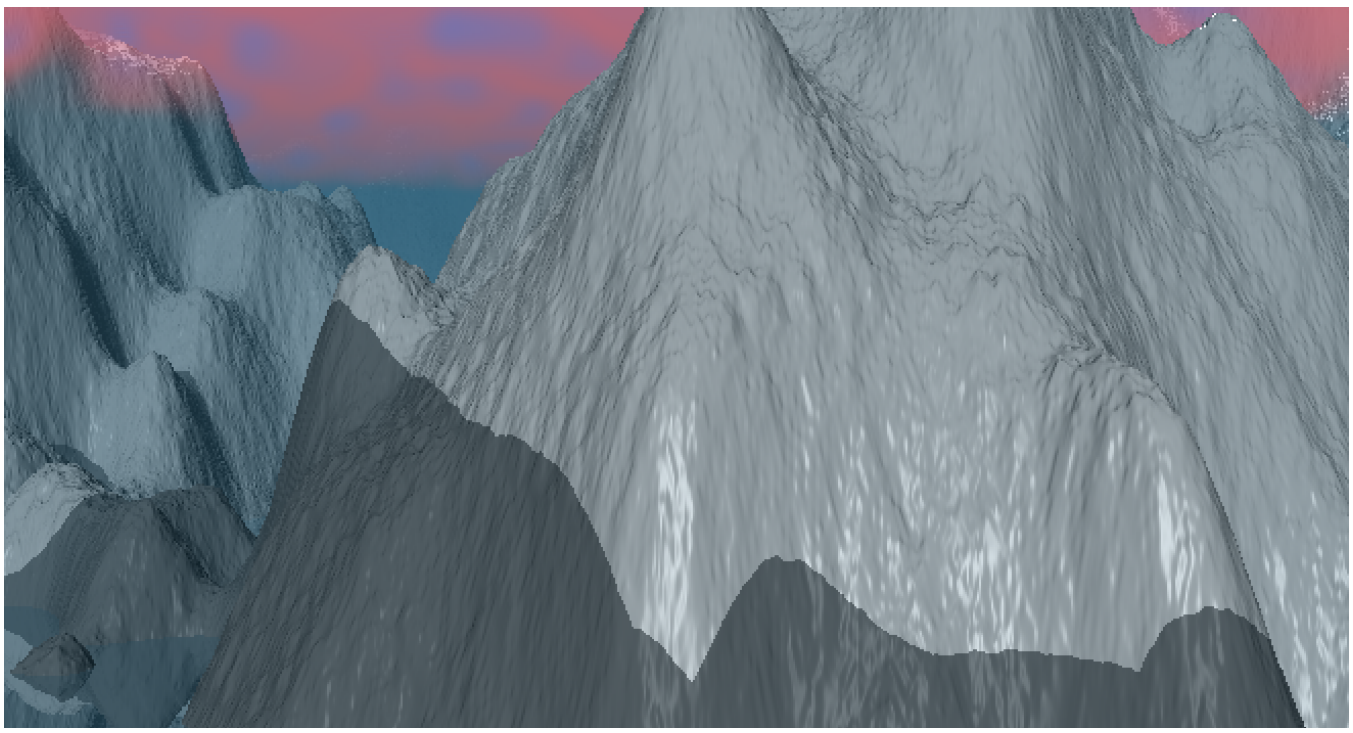

Figure 2: Demonstration of my specular, diffuse, and shadow lighting effects.

6.3 Water

To enable water in my environment, I first decided on a height that I wanted my water to exist at. Throughout my environment, if the terrain drops below this preset height, water is automatically placed at this location. To detect if I hit water, I simply looked at the Y-component of my intersection location. If the Y-component was less than the water height, I know that the ray went through water. I then find the point at which the ray intersected the water surface by using simple division involving the ray's direction, ray's starting location, and the water height.

Using the point that the ray hits the water as the intersection point, I then experimented with different amounts of reflection, diffuse, and specular lighting weightings to get an appearance I felt looked realistic and fitting for my environment.

After I got a lighting that I was happy with, I decided I wanted to model waves and surface perturbations on the water to create a more realistic, time-varying look. To do this, I injected a running time variable into the shader that is incremented at each glutMainLoop cycle. Taking motivation from the bumpy effect I implemented in the previous shader project for this class, I decided that perturbing the surface normals used for my lighting and reflection effects might give the appearance of waves and ripples. I then modified a (float,float)->float sinusoidally based hash function that I found at [6] to give perturbations with local correlations and periodic behavior. I fed this sinusoidal function the (X,Z) location of the water intersection multiplied with the time variable I injected in order to yield perturbations which varied smoothly over the spatial and time domains.

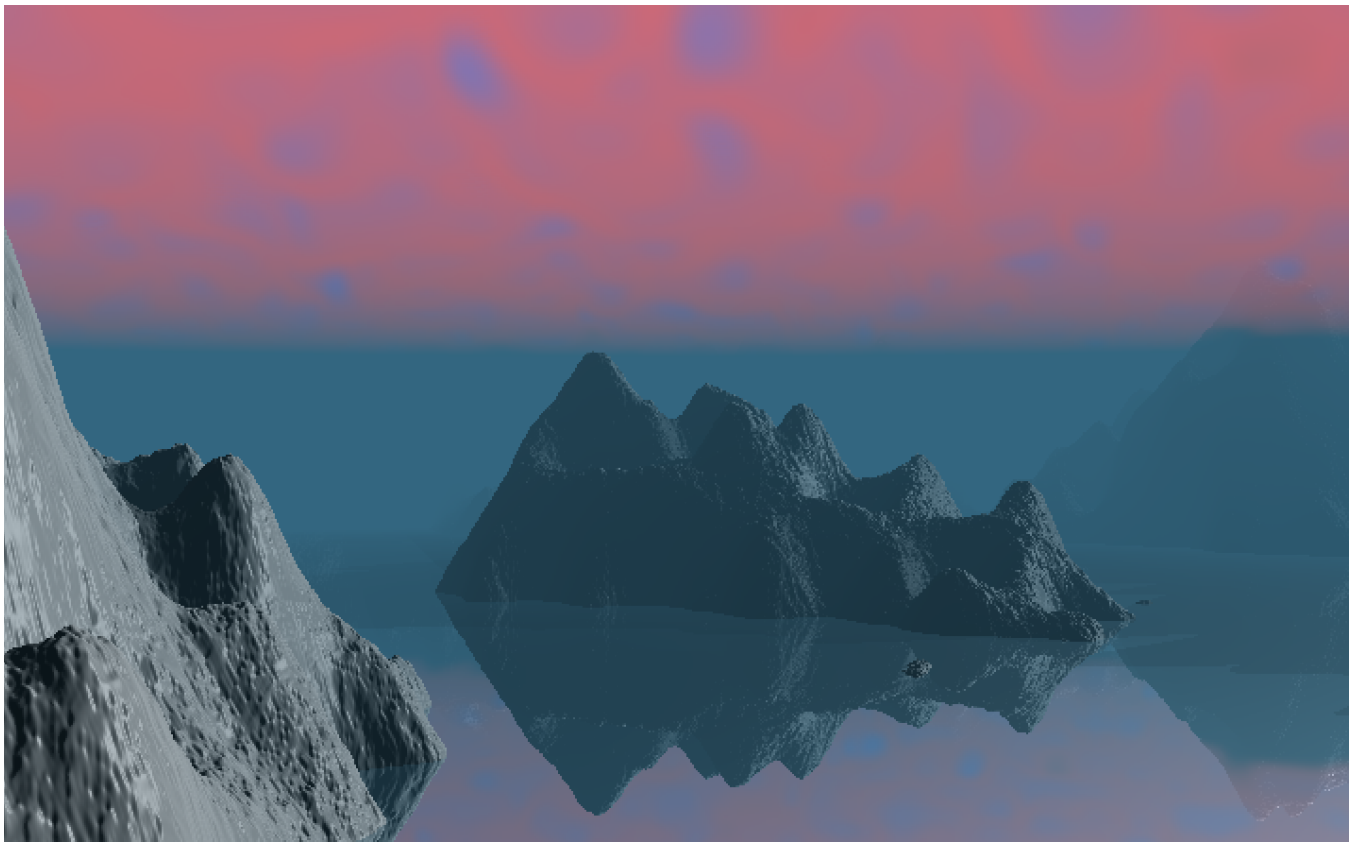

Figure 3: Illustration of my water (without the rippling effect). Notice the reflection, specular, and shadow effects.

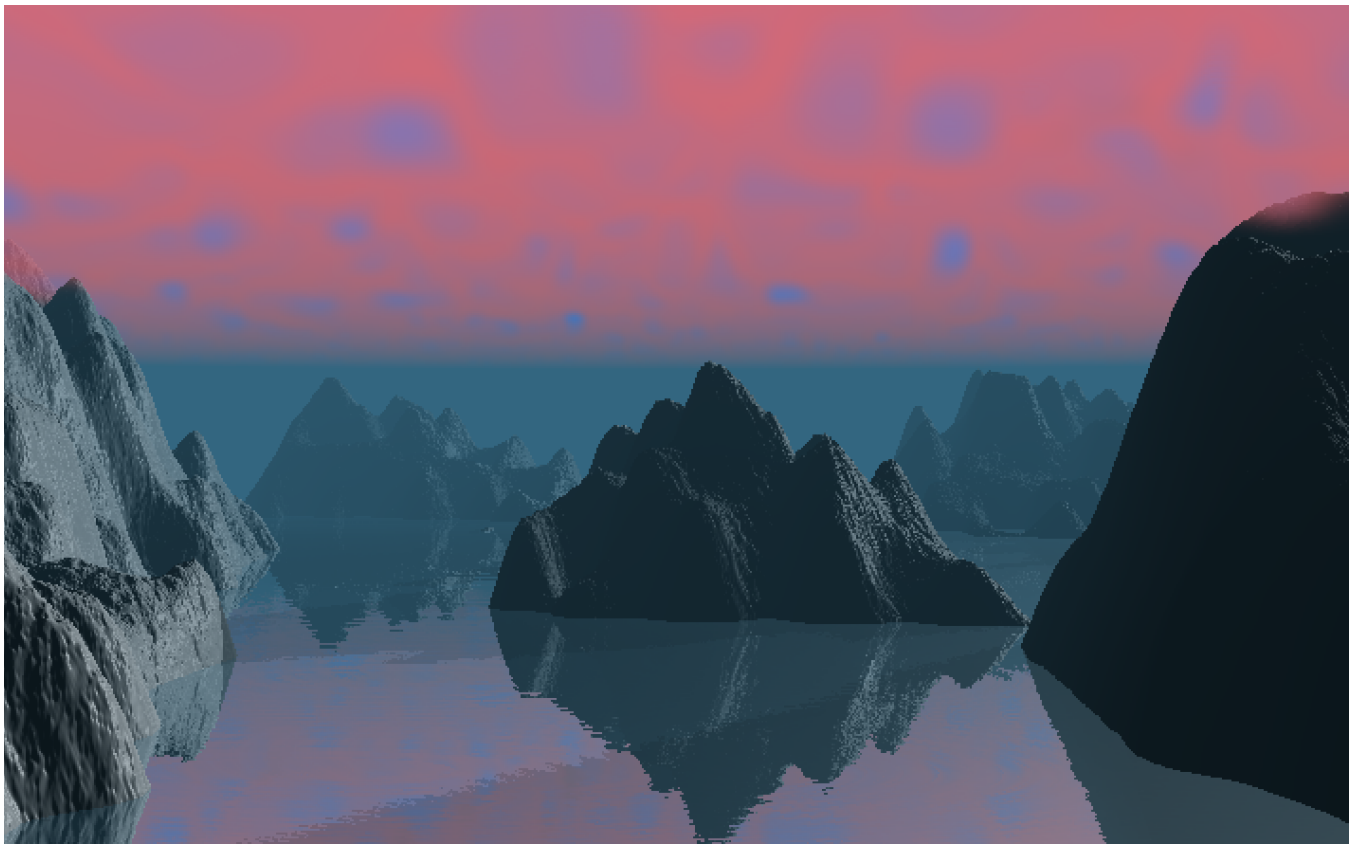

Figure 4: Illustration of my water with the rippling effect. Notice how the sinusoidal hash function gives the appearance of small waves on the surface of the water.

6.4 Clouds

To implement clouds in my environment, I took motivation from [5]. I dissected this shader code, and in doing so came to understand the technique of ray marching that was employed. Essentially, as I shoot my ray through the environment, I sample a "cloud density" value at each step of the ray's progression. I then deterministically calculate the cloud color and opacity using this density value, and I mix the cloud color with the environment behind the cloud using the opacity.

To calculate the cloud density, I utilize a technique I learned from the previously mentioned shader code. This technique is in some ways a 3-dimensional version of the 2-dimensional hash-interpolation technique I used to generate my environment height map. To generate my environment height map, I smoothly interpolated a (float, float)->float hash function to give the height at each (X,Z) location. To generate my clouds, I used a (float, float, float)->float hash function to interpolate in 3 dimensions the density of the cloud at each (X,Y,Z) coordinate point in my world. This hash function was obtained from [8].

Because I wanted my clouds to only appear within a certain region in my environment, I linearly scaled the density values returned from the hash so that they are zero at the bounds of my cloud region and at a maximum in the middle of my cloud region. To make my clouds move with time, I multiplied the (Y,Z) coordinates of the point fed to the hash function by the time variable (mentioned previously in the section on water). I scaled the Z-component the most by time in order to give the clouds the appearance of moving mostly horizontally.

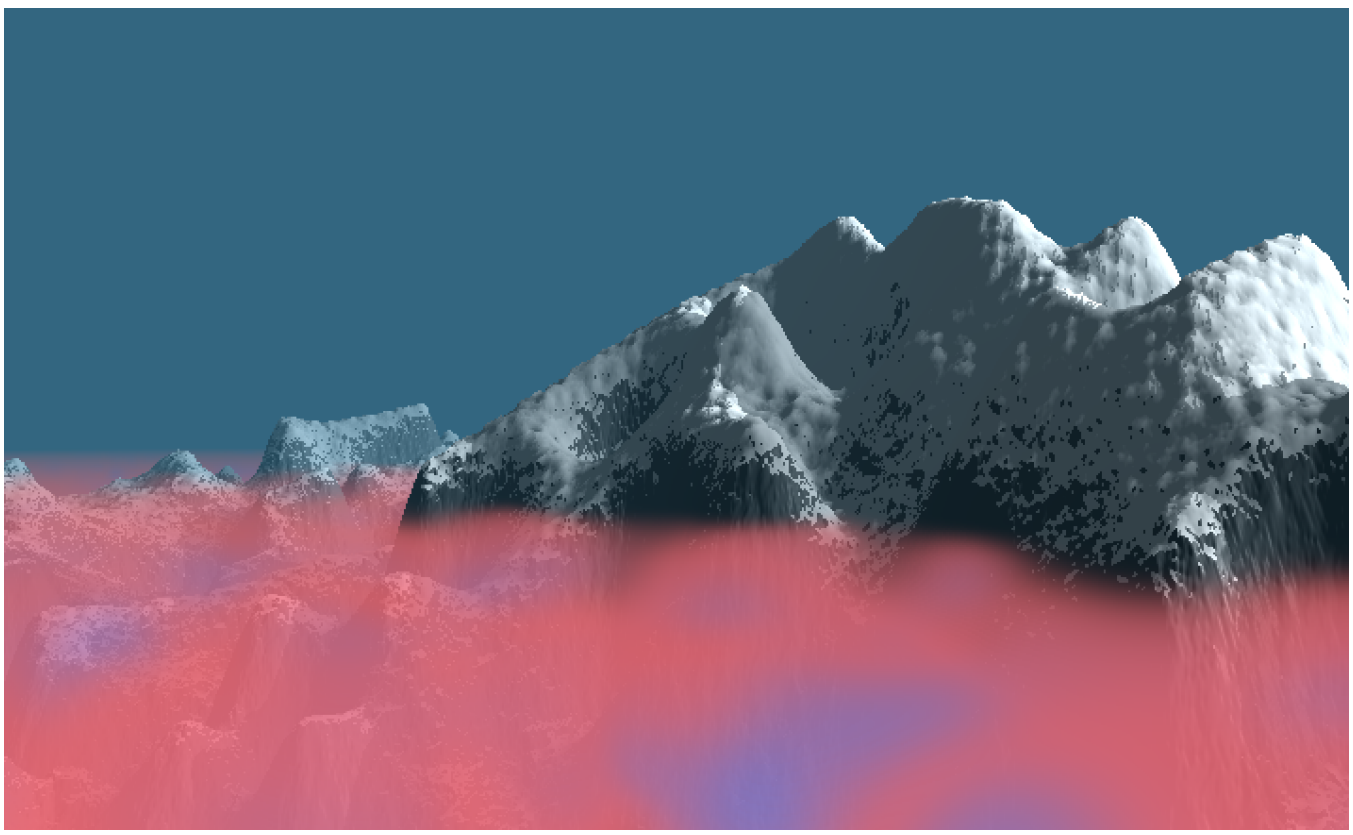

Figure 5: Screen shot of my clouds when viewed from above.

7 Additional Features

As I worked to extend my shader, a number of additional features were added as bells and whistles.

7.1 Fog

One initial problem that I had with my terrain was that because rays were only being cast out to a finite distance, terrain that was farther away would begin to magically appear as the user moved forward. A fog effect that I added

had the pleasant side-effect of eliminating this problem. After doing some basic research for how to implement fog, I found a posting [3] that explained at a high level the general approach that I took for introducing my fog. This approach uses the distance from the camera to the intersection point to find the factor that should be used for linearly interpolating between the color of the fog and the color that would otherwise be displayed for the terrain at that location. The sky was also set up to be considered infinitely far away. In this way, the sky took on the color of the fog, and therefore a smooth transition was achieved between distant terrain and the sky.

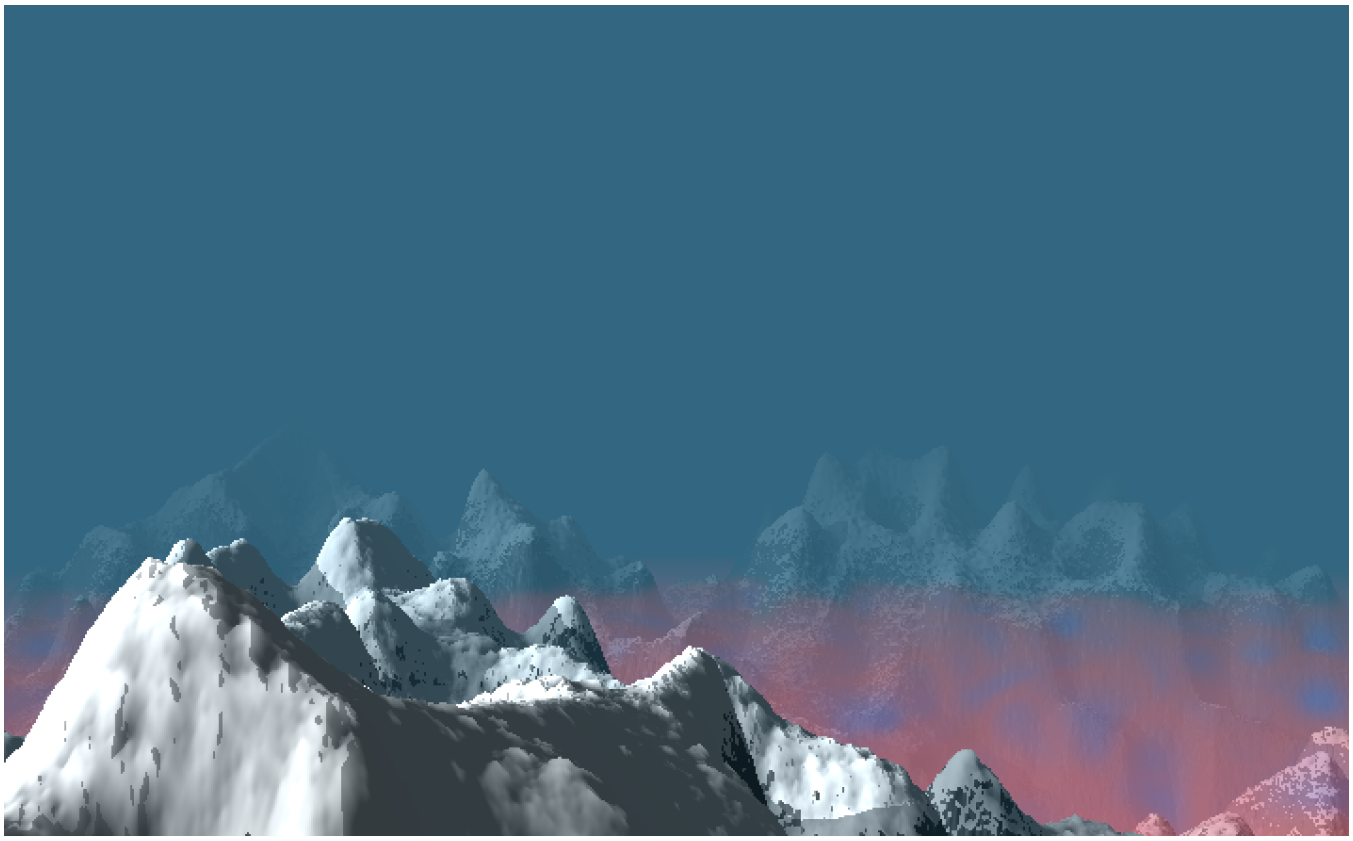

Figure 6: Demonstration of the fog effect. Notice how the color of the mountains off in the horizon is blended with the fog color, giving the appearance of a thick mist.

7.2 Snow

I added an option to create snow toward the summits of my mountainous terrain. The decision was made as to where to place snow by looking at a combination of a given intersection location's Y-value and the closeness of the normal at the location to being perfectly vertical. The intuition behind using the normal in the calculation is that flatter areas (which have normals closer to vertical) are more likely to retain snow, rather than having it slide off. One initial problem that I experienced was that, while it was colored white, the snow-covered areas retained the exact same texture as the dry areas. This gave the snow the appearance of being "painted-on" rather than of actually being a separate layer sitting on top of the mountains. While this effect could still user further work, this negative effect was lessened by having snow-covered spots calculate an updated normal at their locations using neighboring locations that are further away. This has the desired effect of making the snow-covered areas have less definite texture.

7.3 Sun

Early on, I set up a shadow effect as well as diffuse and specular contributions for the coloring of my terrain. All of these supposed a hypothetical sun object from which this light was emanating; however, the sky was completely uniform. In order to contribute an additional sense of realism, I placed a sun in the sky. As the user changes the lighting direction through keyboard input, he or she can also watch the sun move through the sky. This effect is achieved by comparing the direction of the ray leaving the camera with the direction to the light (which is constant, since the light is assumed to be infinitely far away). If a given ray is determined to hit the sky, the angle between this ray's direction vector and the direction vector to the light is then calculated. If this angle is below a certain

threshold, the color to be outputted for this location in the sky is chosen to be an interpolation between the fog/sky color and white; smaller angles correspond to a color closer to white. One important modification that was made was to use the smoothstep function to attenuate the whiteness of the sun as the distance from the center of the sun increases. This causes the sun to have a very bright center and also a large "halo" effect that gradually fades to the fog color. Finally, an additional modification was made so that as the fog density in the scene increased, the intensity of the sun would decrease.

Figure 7: Illustration of the sun, notice the blending with the sky and smooth attenuation of the lighting.

7.4 Moon Mode

The use of the hash function to generate the terrain made the height characteristics of the terrain relatively easy to modify. This was demonstrated in the "moon mode" that I created in addition to my standard mountainous terrain. As has been previously mentioned, my bumpy terrain is generated by successive applications of the hash function using different values for the grid size and different weightings for each pass. A moon-like terrain was created by modifying these grid parameters and modifying the hash function in two ways. First, the hash function was changed so that it outputted negative values instead of positive values. This had the effect of creating valleys and craters, rather than hills and mountains. Second, height values of relatively small absolute value were scaled down to have an even smaller value. This had the effect of making the surface flatter than it had been for the mountainous terrain.

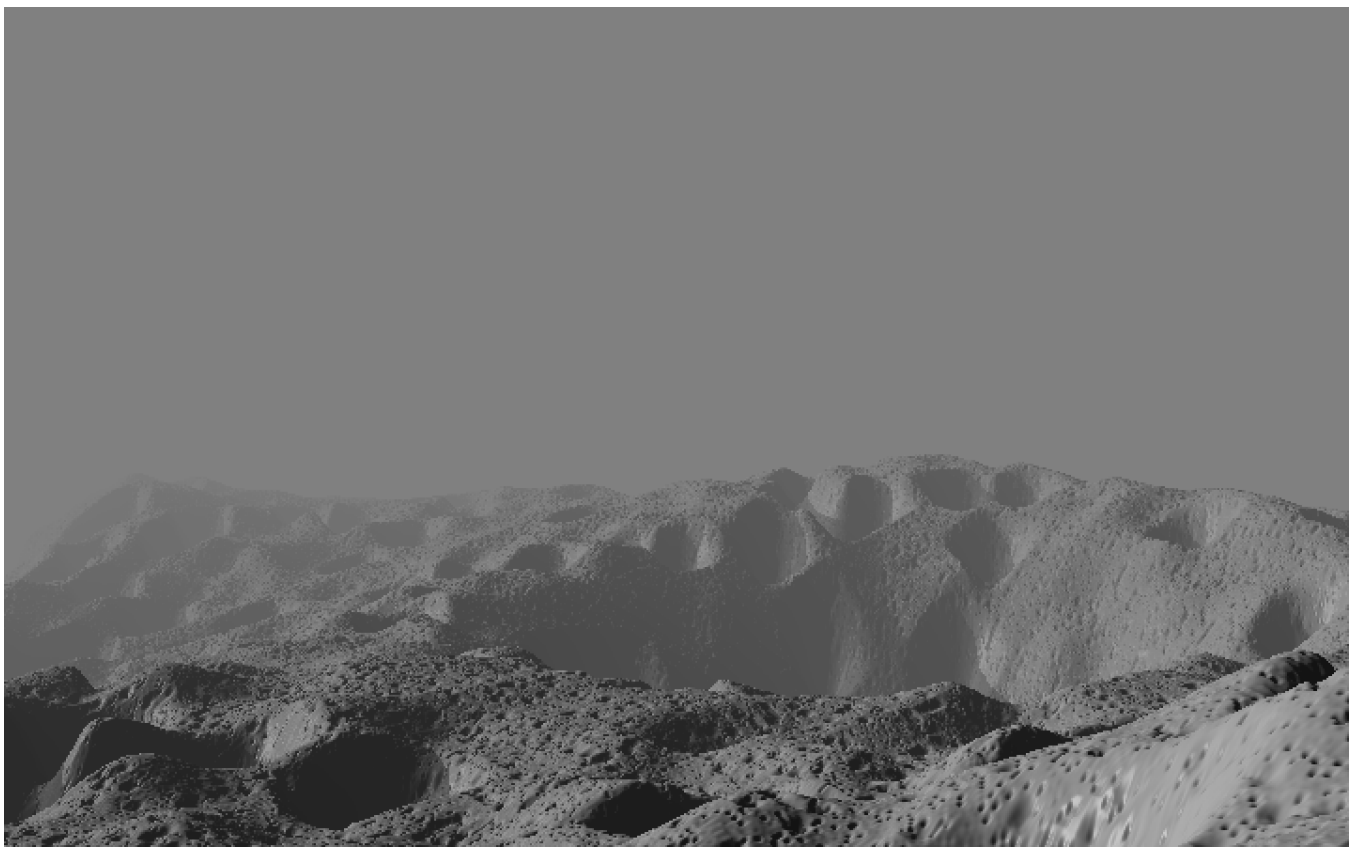

Figure 8: Check out my cool moon terrain.

7.5 Jumping

In order to increase the user's sense of immersion in the world, I also gave the camera the ability to jump in the air momentarily. This was done by making use of a simple kinematics equation for one-dimensional motion in conjunction with the time variable that was being injected into the shader from the C++ program.

7.6 Anti-aliasing

I implemented real time anti-aliasing for my environment by supersampling. The number of rays per pixel is specified as a global variable in my shader code. The aliasing does work in the sense that it generates smoother images and removes pixelation, but it is generally too slow (at least on my computers) to be used pleasantly. With 4 rays per pixel, you can still navigate around the environment, but it is not up to par with what I were hoping. Perhaps on a better computer, my anti-aliasing could be workable. Below are images showing the effect of my anti-aliasing.

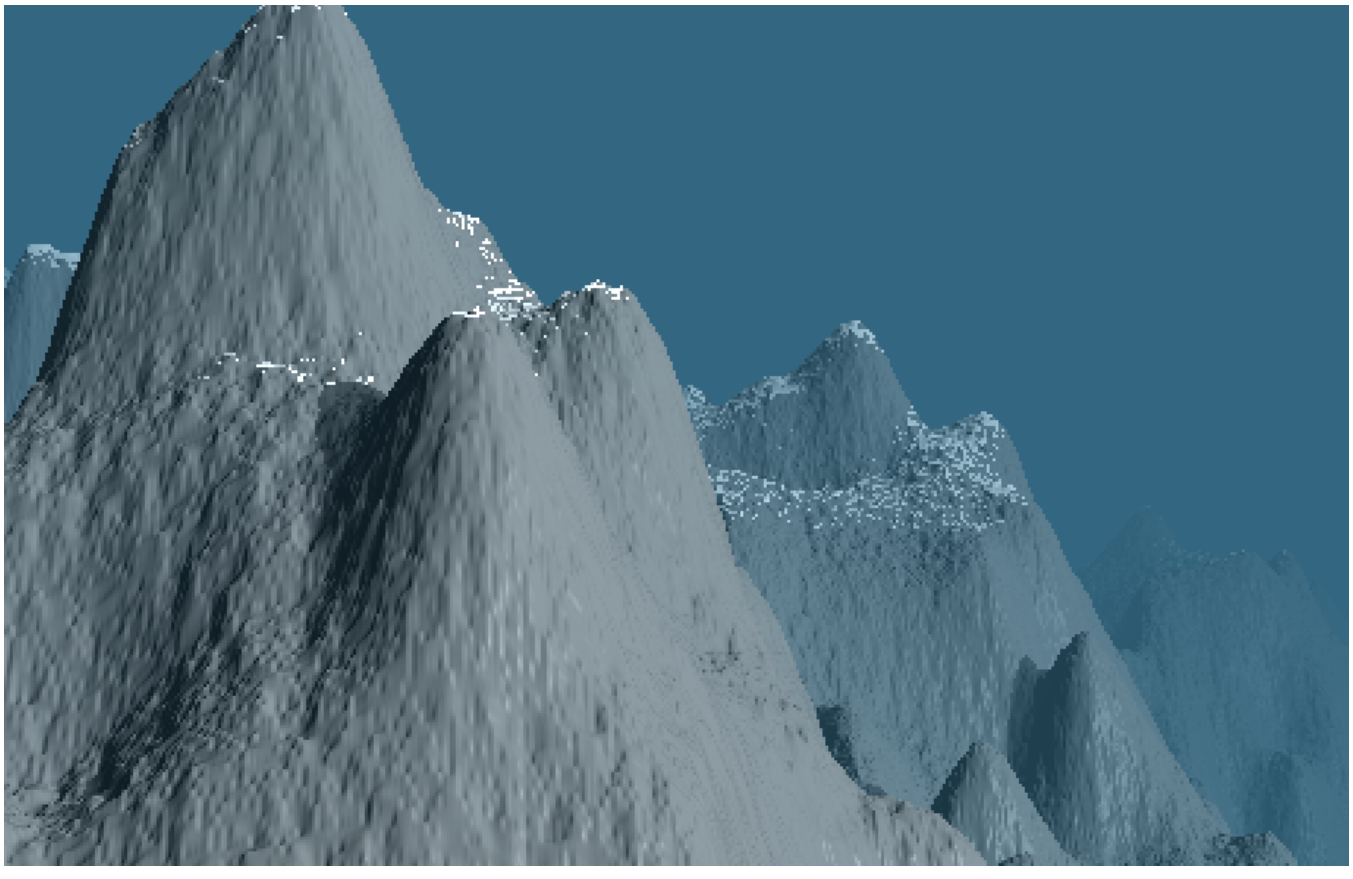

Figure 9: This is an image of my terrain without anti-aliasing. Notice the slightly jagged edges and the pixelated snow on the cliff in the distance.

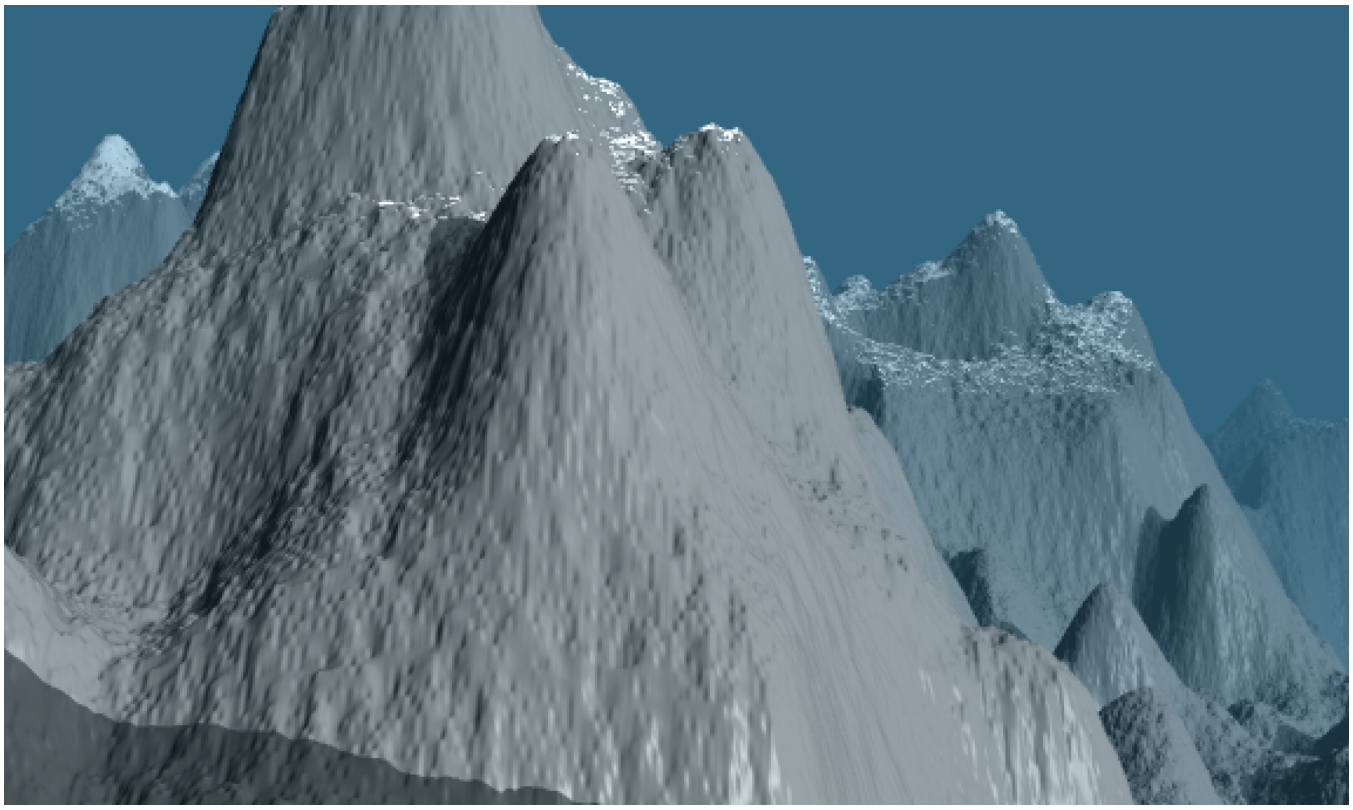

Figure 10: Applying a supersampling of fmy rays per pixel eliminates the artifacts seen in the above image.

References

[1] Alain Fmynier, Don Fussell, Loren Carpenter. “Computer rendering of stochastic models”. 1982. Communications of the ACM.

[2] David Hoskins. “Mountains” Fragment Shader (with help of Inigo Quilez on noise generation). 2013. https://www.shadertoy.com/view/4slGD4

[3] Brendan Kenny. Explanation of Creating a Fog Effect. Derived from OpenGL ES 2.0 Programming Guide. pg. 224. http://stackoverflow.com/questions/11694458/how-to-create-fog-using-open-gl-es-2-0-or-webgl

[4] Paul Martz. Generating Random Fractal Terrain. www.gameprogrammer.com/fractal.html

[5] Inigo Quilez. ?Clouds? Fragment Shader. 2013. https://www.shadertoy.com/view/XslGRr

[6] User appas. Provided a Two-Dimensional Hash Function. http://stackoverflow.com/questions/4200224/random-noise-functions-for-glsl

[7] User DMGregory. Overview of Ray-Marching. http://gamedev.stackexchange.com/questions/67719/how-do-raymarch-shaders-work

[8] User ufomorace. Provided a Three-Dimensional Hash Function. Originally obtained from the work of Inigo Quilez in the ?Clouds? Fragment Shader (cited also). http://stackoverflow.com/questions/4200224/random-noise-functions-for-glsl

[9] Evan Wallace. C++ Program with Embedded Shaders for Fractal Display on VR Headset. https://gist.github.com/evanw/7406147