Com S 336
Fall 2023
Homework 4

Please submit an archive on Blackboard including the files indicated at the beginning of each problem.

1. (*Please turn in your modified version of Lighting3*.js.)

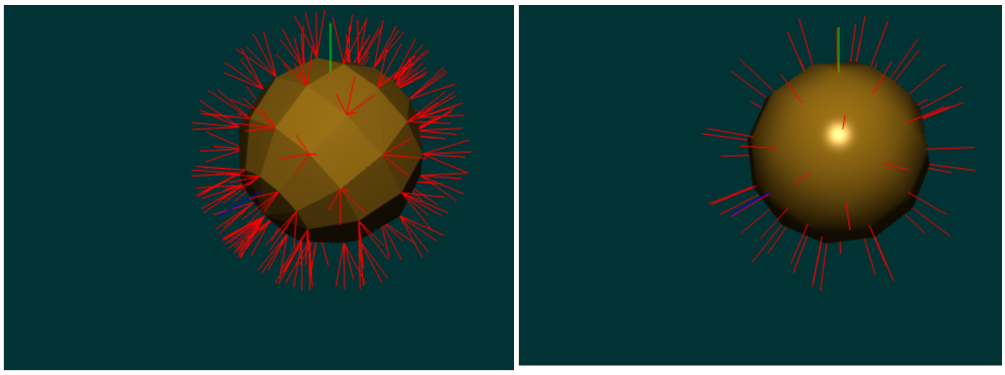
Modify the shader for Lighting3 to use the Blinn-Phong reflection model. That is, instead of using R dot V for the specular component, use N dot H, where H is the "halfway" vector obtained by normalizing L + V. If you make the exponent about 4 times as large, the results should look about the same.

> Rationale: Blinn came up with this as an optimization. It is not hard to show that, when the vectors are aligned in the same plane, the angle between N and H is exactly half the angle between R and V, so this works basically the same way as the Phong reflection model. You just have to increase the exponent by a factor of about 4 to get the same visual effects. The idea was that if both the light and the viewer are far away, then the angle between L and V is relatively constant. That way, H could be calculated once and reused for many vertices, while the in the Phong model the reflected vector R has to be calculated for every fragment.

2. (*Please turn in your modified versions of Lighting3WithObj.js and Lighting3WithObj.html*. **Note that this application depends on hierarchy/CS336Object.js, hierarchy/Camera.js, and three/OBJLoader.js.**)

Modify Lighting3WithObj.js and Lighting3WithObj.html to draw a representation of the normal vectors as lines. You can assume that the model data always has three floats per vertex and three floats per normal vector. Note that in the case of the vertex normals, the same vector will be redrawn each time that vertex appears in the vertex array; this is ok. You can also assume that there is no nonuniform scaling of the model (that is, when drawing the normals you can transform with view * model rather than using the normal matrix). You can just draw the normals as red lines using gl.LINES. Add a text box for scaling the length of the drawn normals. Note that the application has a key control to toggle between using face normals or vertex normals, based on the status of the flag `useFaceNormals`. You should draw the corresponding normal vector based on that flag.

Here are a couple of examples using the basic sphere model. Note that in both cases we are drawing four lines at each vertex for the normal vectors associated with that vertex, but in the right-hand image, it's the same vector all four times.



Notes:

- You'll probably want to define a new shader pair for this
- There isn't any way to tell OpenGL to just "draw a vector" - you have to generate vertex data for the beginning point and end point of each line, to be used by the drawArrays call with gl.LINES. This is not complicated - every three floats in the model's vertices array represents the x, y, and z coordinates of a vertex, which is one endpoint of the line. The corresponding three floats in the model's normals array represents the x, y, and z components of the normal vector. From that, and the given scale factor, you can find the endpoint of the line you need to draw. The total number of vertices for the lines is always twice the number of vertices for the model.
- When the **`useFaceNormals`** flag changes, OR when the scale changes, you'll need to regenerate the data and reload the buffer. Important: DON'T do this every frame, just when a change is required by user input, and DON'T just keep calling, e.g.,

```
myNormalLineBuffer = createAndLoadBuffer(normalLineData);
```

when the data changes. This allocates a new buffer without disposing of the previous one, which you might realize is a huge memory leak. Instead, use the same buffer handle to replace the data, e.g.

```
gl.bindBuffer(gl.ARRAY_BUFFER, myNormalLineBuffer);
gl.bufferData(gl.ARRAY_BUFFER, myNewData, gl.STATIC_DRAW);
gl.bindBuffer(gl.ARRAY_BUFFER, null);
```

3. (*Please turn in your modified version of Lighting4.js*. **Note that this application depends on hierarchy/CS336Object.js.**)

The file Lighting4.js is similar to Lighting3.js, except that a plane is also drawn below the rotating model to look like a "floor". In addition, the light position is represented by an instance of CS336Object, and you can move the light around with the key controls.

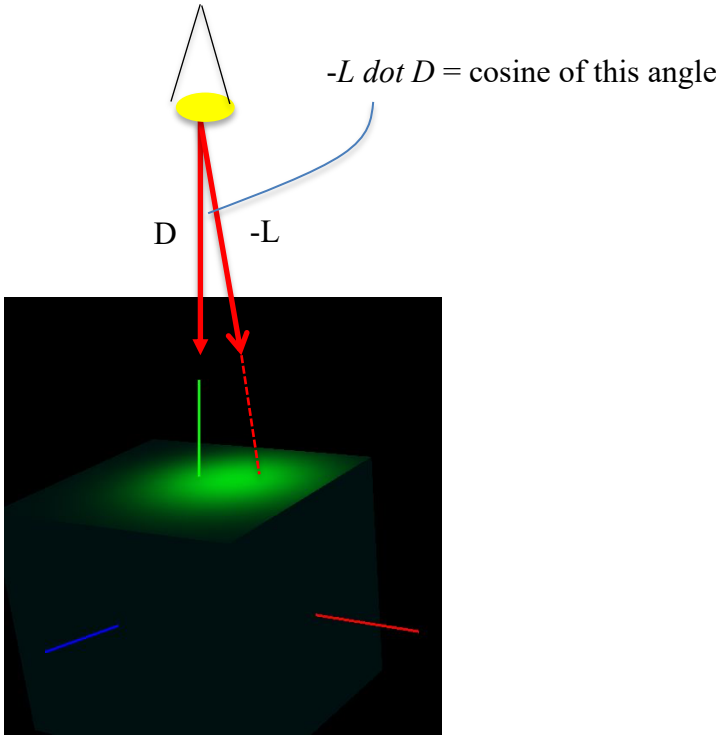
There is a yellow line that represents the object's forward direction (negative z-axis), but this direction currently has no effect on the scene, since the only the light's position is being used. Try it out. Just for shits and giggles there is also a "flat shadow" that is rendered using a simple projection matrix; see lines 480-500 or so to see how this is done. If you move the light object *down* with the w key (it is actually moving forward along its own negative z axis) you'll notice the shadow gets bigger.

Lighting4.html and Lighting4.js are in the **`lighting`** directory, they are dependent on the working CS336Object.js in the **`hierarchy`** directory.

Your task is to make the point light into a *spotlight* whose direction is the negative z-axis of the object described above. One simple way of defining a spotlight is with two parameters (that you will need to define as uniform variables) in addition to the light position:

- A direction, *D*, which is a unit vector representing the direction in which the center of the light cone is pointing
- An exponent, *c*, similar to the "shininess" exponent in the ADS reflection model, that determines how the light intensity decreases as points move further from the center of the light cone

If *L* is the unit vector from a fragment's position toward the spot position, then *–L dot D* is the cosine of the angle between that vector and the direction of the spotlight. See illustration below:



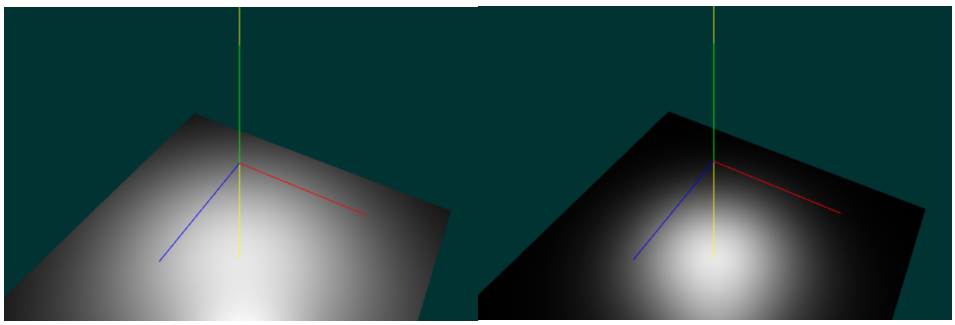
The quantity

$$(max\ (-L\ dot\ D\ ,\ 0))^c$$

can then be used to modulate the spotlight's contribution to the fragment's color. "Modulate" just means that you have some number between 0 and 1 that you are using to multiply some existing value. **In particular, you do all the ambient + diffuse + specular calculations normally, and then apply this factor only at the end so that fragments outside the light's cone are darker.** The exponent *c* determines how tightly focused the spot cone appears. (*Also remember that we calculate L in eye coordinates, so for the dot product to make sense, the vector D had better be transformed into eye coordinates too!*)

The existing keyboard controls can then be used to "aim" the light. Add two more keyboard controls: use 'c' to increase the exponent and 'C' to decrease it, which should make the cone more focused or less focused.

Here are some screenshots with the light in its initial position (pointing straight down). In the left image, the spot exponent is 15; in the right image, the spot exponent is 50.

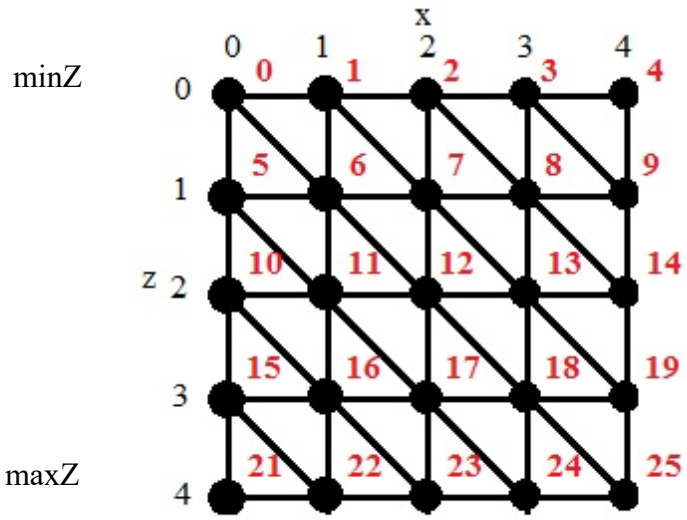


4. (*Please turn in your modified version of homework4/HeightMap.js*.)

One easy and useful way to create mathematically based models is to use a *height map*, in which a function of two variables (typically x and z) is used to generate a 3rd value (e.g. y). From an array of 3d coordinates (x, y, z) generated this way, we can create a wireframe, a polygonal mesh, and normal vectors for rendering.

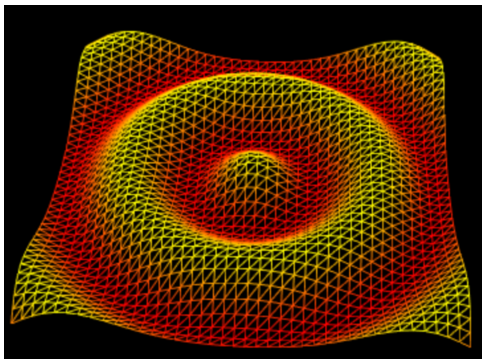
The basic idea is something like this: first compute sample values from the function at equally spaced points in the x-z plane. You can imagine these as a 2-d array. In the x-direction we divide the interval [minX, maxX] into numPointsX equally spaced values, and in the z-direction we divide the interval [minZ, maxZ] into numPointsZ equally

spaced values. In the illustration below, we are looking down the y-axis at the x-z plane, and the parameters numPointsX and numPointsZ are both 5. The black numbers are the rows and columns. The red numbers refer to the *logical* indices of the vertices; this is visualized here as a 2d array, but in practice it would be an ordinary array of floats, in which the three coordinates for the vertex with logical index *a* would be located at actual indices *3a, 3a + 1*, and *3a + 2*. Note that the logical index of the vertex at row i, column j is *i * numPointsX + j*. The diagonal lines indicate the way the vertices are grouped into triangles.

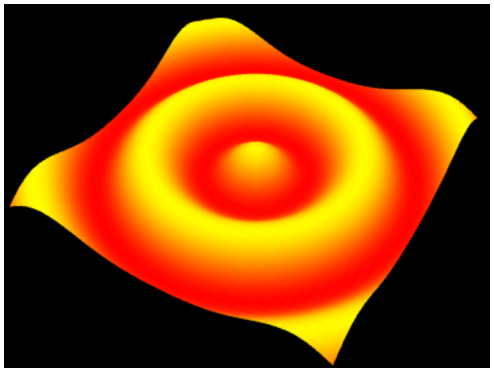


You can see this in the sample code lighting/HeightMap.js, where the methods for generation of vertices and wireframe indices already implemented. **Your task is to complete the implementation of the two methods for generation of indices for the polygons, and generation of vertex normal vectors**. There is a generous amount of internal documentation in the code to get you started.

You can experiment with samples of the wireframes using HeightMapTest1.html, which should work out of the box. Edit the top of HeightMapTest1.js to try out different functions. For the "ripple" function you should see something like this, where the fragment shader picks a color for each fragment based on its height (y-value) in world space:



You can try out your implementation of the mesh indices using HeightMapTest1a, which uses the same shader, but uses the mesh indices instead of wireframe indices, and uses gl.TRIANGLES instead of gl.LINES. It should look about like this:



You can try out your implementation of the normal vectors using HeightMapTest2.html. Using the same “ripple” function for the height map, you would get the image below: