# Com S 336
# Fall 2020
# Homework 4

Please submit an archive on Blackboard including the files indicated at the beginning of each problem. (The first three should actually be quite short, but require you to read and edit the lighting shader code.)

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

Modify the shader for Lighting3.html as follows:

a) Specify the light as a *direction* (a vector pointing toward the light), rather than as a position (this corresponds to the situation that the light source is infinitely far away, e.g., the sun). Note: to test this you'll need to modify the javascript code that sets the uniform variable for light position, so that you pass in a direction vector instead.

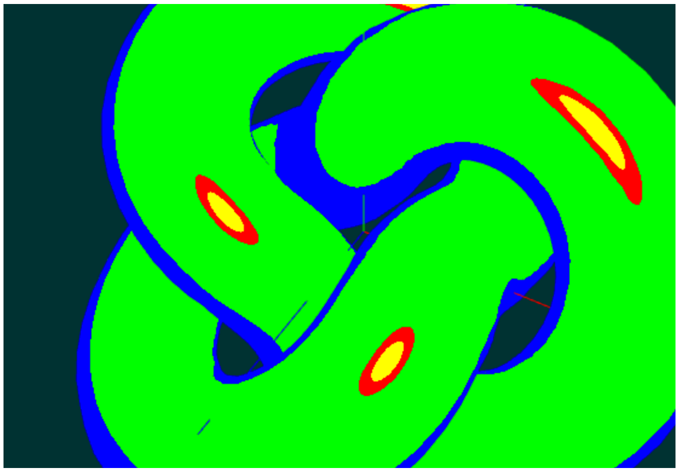
b) Use the Blinn-Phong reflection model. That is, instead of using R dot V for the specular component, use N dot H, where H is the "halfway" vector obtained by normalizing L + V. If you make the exponent about 4 times as large, the results should look about the same.

> Rationale: Blinn came up with this as an optimization. It is not hard to show that the angle between N and H is half the angle between R and V, so this works 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 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 version of Lighting2c.html.*)

Modify the fragment shader in Lighting2c.html to get an effect like that in the screenshot below.

That is, **after** the lighting calculation is done, instead of directly assigning the resulting color to gl_FragColor, use it to find a "greyscale" intensity, (red + green + blue) divided by 3. Then use that number to select one of four solid colors to assign to gl_FragColor. (You can pick the colors and the cutoff values as you wish.)



3. (*Please turn in your modified versions of Lighting4.html and Lighting4.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 kicks there is also a "flat shadow" that is rendered using a simple projection matrix; see lines 350 - 380 or so to see how this is done.

There is also a file Lighting4.html. You will need to make some minimal modifications to both files for this problem. These files are in the **`lighting`** directory, they are dependent on the working CS336Object.js that is in the **`hierarchy`** directory.

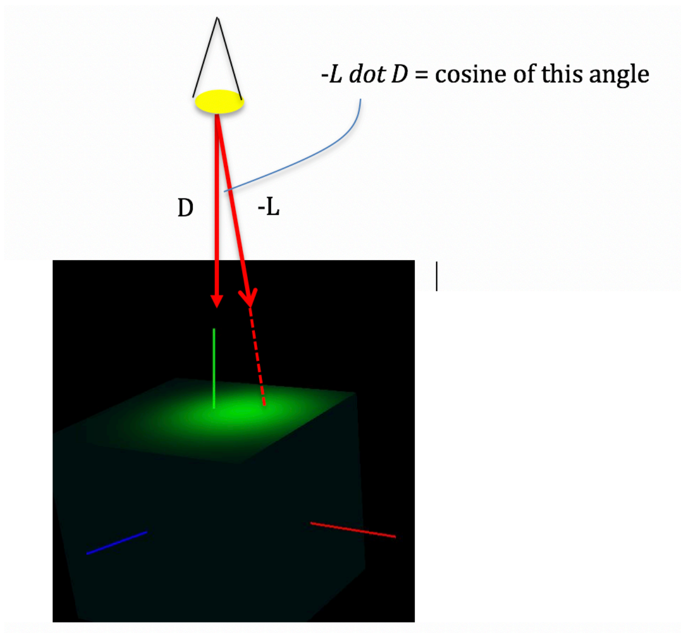
Your task is to make the point light into a spotlight. One simple way of defining a spotlight is with two parameters (that you will need to define as uniform variables) in addition to the light position:

- A direction, *D*, which is a unit vector representing the direction in which the center of the light cone is pointing. In this case, it is aligned with the negative z-axis of the light object.
- An exponent, *s*, 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

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

can then be used to modulate the light'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 *s* determines how tightly focused the spot cone is. (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.

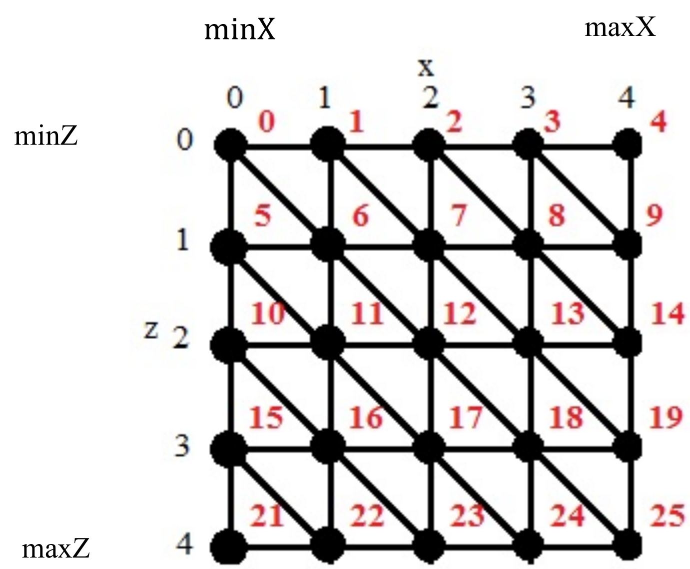
4. (*Please turn in your modified version of HierarchyProblem1.js*.)

Take a look at the simple scene in **`examples/homework4/HierarchyProblem1.html`**, showing a unit cube gently rotating side to side. Then take a look at the video clip here: http://web.cs.iastate.edu/~smkautz/cs336f20/homework/hw4/hierarchy_problem1.mp4

Modify HierarchyProblem1.js so that the result looks like the video clip.

5. *(Please turn in your modified version of lighting/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 illustration below, we are looking down the y-axis at the x-z plane, and the parameters numPointsX and numPointsZ are both 5.

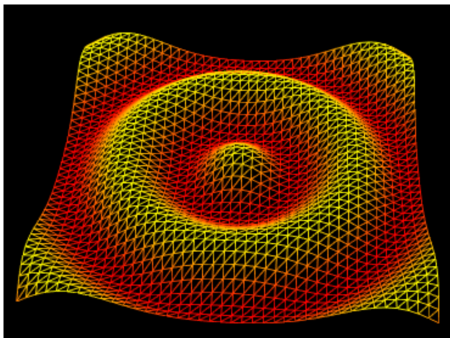


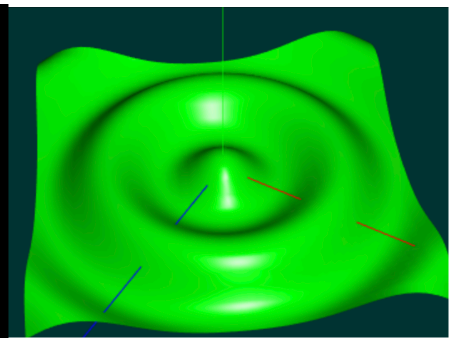

The red numbers refer to the logical indices of the vertices; the black numbers are just rows and columns. (This is visualized here as a 2d array, but in practice it would be an ordinary array of floats, in which the three coordinates for the vertex with logical index *a* would be located at actual indices $3a$, $3a + 1$, and $3a + 2$.).

You can see this in the sample code lighting/HeightMap.js, which includes the methods for generation of vertices and wireframe indices already implemented. **Your task is to complete the implementation of methods for generation of indices for the polygons,**

**and 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 the image at left:





You can try out your implementation of the mesh indices and normal vectors using HeightMapTest2.html.  Using the same function for the height map, you would get the image at right.