# Wednesday 2/15 – 3 hours

I want this to be able to do a live rendering of the image in a window and I’m imagining the user dragging an orange dot fixed in a circle that represents the light source around the image in the center of the screen.

I’m going to implement it in LibGDX. This is a Java library primarily used for game creation but I’m familiar with the methods for drawing and getting mouse input.

Plan for implementation:

1. Create and set up a LibGDX project
2. Find a way to extract the pixel values form a .tga file
   1. Create a 2d array of integers representing the shading on each pixel
3. Create the UI
   1. Accepts mouse input and moves the light source
   2. Draws the output image to the screen
4. Develop a shading algorithm based on the gradient of the surface relative to the light vector.
   1. This will involve looking at the surrounding pixels and determining what the slope is at any given pixel in the direction of the light vector.
   2. Then I will apply a function to the slope to get a shading for each pixel.
5. Develop a self-shadowing algorithm
   1. Tracing light rays?
      1. From the light source?
      2. From the ‘ground’ up?
      3. Maybe follow the ground and ‘look back’ and see if any pixels cover the light?
      4. I’ll think about an O(n^2) algorithm
   2. This can be applied over the shading algorithm because it will simply zero out any calculation the shader made
      1. It might actually be faster to do this one first so that the shader can only calculate pixels that are in light in the first place.

Today I got the project set up and I got a light source displaying and rotating around the center of the screen based on mouse input. I also parsed the image file into a 2d array or integer values representing the shade of the pixel and was able to re-draw it to the screen. The only thing left to do is write the algorithms for directional shading and self-shadows. I have a method that takes both shaded files and outputs a combined shaded output that is to be displayed to the screen.

In working today, I realized that it might be neat to have a button that saves the current output to a file, but I don’t know if it is necessary.

# Saturday 2/18 – 1 Hour

I started writing the self-shadowing algorithm. It’s a O(n^2) algorithm that traces a ray of light back toward the source for each pixel and checks if it collides with a bit of terrain along the way. It’s O(n^2), but for each pixel it could loop up to ~150 times within to trace the ray of light with the current increment set. I can generate the image in about 1/8th of a second, but I’m sure there are ways to make some of the computation happen outside of the render loop. I’ll think about it.

Now for a problem I’m having. The way I currently do the math is in doubles, and then I cast them to each to an int to check the values on the height map. It leads to some weird behavior when the angle of the light changes across each of the multiples of 90 degrees. This is probably because it all of a sudden rounds the values to a different integer per the same ray. This could be fixed in one of two ways that I can see. Both of these ways will slow down the algorithm so I’m not sure how feasible they are.

The first is that I could turn the increment of the ray down dramatically. That way, it will do the height checks more finely. This directly affects the runtime and isn’t the most innovative solution in my opinion.

The second (and my preferred) way is to take a weighted average of the pixels on the height map surrounding a given double. For example:

**Given:**

rayX = 5.6

rayY = 8.3

**Check If:**

rayZ < (0.6\*0.3\*HM(6, 9)) + (0.6\*0.7\*HM(6, 8)) + (0.4\*0.3\*HM(5, 9)) + (0.4\*0.7\*HM(5, 8))

But this is a problem for tomorrow.

And a side note. In order to get this working, I converted the TGA to a PNG file because java has built-in libraries to handling PNG files where it doesn’t for TGA files. I’m not sure if this varies too much from the outlined project, but if it is, I can implement a library to load a TGA.

# Sunday 2/19 – 2 hours

For self-shadowing, I tried to implement the quad sampling of the image to get a more accurate measurement of the height rather than just converting the value to an int. It took far too long to calculate. Instead, I changed the conversion from a double to an int to use Math.round(). It worked fantastically to eliminate the weird things I was seeing as I rounded the vertical and horizontal angles. It is slightly slower, but not too bad. I’ll keep trying to make it faster.

As far as directional shading goes, I have the majority of the algorithm for it written now and just need to calculate the vector pointing into the plane of each pixel. This is definitely going to be the hard part. I think I need to take the pixels in each direction and treat them as eight vectors, then average out the direction of those lines.

I can calculate the average slope in the X direction and the average slops in the Y direction. If I take the cross product of those two vectors, I will get a vector that is perpendicular to both of them, pointing downwards into the plane. I have already written the code to find the angle between vectors and apply a shading value to image based on that.

All in all, today I made the self-shadowing algorithm work much, much better. Albeit a tad bit slower. And I got the framework for the directional shading algorithm all laid out. I’ll have another go at it after exams.

# Wednesday 2/22 – 3 Hours

Today I worked a lot on generating the 2D array of normal vectors to the height map for use in the directional shading algorithm. These are precomputed when the image is loaded in so that the minimal amount of work needs to be done during rendering.

I also fixed a few bugs to do with incorrect or unscaled vectors.

I upped the step size on the ray tracing algorithm because it seemed to increase the speed dramatically with very little cost to quality of the output. You can see some incorrect features in the shadows if you look hard enough, but it is largely unnoticeable. If this didn’t work to speed up the rendering, I would have put more time into thinking about ways to precompute other data structures to make the ray tracing faster. I know of one where you can chunk up your image, and if the ray is higher than every pixel in the chunk you are approaching, you can skip over it all together. This seems like a lot of work, but also has a large speed benefit.

It actually looks and runs great now so I’m going to leave it as is. I went through and made sure everything had comments and was easy to understand. I think this is it for the project!