UM EECS 487
Lab 5: Lighting and Vertex Buffer Object

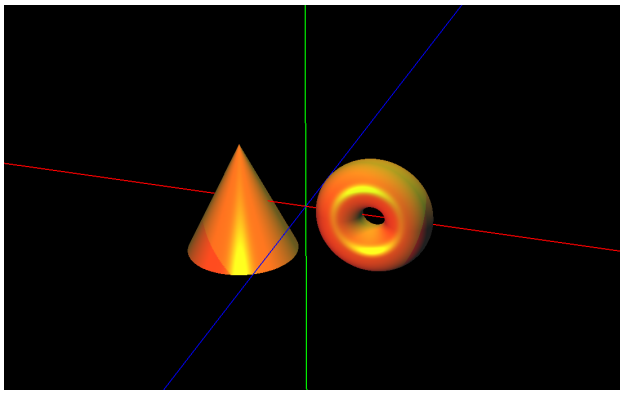

FIGURE 1. Screenshot showing a GLU cone and a GLUT torus with all three lights on.

In the beginning the world is formless as there is no light. You think to yourself, "Let there be light," and put three lights in the world: one light to lit the world, one to lit the way, and one to put a shine on things. In doing so, you became intimately aware of how differently modeling transforms and viewing transforms effect various lights. You will have to think about where *and when* (in the code) to modify the position of your lights to achieve the behavior of the three lights:

`SUNLIGHT:` a yellow directional light from a far away source, like the sun; sunlight lits the world and maintains its position and direction relative to the world in eye space. Pressing the '`1`' key toggles this light on and off.

`EYELIGHT:` a green spotlight, like a light mounted on a helmet or eyeglasses; eyelight lits the way and is anchored to the eye and maintains its position and orientation relative to the eye in world space. Since eyelight is a spotlight, whenever you change the position of the light, you must also adjust its spot direction. The eyelight has been set up to attenuate: as the camera gets closer to objects, eyelight reflects brighter. Pressing the '`2`' key toggles this light on and off.

`OBJLIGHT:` a red light anchored to the objects in the world. It maintains its position and orientation relative to the objects in all cases. This light has a specular component that puts a shine on the objects. Pressing the '`3`' key toggles this light on and off.

Upon start up, the support code shows you a scene similar to image in Fig. 1, but without the nice lighting. In `lighting.cpp:init()` you initialize the position of the three lights. Everytime you move your camera or perform some modeling transform on the objects, you would also need to reposition some or all of your lights—these would be at the end of `lighting.cpp:kbd()` and `lighting.cpp:motion()`. Your surfaces will not interact with your light until you specify how the material of the surface reflects light. Use `glMaterial*()` to set the diffuse and specular responses of the material in `lighting.cpp:init_properties()`. Search for both "YOUR CODE HERE" and "AND/OR HERE" in the `lighting.cpp` for further instructions. You also need to enable some OpenGL states in `lighting.cpp:init()` to ensure OpenGL has an understanding of depth (otherwise your objects may behave and interact weirdly) and to get nice shading on your objects.

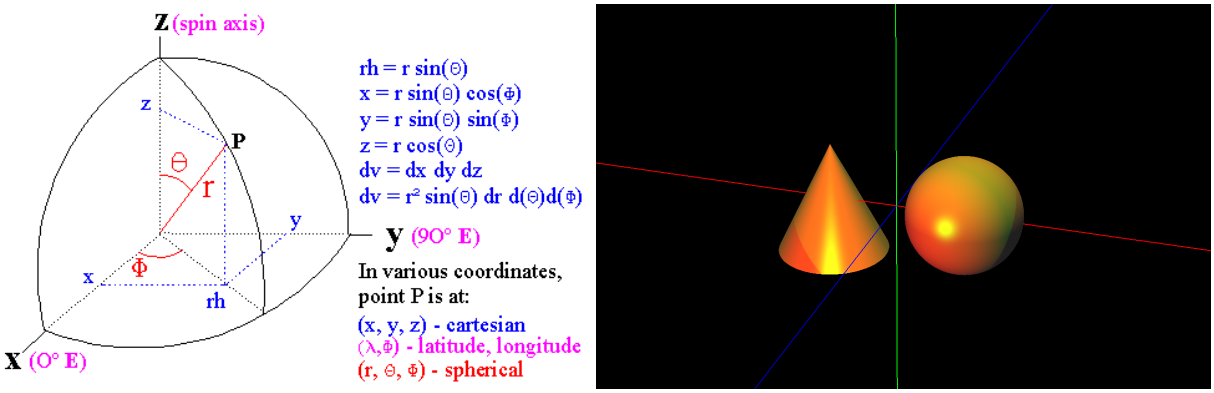

FIGURE 2. The spherical polar coordinate system (*Image: G. Steinle-Neumann*). On the right, a screenshot showing a GLU cone and a home-brewed sphere with lighting on.

All the keyboard shortcuts to control the camera work as in the previous lab on viewing transforms. In addition, holding the "ALT" key while pressing the shortcut keys allows you to apply translation transforms to the objects as in the previous lab on modeling transforms. Similarly, holding down the "CTRL" allows you to apply rotation transforms to the objects. As before, pressing the '`I`' key anytime in the program resets the objects to their initial positions and orientations.

To test whether you have implemented the three lights correctly, observe their behavior and see if they conform to the following:

`SUNLIGHT`: set up to shine from above the world, including above the camera. If you roll the camera upside down, objects now look upside down on screen, but sunlight still shines on top of the objects, even if the top of objects is now at the bottom of the screen.

`EYELIGHT`: when you change the camera location or orientation, eyelight follows the camera, i.e., its light slides over objects as either the camera or objects move out of its range. Eyelight also attenuates: as the camera gets closer to objects, eyelight reflects brighter.

`OBJLIGHT`: when you rotate the objects around any axis or when you move them along any axis, this light follows the objects and shines on the same spots on the objects.

You may want to implement and test your light one at a time, using the keyboard shortcuts to turn individual light on or off.

VERTEX BUFFER OBJECT

A Canonical Sphere. An arbitrary point on the surface of a sphere can be described in terms of spherical polar coordinates r, θ, and ϕ as shown in Fig. 2. The radius r is constant, $0 \leq \theta \leq \pi$, and $0 \leq \phi \leq 2\pi$. The points on the sphere can be generated by selecting a constant r (choose 1.0) and varying θ and ϕ in regular increments. The xyz-coordinates are related to the $r\theta\phi$-coordinates as:

$$z = rcos\theta, \quad x = rsin\theta cos\phi, \quad y = rsin\theta sin\phi.$$

In `objects.cpp:drawSphere()`, draw your sphere using *only* `GL_TRIANGLE_STRIP`. Modern OpenGL no longer supports `GL_QUADS`, thus you *must* use `GL_TRIANGLE_STRIP` to model the sphere. Perform the drawing using vertex buffer object and `glDrawElements()`. You are not required to write your own vertex shader for this lab, instead, you can use the fixed-function pipeline and client-side vertex attribute locations (`GL_VERTEX_ARRAY`, `GL_NORMAL_ARRAY`

are the only two you'll need). Recall the vertex ordering of a `GL_TRIANGLE_STRIP` as shown in Fig. 3. Don't forget to set the normal of your vertices. Make sure the normals are of unit length, i.e., normalized. You can either do this manually or by enabling an OpenGL mode (find it!). The normals are to be passed to the CPU by the use of vertex-array object also. The sphere you draw should be of the given radius. Pressing the '`M`' key increases the size of the sphere, pressing the '`m`' key decreases its size. You should define and populate the vertex buffer objects only once and reuse them to draw sphere with different radii. The right image of Fig. 2 shows a lit home-brewed sphere along with a GLU cone. You can toggle between showing the GLUT torus or your home-brewed sphere by pressing the '`0`' key. In `objects.cpp:drawWorld()` you can uncomment the call to `gluSphere()` to see the expected look and behavior of your sphere. Make sure you comment out the call to `gluSphere()` and uncomment the call to `drawSphere()` before you turn in your lab. Search for "YOUR CODE HERE" in `objects.cpp` to find all the places relevant to this task.

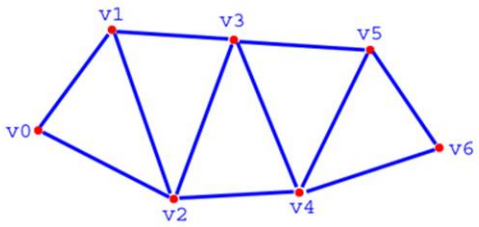

FIGURE 3. Vertex ordering in a `GL_TRIANGLE_STRIP` (*Image: cobrahc*).

IDE SETUP

This project requires the use an OpenGL extension loader such as the GLEW library. Please consult the course note at `http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/ide/options.html` on how to install and use GLEW.