# Procedurally Generated Terrains

Yanni Dahmani, Sebastian Le Bras

August 12, 2016

## Part 1: Intro

For our final project we decided to implement a procedural terrain generator. We we're, like other groups, influenced by the recent video game by Hello Games called "No Man's Sky". The idea of generating ever different and realistic terrains fascinated us, and we believed it would be a great way to pool together our newly acquired CS148 knowledge and skills. Here we present our findings and our methods of rendering these procedural and realistic mountains. All following described work was done in tandem with one person writing code and the other thinking and researching. We took turns in the positions.

## Part 2: Relation to Graphics

Procedural Terrain Generation presents a few challenges. The first and perhaps most pivotal to the creation of the generator is providing an algorithm that creates new random believable terrains. After some research into different methods, we decided that to accomplish this feat we would use the Diamond-Square Algorithm. The Diamond Square algorithm presented enough of a technical challenge to be interesting but was also straight-forward enough to accomplish within the allotted time and allow room for other improvements.

Other technical challenges that the project presented included Shading (for which we implemented Phong Shading), Shadow Mapping, Texturing and Materials, Skybox implementation, Laplace Smoothing, and multi-pass rendering (used in the shadow mapping methods). OpenGL 3.3 was the main programming environment used. Multiple shaders we're used for different object rendering, and many of the algorithm's used (such as phong shading and laplace smoothing) we're taught in class, yet we created our own implementations.

## Part 3: Diamond Square Algorithm

The Diamond Square algorithm is a method for generating random height maps first introduced by Fournier, Fuddell, and Carpenter at SIGGRAPH 1982[1].The Diamond-Square Algorithm requires the use of a $(2^x + 1)$ by $(2^x + 1)$ grid where $x$ is the number of recursive steps we will apply. This is due the midpoint nature of Diamond Square, i.e. at every level of recursion the width/height of the grid.

The Diamond Square algorithm overall can be broken into two main phases with the second phase containing two important steps. For this description, without loss of generality, we will take a small yet nontrivial example where we apply Diamond Square on a $5 \times 5$ grid.

The first phase is the initialization phase. As can be seen in figure 1, we initialize the four corners of our $5 \times 5$ grid to some height value. We can do this in two ways: either we set all four corners to a predetermined height value (i.e. 0) or we set each corner to a randomized height value. This choice depends on how random we want the resulting terrain. If a more peak and valley terrain is desired, setting the four corners to 0 works well (or some other constant). If something more random is desired, then randomly setting each corner is better.

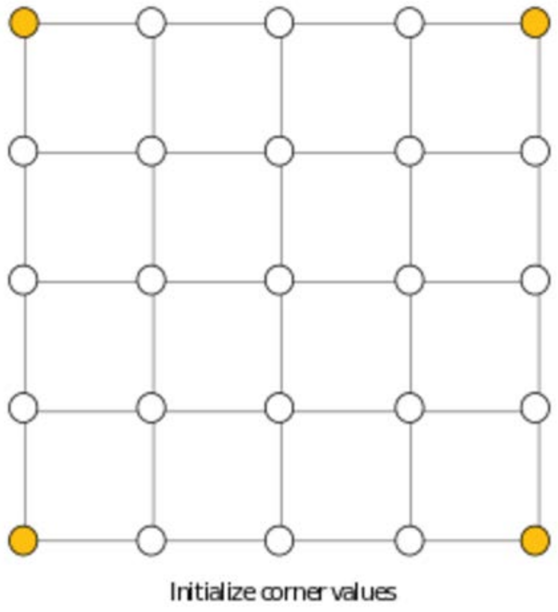

Figure 1: Initialization phase of Diamond Square

The next phase, the recursive phase, is the actual heart of the Diamond Square algorithm. The recursive phase can be broken down into two steps: the Diamond Step and the Square Step. Each of these steps are named for the shapes they create.

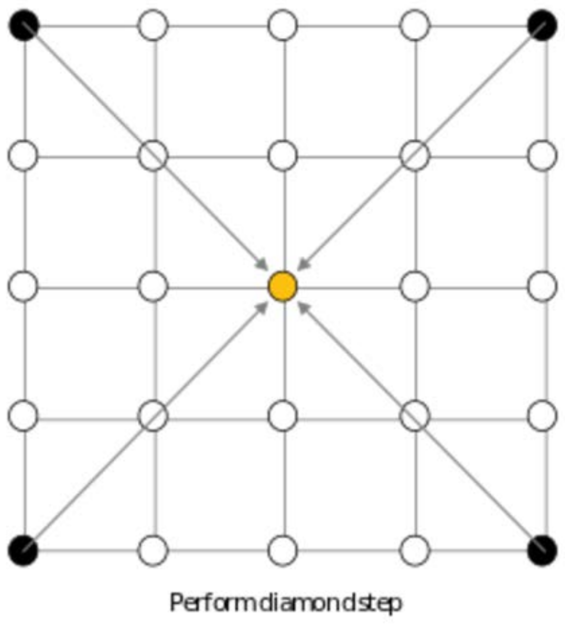

Figure 2: Diamond step of Diamond Square

As can be seen in figure 2, the diamond step consists of finding the midpoint of our square and setting it to the average of our corners plus or minus some random offset predetermined by us (more on the offset later). This diamond step now creates four diamonds in our grid. If we take our simple $5 \times 5$ grid again, we take our four corners, average them, and then set the yellow midpoint of our square to the average plus some offset.

Following the diamond step comes the square step. As can be seen in figure 2, our diamond step has created some number of diamonds (in the case of our $5 \times 5$ grid four diamonds with each diamonds fourth vertex appearing off the grid). From here we find the midpoints to these diamonds and set them to the average of the diamond vertices plus or minus some offset. As can be seen in 3, our four diamonds have now been turned into four squares who's new yellow vertices derive their height values from the average of the diamond vertices.

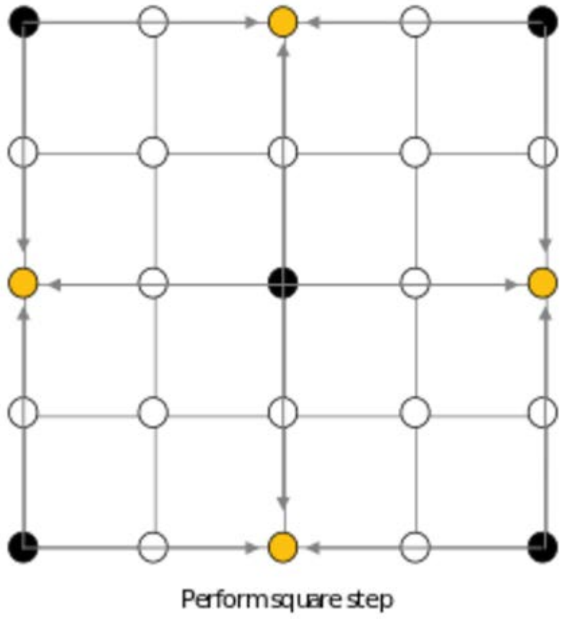

Figure 3: Square Step of Diamond Square

We have now went through one recursion of the Diamond Square algorithm. We now recurse on our $5x5$ grid and complete another round of diamond step and square step. We now have four squares, so we now find the midpoints of each of those squares and set them to the average of their corners plus or minus some offset. This can be seen in Figure 4 where we have created twelve new diamonds out of assigning for new yellow vertices. We then apply a square step on each of these newly created diamonds. For our simple $5 \times 5$ grid this will be our final step. As can be seen in figure 5, the 12 diamonds' midpoints have now been filled with new yellow vertices, thus filling out our entire $5 \times 5$ grid. If our grid we're to be larger, i.e. a width and height of $2^{10} + 1$ or 1025, there would be ten recursive steps total as opposed to the two from our $2^2 + 1$ grid.

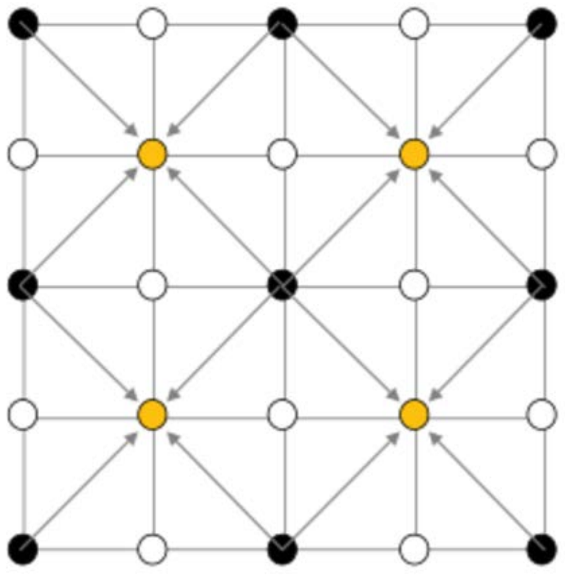

Figure 4: Second Recursive Diamond Step of Diamond Square

We now need to describe how we determine the offset to apply to each new midpoint vertex for the diamond and square steps. We need a range that a new height offset can be chosen from such that the new vertex can have a height anywhere from below to above the average of the diamond/square vertices. To do this we give a range value $r$ into our algorithm such that at every instance of diamond/square step the algorithm chooses an offset from $-r$ to $r$ to add to the average. Now if this we're all that determined the heights of our terrain's we'd end up with some extremely jagged and overall drastically varying peaks in our terrain. This is caused by our range remaining constant through out each recursive step of Diamond Square. To solve this error we include a roughness factor $H$ that is multiplied onto our range $r$ such that in every following recursive step our new range $r_{new}$ becomes $r_{new} = 2^{-H} \cdot r$. For us this roughness factor $H$ ranges from 0 to 1. The smaller values of

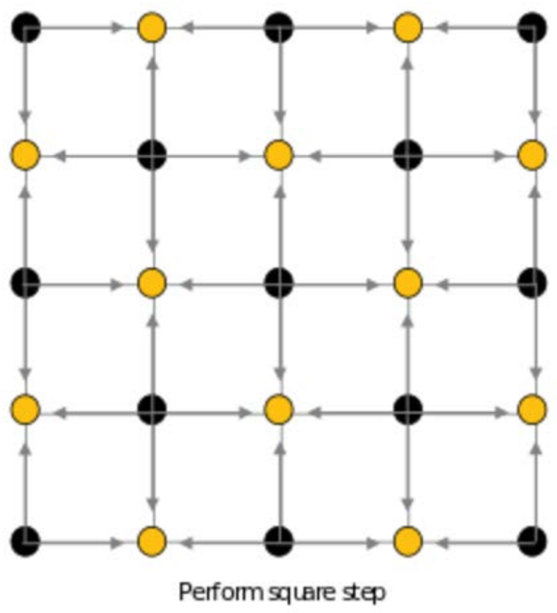

Figure 5: Second Recursive Square Step of Diamond Square

$H$ correspond to rougher and spikier terrains while the higher values of $H$ correspond to smoother and more realistic terrains. If we were to make $H$ even higher, we would end up with a terrain that looks like a rolling hill. This roughness factor essentially gives us a built in smoothing function.

This pretty much sums up our implementation of Diamond Square. In the following sections we describe our process of implementing our terrain generator as well as the difficulties experienced and extra features implemented.

## Part 4: Intermediate Steps and Results

Our very first hurdle was creating the grid. While fairly straightforward, it was nice having something appear on the screen. The initial grid was rendered with GLut in OpenGl 1.0

which was then painstakingly ported to OpenGl 3.3. Our next step was to implement the

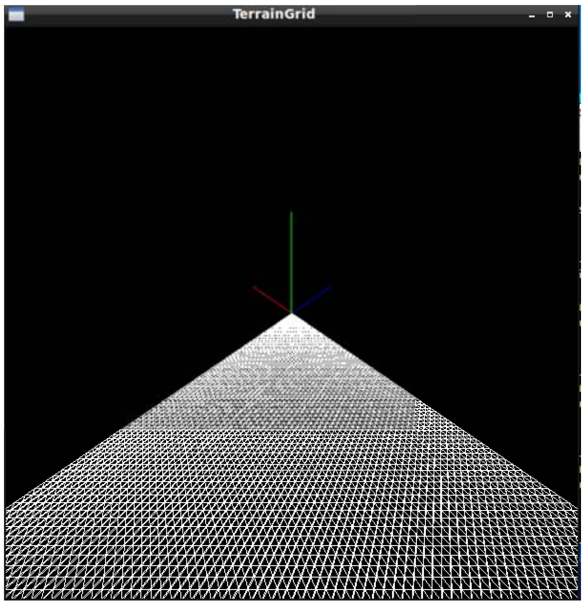

Figure 6: First Grid

Diamond Square algorithm and provide some coloring so that we could see different heights. This is shown in figure 8 and figure 7 below.

The next step was to implement a texturing based upon height. To do this we picked scaled ranges based upon the max possible height where each range corresponded to some terrain. As can be seen in figure 9 these initial textures we're correctly mapped to certain height ranges, but were the wrong textures. We would later implement three textures with grass being the lowest range, rock being the mid range, and ice being the max range. Also as can be seen in 9 phong shading was already implemented at this time.

Our Texturing of the terrain is dependent on the height(y) of each vertex. However this created a harsh stratification of the terrain which did not look appealing. We added in a transition between the textures that made the transitions softer and look more realistic.

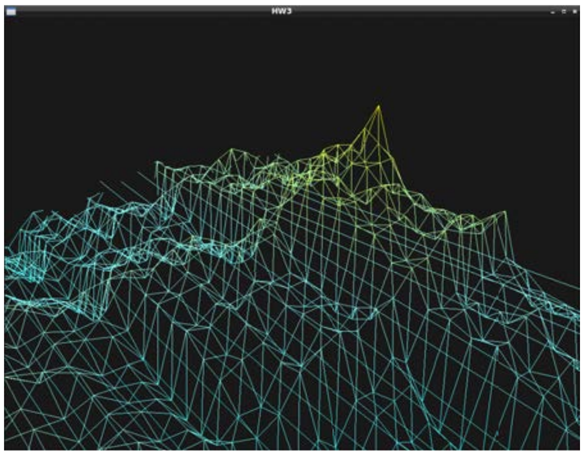

Figure 7: Height Coloration on mesh grid

In the figure 10 it is most evident in the transition between the grass texture and the rock texture.

At this point it should also be noted that we had implemented a skybox as well as fixed errors in our Diamond Square Algorithm (described in Part 5).

We also implemented shadows using a shadow map that employed a multipass rendering where it rendered the main object (our terrain), determined the shadow map values, then rerendered our terrain, shadow, and skybox. The final thing we implemented was a key control that allowed us to move the light in our scene to better show our shadow features.

This ends the discussion of our intermediate steps and results. The following section will describe the challenges we experienced during these intermediate stages implementing these features.

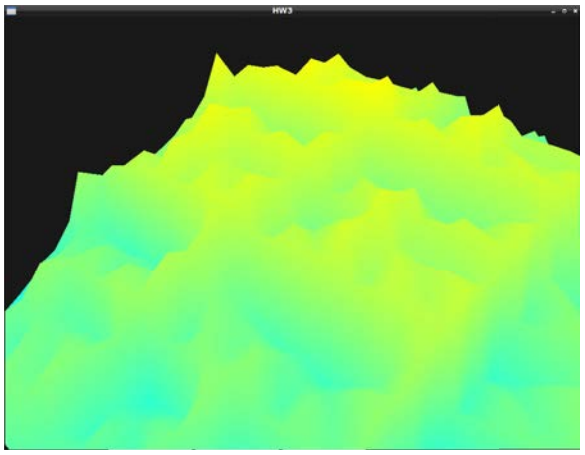

Figure 8: Height Coloration on filled grid

## Part 5: Challenges

This project presented a plethora of challenges. At the offset of the project we wanted to implement a terrain generator and introduce a water simulation into the landscape. We spent time at the offset of the project researching how to implement water generation and even came up with milestones for water generation. However, when we began to work on the terrain generation aspect of the project, we realized that we wouldn't have the time to create a satisfactory water simulator and terrain generator, so we decided to stick with the terrain generator, which was the primary objective of our project.

A surprisingly challenging aspect of the project was working with textures. Since we had worked in We had not anticipated that texturing our landscape would present a significant technical challenge, but we found ourselves stumped on several occasions. One particularly frustrating challenge was getting our textures to cooperate with the skybox we created. We were running into an issue where the skybox and the terrain would generate,

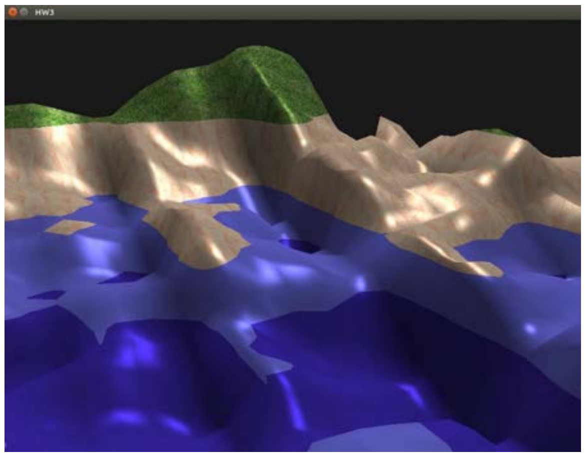

Figure 9: Initial texture

but the skybox was the only thing properly textured. The terrain was only able to use one of the textures that we passed into the terrain shader when the skybox was rendered, which made our terrain look homogeneous and quite boring. However, when the skybox was not rendered the texturing appeared as desired. We eventually found that this issue was with order of calls to "shaderName.Use()" for the terrain shader and the skybox shader, but not until after several disheartening hours.

Another issue with textures was finding an appropriate indexing for them. We found that when we indexed a texture to a square created by two triangles, we were getting very patterned results. This made our terrain look very "un-terrainy". We played around a while with different indexing and wrapping options until we finally found a setting that made our terrain look as realistic as possible.

When we started rendering our scene on a larger grid, we realized that our terrains we're making very noticeable fractal patterns. Upon further inspection we realized that our diamond-square algorithm wasn't making the correct recursive calls and was populating small

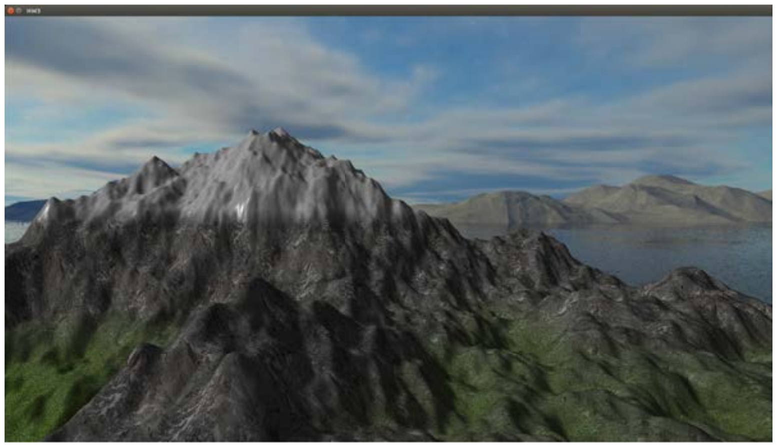

Figure 10: Texture Fading

sections to completion rather than recursing by passing in half the width. We redesigned the algorithm which took some time but we ended up happy with our results.

Indexing the GL_TRIANGLE_STRIP that rendered the terrain was an interesting challenge as well. We initially wanted to create the grid using GL_TRIANGLES but we found that we would have to store an unnecessary amount of index information, so we opted to use GL_TRIANGLE_STRIP instead. However, we then came across the issue of how we would properly index the GL_TRIANGLE_STRIP. We tried to come up with a solution but we found that we were not familiar enough with how GL_TRIANGLE_STRIPs behave, so we referenced a website to come up with the indexing[4].

## Part 6: Final Results & References

We referenced the following sources:

[1] https://design.osu.edu/carlson/history/PDFs/p371-fournier.pdf

[2] http://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping

[3] http://learnopengl.com/#!Advanced-OpenGL/Cubemaps

[4] http://www.learnopengles.com/android-lesson-eight-an-introduction-to-index-buffer-objects-ibos/

[5] http://www.gameprogrammer.com/fractal.html

We will submit a youtube video that demonstrates our project's capabilities live.

Below we have also posted several pictures that illustrate the project as well with captions that explain what each picture features. Note all generated terrains are implemented on $1025 \times 1025$ grids with default values of 16 iterations of Laplacian Smoothing, non randomized initial corners, range offsets of 10, roughness factor of 1 unless otherwise specified. Note also that the terrains vary significantly from each other. This is a direct result of our implementation:

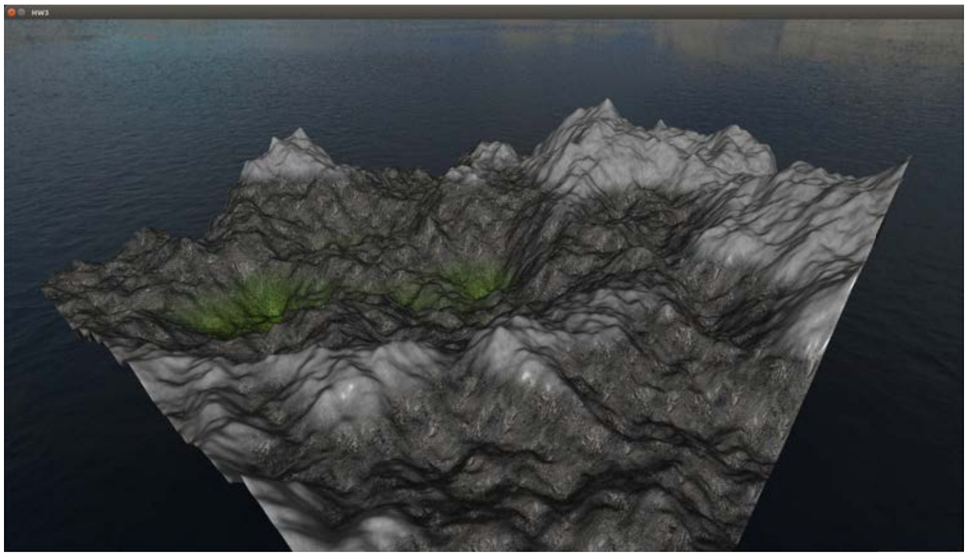

Figure 11: Shadows on Terrain

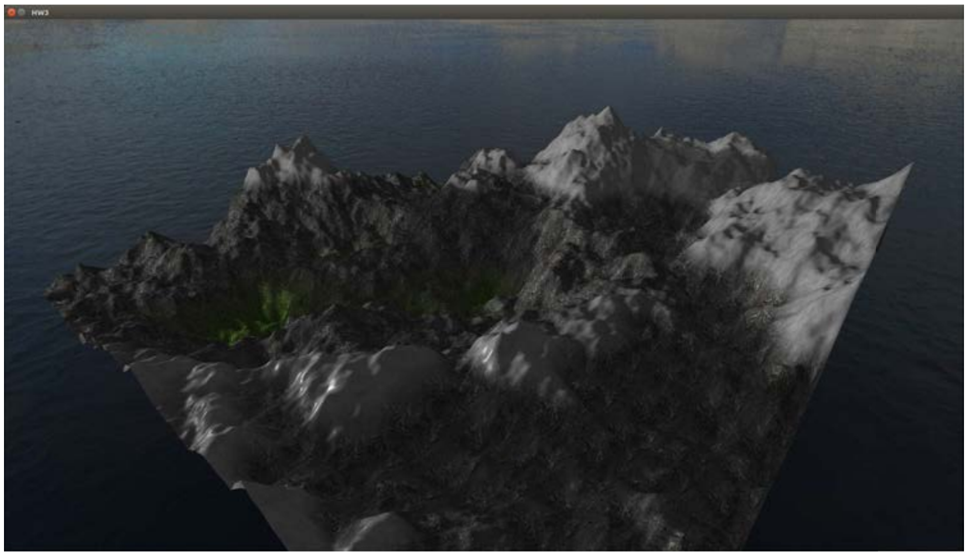

Figure 12: Shadows on Terrain

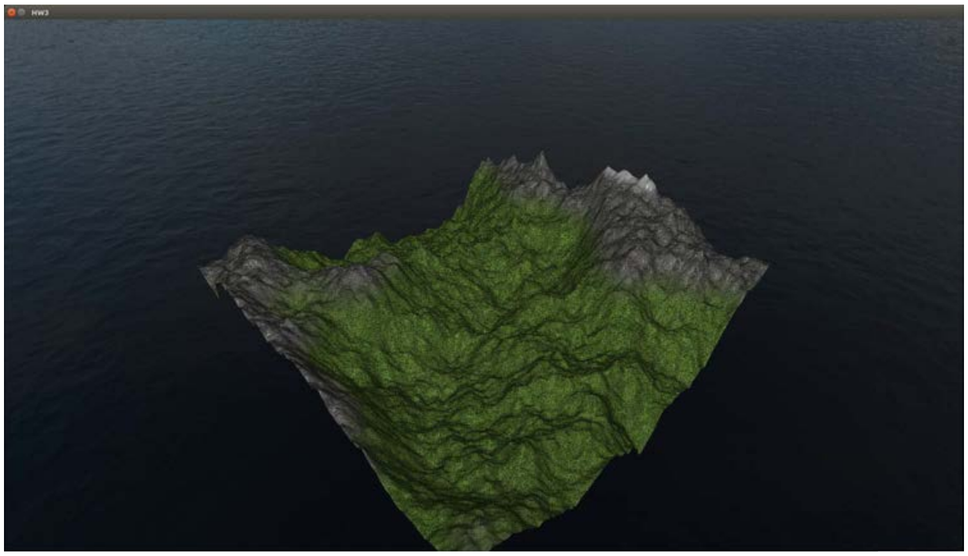

Figure 13: Different Light Placement

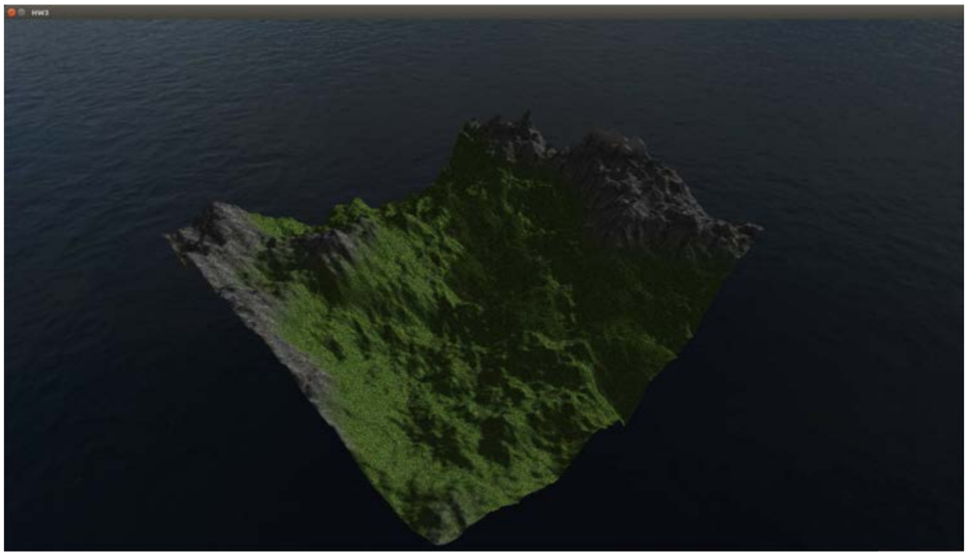

Figure 14: Different Light Placement

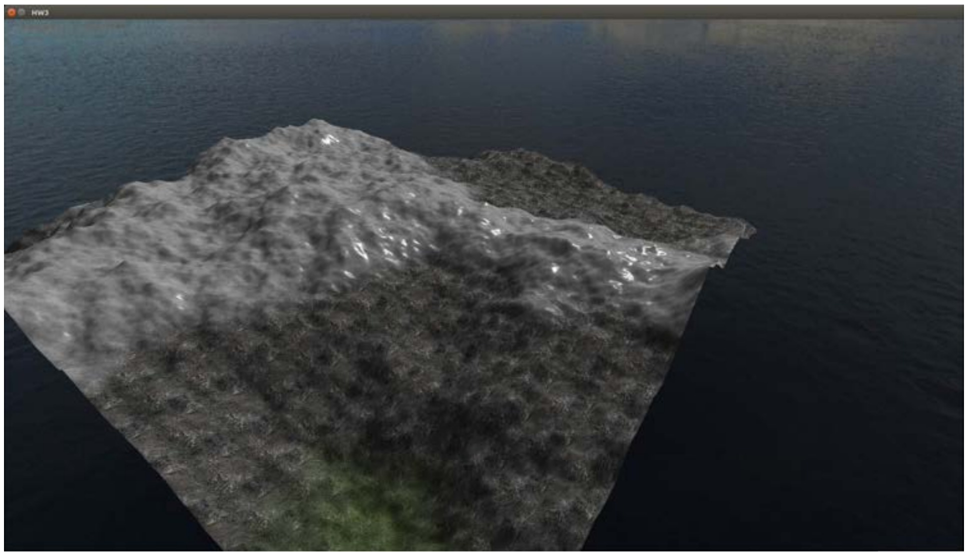

Figure 15: 5 Range Value

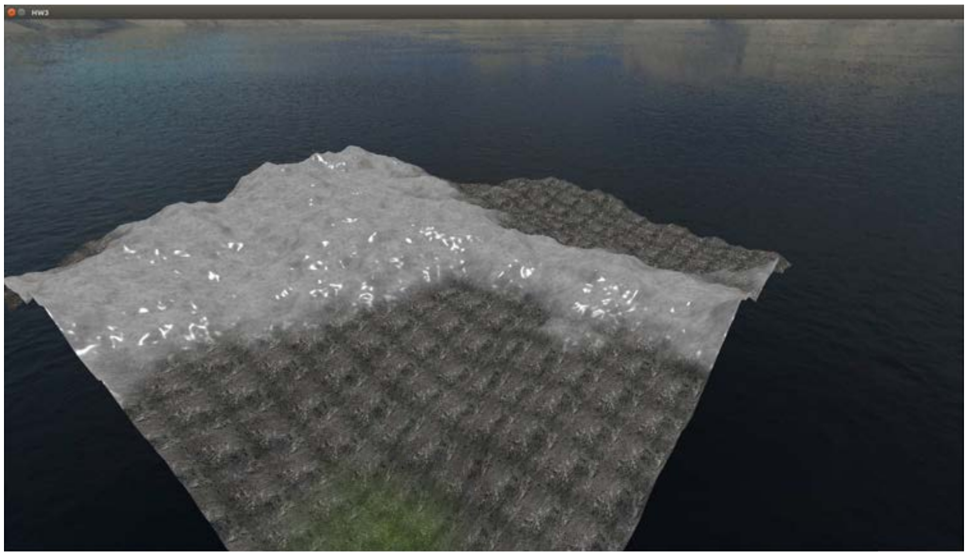

Figure 16: 5 Range Value and Ice Specularity

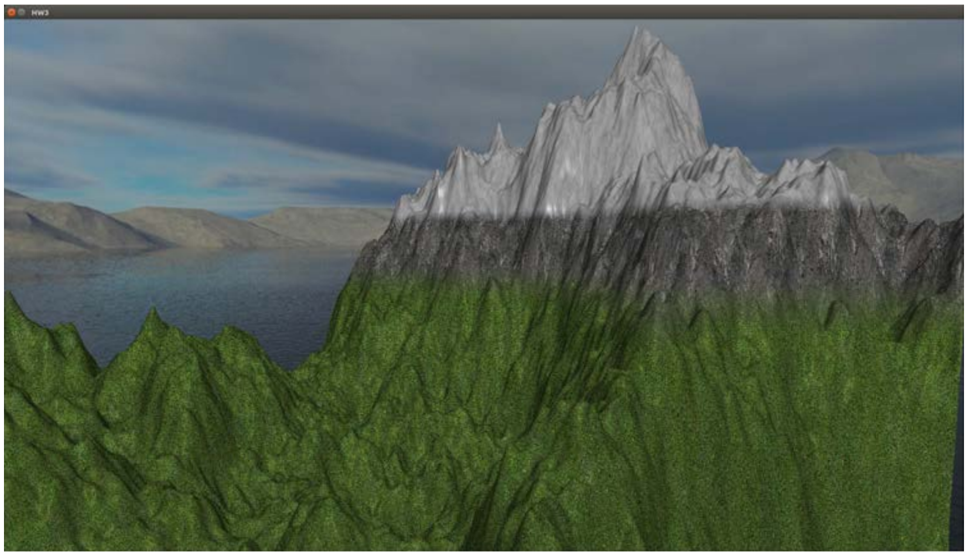

Figure 17: 20 Range Value

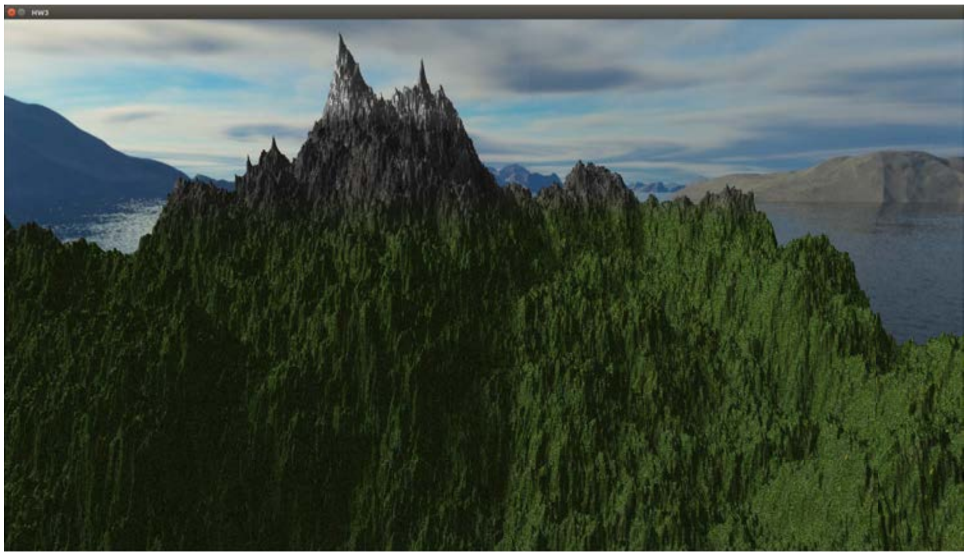

Figure 18: 0.8 Roughness Factor

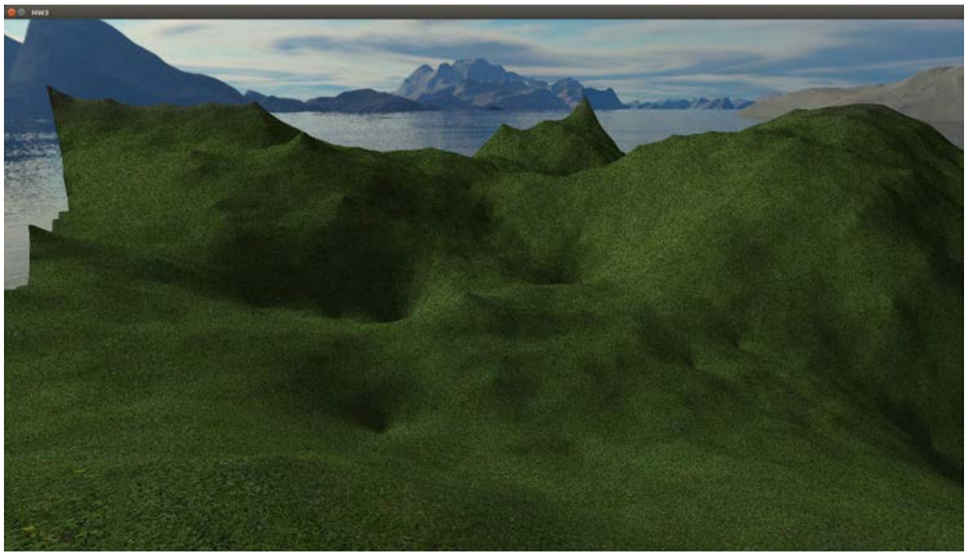

Figure 19: 1.5 Roughness Factor

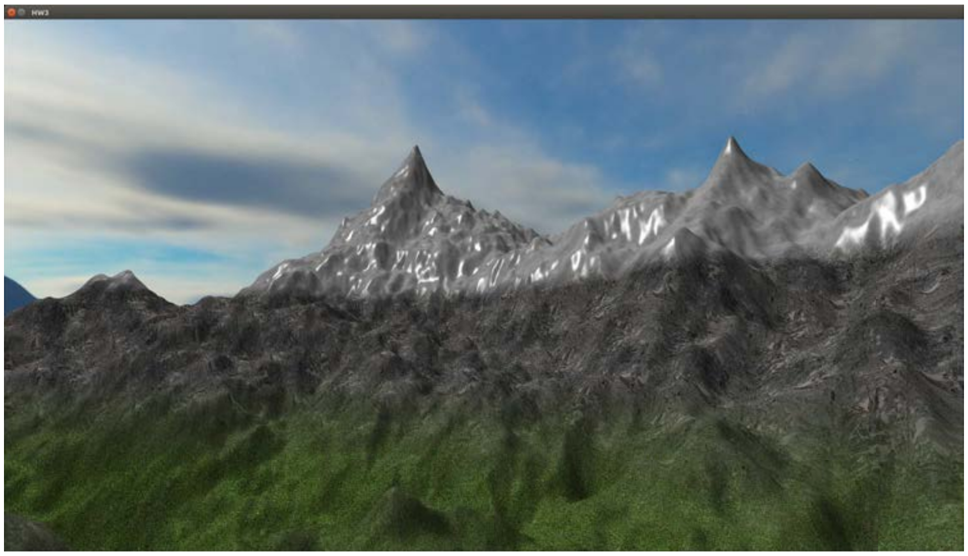

Figure 20: 16 iterations of Laplacian smoothing

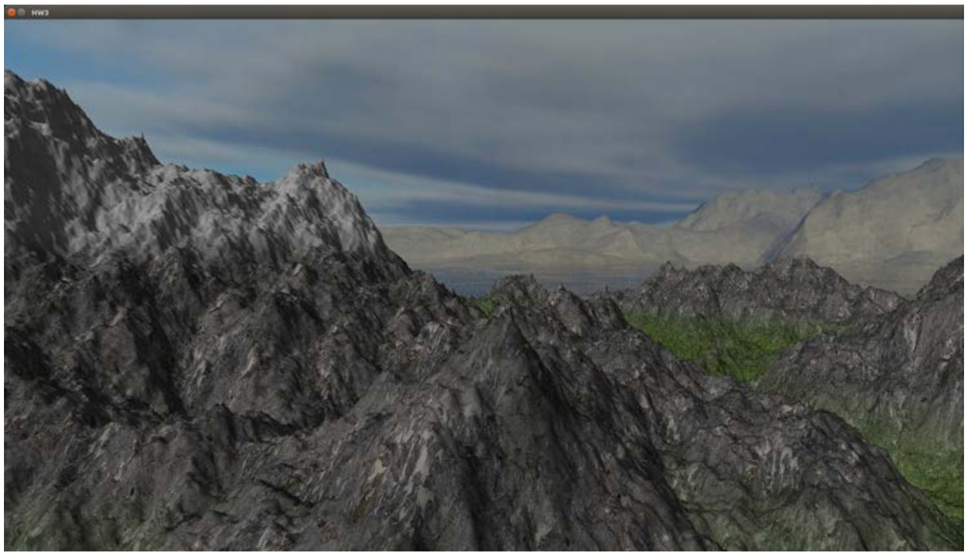

Figure 21: No Laplacian Smoothing

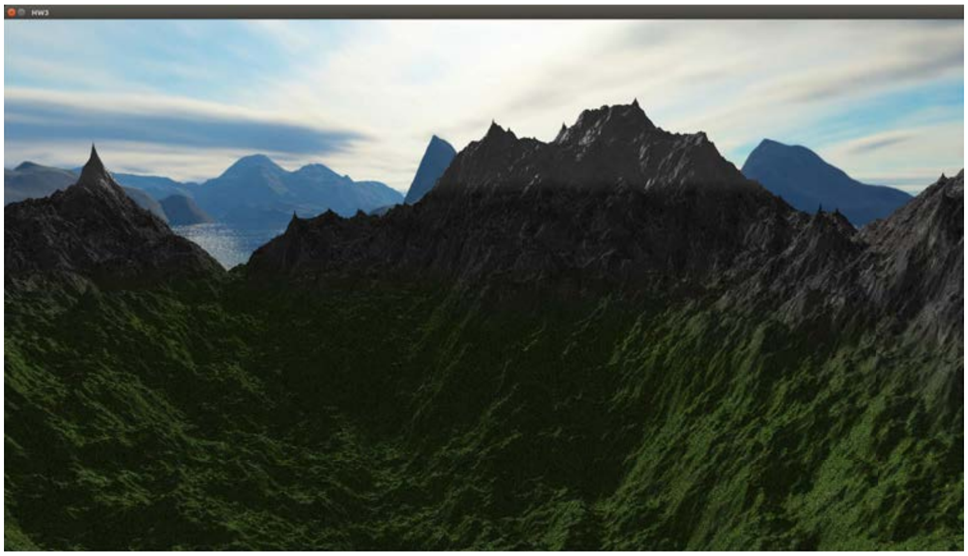

Figure 22: No Laplacian Smoothing

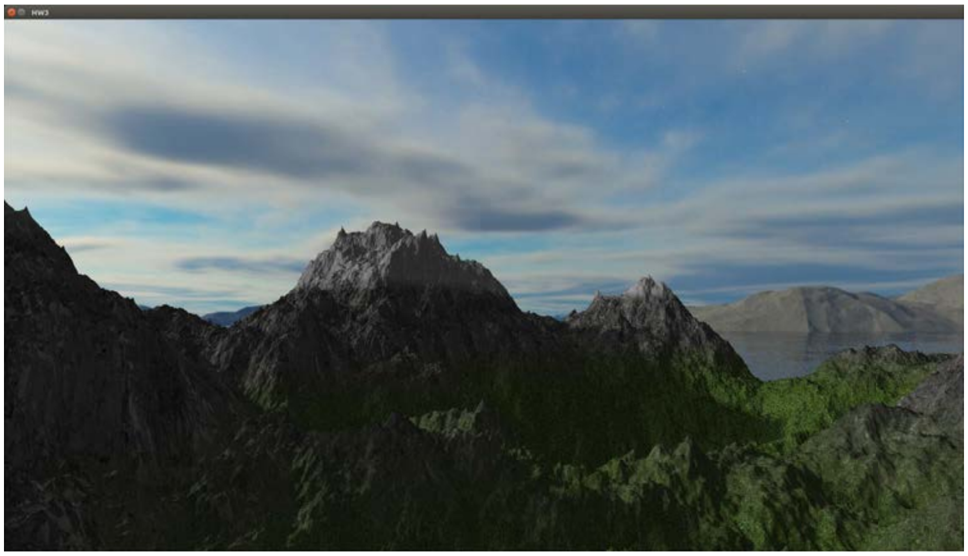

Figure 23: What happiness looks like