1 Making a Sphere

This time the sphere has its North and South Poles at $(0,1,0)$ and $(0,-1,0)$ where we expect them. The *prime meridian* ($0°$ longitude) is the right half of the unit circle $x^2 + y^2 = r^2$. Measuring angle θ_{lat} CCW from the South Pole, it is $(\sin\theta_{\text{lat}}, -\cos\theta_{\text{lat}}, 0)$, $0 \leq \theta_{\text{lat}} \leq \pi$. This is not the usual way of measuring latitude, which goes from $-90°$ (South Pole) to $90°$ (North Pole), but it makes things easier for generating the vertices of the sphere polyhedron.

Increasing longitude is CCW about the y-axis, the lines of longitude first moves away from us into the whiteboard, and we don't see them until longitude $180°$ or π radians.

The Sphere class is parameterized by n_{lon} and n_{lat}, the number of lines of longitude and latitude. The index i_{lat} goes from 0, the South Pole, through n_{lat} lies of latitude $1\ldots, n_{\text{lat}}$, to $n_{\text{lat}} + 1$, the North Pole. So,

$$\theta_{\text{lat}} = \pi i/(n_{\text{lat}} + 1).$$

The index i_{lon} goes from 0, the prime meridian, to n_{lat}, the prime meridian again! So,

$$\theta_{\text{lon}} = 2\pi i/n_{\text{lon}}.$$

If we want to map a single texture onto the entire sphere, x and y have to go from 0 to 1. So x is θ_{lon} divided by 2π, and y is θ_{lat} divided by π. For each i, j, the vertex is assigned those texture coordinates, and the fragment shader uses their interpolation between steps.

I will show you my face mapped to a sphere this way. Enjoy.

2 Tangent coordinates

The *tangent* coordinate system at a point p on the Earth (or Moon) is the standard way you look at a map: the origin is p; east is x; north is y; and up is z. Yet *another* coordinate frame? It's even worse! It is an uncountably *infinite* number of coordinate frames: one for every point p on the sphere except the North and South Pole.

Inside the shaders, we already know p and n, the origin and the z direction. If we knew the x direction, called the *tangent*, we could use the cross product to get the y direction, called the *bi-tangent*. We get the tangent the same way we got the normal. Pass in a vector for each vertex as an attribute and interpolate from vertex shader to fragment shader.

What is the tangent direction for vertices on the prime meridian? If you are standing on the right hand silhouette of the Earth, then your east is pointing in the negative z direction of the world (not Earth) coordinate system: $(0,0,-1)$. We get the position of vertices on other lines of longitude by rotating about the y axis. Ditto the tangent direction.

3 Bump Mapping

The n we are using for smooth shading is "fake". The actual normal to the polyhedron gives the true flat appearance. Instead we use *Phong shading*, meaning we interpolate n from the vertices. Phong shading makes the surface look smoother than it really is. If we look near the silhouette, we see it is actually polygonal, not circular.

Bump mapping goes the other way, wiggling the normal to make the surface look less smooth. If there were a bump on the surface, the normal would point to the left on the left side of the bump and to the right on the right side of the bump. So we will fake n even more: altering it to have the value it would have if there were bumps. It will look like bumps, but if we rotate to the silhoutte, we will see that these bumps are more "Stupid Human Tricks".

Where do we get these wiggled normals? Use the texture coordinates to look them up in a *normal map* texture. Textures contain colors with coordinates in the range 0 to 1, but (unit) normal vectors have coordinates in the range -1 to 1. To store the normals in a texture, we add 1 and divide by 2. Most of the surface of the Moon is level, so the normal is $(0, 0, 1)$. That makes color $(0.5, 0.5, 1)$, which is half white and half blue $(1, 1, 1)/2 + (0, 0, 1)/2$. So a normal map looks mostly pastel blue.

Notice that up is always $(0, 0, 1)$, right? The normals encoded in the texture are in the tangent frame. But thanks to the tangent (x) vector, plus p and n, we can compute tangent2model in the fragment shader and figure out the wiggled value of n in model coordinates.

We get k_d and k_s from two other textures. k_d is a picture of the moon. k_s is black inside dusty craters but white on their shiny edges.

4 Height Maps

Where do the bump normals come from? Using radar or lidar, one can determine a *height map* for the Moon or Earth. The normals are derived from the height map. The height map is a single number (height) per pixel instead of three numbers. Back when GPUs had limited memory, one would store the height map on the graphics card and calculate the normals as needed. Let's take a walk down (limited) memory lane.

Given a surface defined by $f(x, y, z) = 0$, the normal vector is the *gradient*

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right),$$

which has the three *partial derivatives* as coordinates. The partial derivative $\frac{\partial f}{\partial x}$ is the derivative of $f(x, y, z)$ with respect to x, treating y and z as constants. So for example, a sphere is $f(x, y, z) = x^2 + y^2 + z^2 - r^2 = 0$, and $\nabla f = (2x, 2y, 2z)$. That's actually $2(p - o)$ instead of the $p - o$ that we used, but it was normalized so it didn't make a difference.

A height map is $z = h(x, y)$ or $f(x, y, z) = z - h(x, y) = 0$ in which case

$$\nabla f = \left(-\frac{\partial h}{\partial x}, -\frac{\partial h}{\partial y}, 1\right).$$

We only know $h(x, y)$ at each pixel. It's not like a polynomial or sin that we can take a derivative of. So we estimate its derivative with respect to x by taking the slope of the line from $(x - \Delta x, h(x - \Delta x, y))$ to $(x + \Delta x, h(x + \Delta x, y))$,

$$\frac{h(x + \Delta x, y) - h(x - \Delta x, y)}{2\Delta x},$$

where Δx is one pixel. Similarly for y. However, texture2D has x and y go from 0 to 1 instead of width and height. For the top (numerator) $\Delta x = \Delta y = 1/512$ but for the bottom (denominator) $\Delta x = \Delta y = 1$.