Computer Science 591B, Graphics

http://www.cs.umass.edu/∼elm/graphics/

Assignment 4: Gouraud Shading

This assignment is an extension to assignment 3. If you did not finish assignment 3, you may start this assignment by copying the sample of assignment 3 from the class web page.

Once again, following the steps laid out here is designed to make the assignment easier. I recommend that you try to do the assignment in the order presented here, confirming along the way that each step works as expected before going to the next part.

1. The web page given here (http://www.scienceu.com/geometry/facts/solids/coords/icosa.html) gives the coordinates for the 12 vertices of an icosahedron. The first vertex is given by the point (-0.692, 0.000, 0.427), and so on. The page also shows how to form the 20 triangles which make up the faces of an icosahedron by using the 12 vertices (numbered 0 to 11) that are defined above it. The first face uses vertices (9, 2, 6) and so on. Use the coordinates and faces shown on this web page to replace the cube from assignment 3 with an icosahedron. MAKE SURE THE ICOSAHEDRON LOOKS GOOD BEFORE CONTINUING!!!

2. Next, you should add lighting to your rendering. In this step, you should add a "bi-directional" light coming from a distant source in a direction defined by a vector light_dir. Instead of rendering each pixel as simply the color of the face it came from, you should use the cosine of the angle between a face's normal and the light direction to multiply the intensity of the original color value. Don't forget that you will have to apply this multiplier separately to the red, blue and green color channels separately before you combine these into a 32 bit color value. To implement bi-directional lighting, which makes it look like lighting is coming from two opposite directions at the same time, you should take the absolute value of the cosine between the normal and the lighting direction as a multiplier. At this point, your rendering should show faces that are perpendicular to the lighting direction as completely dark, and faces should be the brightest when they are pointing directly at the lighting direction, or in the opposite direction from the light.

3. Next, you want to be able to control the proportion of ambient lighting and directional lighting (from a light source). Define a parameter lighting_mix, such that when lighting_mix is .2, there is 20% ambient light and 80% directional light, and so on. When you set lighting_mix to 1, you should recover the original rendering appearance from problem set 3. Make sure you take the absolute value of your cosine operation from above BEFORE you add the ambient lighting, or you will get very strange results. Refer to the formulas from class or in the book if you are confused. When adding a small amount of ambient light (say 20%), faces should never be completely dark, even if they are pointing orthogonally to the lighting direction (unless of course the color of the face is black).

4. Finally, you should implement unidirectional lighting. This is done by setting a lower threshold of zero for the cosine of the angle between the surface normal and the lighting direction. That is, instead of using the cosine directly as a multiplier, you should set it to zero if it is negative. If you try this directly without changing anything else, you will notice that you get a strange behavior. Some faces that are on the same side of the icosahedron as the lighting source will be bright, while some faces on the OPPOSITE side of the icosahedron will be bright.

   The reason for this is that when you originally defined the icosahedron some of the faces were defined such that their normals point outward away from the center of the object and some were defined such that their normals point INWARD toward the center of the object. To get the proper lighting behavior, you want all of the normals to be defined so that they point away from the center of the icosahedron.

If you want, you can try to think about the geometry of how the icosahedron is defined and try to fix the problem directly, but this is VERY DIFFICULT. An easier approach is to devise a simple test to say whether a triangle's normal is pointing outward (which is correct) or inward (which is incorrect). If it is pointing inward, then all you need to do to rectify the situation is swap two of the vertices in the definition of that triangle. You can do this automatically for any face of the icosahedron which fails the test.

Once you've done this, you should get appropriate unidirectional lighting like I will demo in class.

5. The final program you turn in should do a mixture of 20% ambient lighting and 80% unidirectional lighting from the right side of the object.
6. EXTRA CREDIT (10 points): Make the icosahedron still and make the light source move around it randomly. This is easy!

Turn in your code as usual to Alex.