CMSC 23700	**Introduction to Computer Graphics**	**Project 3**
Winter 2012		**Feb 22**

Quaternions and Advanced Lighting

Due: Friday, March 2 at 6:00pm

(v1.0 Feb 22: initial version)

1 Summary

The purpose of this project is to gain experience with more advanced rendering effects, the mathematics of quaternions, and our first brush with animation and motion.

As with previous projects:

- You complete this project in pairs or individually.
- Your code will build upon GLFW (documented here) and a OpenGL 3.2 core context.
- The interactive control of variables is via AntTweakBar.
- Maclab macs are the reference platform for development, debugging, and grading.

2 Software

Unlike with previous projects, a `proj3.c` is not being provided. You can use your previous `proj2.c` as your starting point (to avoid the hassle of merging your modifications into a new file). Your code should still support the keyboard and mouse interactions detailed in the Project 1 description, except for the scene initializations and descriptions triggered by "1", "2", etc., which are not needed here. Instead, your project should be launched as "`./proj3`" and use a single fixed AntTweakBar window to permit controlling the various components of the project listed below, indicated by bullet points below.

The `spot` library has been expanded from the last project to provide more helper functions. This includes quaternion related functions (like `spotQuatToM3()`) and a timer function `spotTime()`. As before, the documentation for the library is the `spot.h` header file. You are again responsible for reading through this to learn what functions are available and what they do, and how the functions are distributed amongst the source files. You shouldn't have to modify anything about `spot` to complete the project. If you do make modifications to `spot`, justify these in comments around the modification.

3 What You Implement

Project 2 used Phong *lighting*, which will continue to be used here. Use Phong *shading* for all parts of this project (no need to toggle between this and Gouraud shading).

- Offer the same controls for K_a, K_d, K_s, and shexp as before.

As described in the next section, you will allow one of the objects in your scene to be rotated according to user interaction. Let this be called the "spinning object"; other sections will refer to this.

- Permit the spinning object to be toggled between cube1 (generated by `spotGeomNewCube1()`), the soft cube (from `spotGeomNewSoftcube()`), and any other shapes you find interesting.

Your project will consist of a single scene that will be adjusted and rendered in different ways. Feel free to be creative in how you set up your scene to demonstrate mastery of the methods involved. Do your best to ensure that all the different parts can operate simultaneously: the freely spinning object (Sec. 3.1) can be environment mapped (Sec. 3.2) as the textured light (Sec. 3.3) casts a (moving) shadow (Sec. 3.4) of it.

3.1 Rotations via Quaternions

In Project 1 you implemented a mode "m" for your interface that allowed you to translate a particular object (your choice) in your scene. Now you will add rotation of a particular object, and allow that rotation to continue while the user is not interacting with the object. The incremental changes to object orientation, and the object orientation itself, should be implemented with *quaternions*. Use rotation matrices only as part of building the model transforms sent to the shaders. The mode "m" rotations described here are just for the spinning object, not for the camera (as in mode "v"). Make the effect of the rotation match what you developed for the Project 1, i.e. dragging to the right in the main part of the window moves the front of the object to the right.

- Permit clicking and dragging in the main (upper right $4/5$s) part of the window to rotate the object about its model-space origin (and translating the object should still moves in world-space the model-space origin). Horizontal motion should cause rotation around the vertical $\mathbf{v}$ view-space basis vector, and vertical motion should cause rotation around the horizontal $\mathbf{u}$ view-space basis vector.
- Clicking and horizontal dragging in the lower $1/5$ of the window should cause rotation around the into-screen $\mathbf{n}$ view-space basis vector.
- Once you let go of the button, the object should *continue* spinning along the axis it was last rotated around, and with the same angular velocity as at the time the mouse button was released. The object should stop spinning only when, in mode "m", you click and release the button without moving it (that is, no rotation, or zero angular velocity). The spinning should continue during changes in other modes.

This will mean *not* calling `glfwWaitEvents()` during the `main` loop, so that rotation with a particular angular velocity can be shown with the repeated calls to `contextDraw()`, according to the time of the rendering. Exponentiation and logarithms of quaternions (or their conversion to an angle/axis representation) may be useful here.

3.2 Environment Mapping

Permit the spinning object to appear as a purely reflective surface, on which the eye direction is reflected across the surface normal to index into the environment map. Note that GLSL has a `reflect()` function. The eye direction must vary across the scene (according to the relative position of the eye position and the position of the fragment being rendered, as opposed to being the fixed $\mathrm{at} - \mathrm{from}$.). This will be checked by look at the how the environment map is reflected in cube1 with a close eyepoint. Use whatever texture filtering seems to look good.

- Allow toggling of whether the spinning object is environment mapped, versus appearing with normal Phong lighting and shading. You don't have to implement any notion of partial reflectivity, or a blending of Phong and environment mapping.
- Allow toggling between the cubic environment mapping stored in `textimg/cube-sample.png` and one of the other environment maps that will be provided.

3.3 Textured Spotlight

Instead of always using a directional light (at infinity), permit the scene to be rendered with a textured spotlight. Use whatever texture filtering seems to look good. The cross-sectional shape of the beam is square, and texturing within that square determines the pattern of cast light. The spotlight should always be shining towards the world-space origin, but its location relative to the origin should be controlled (like the directional light direction) by rotations in mode "l". Consistent with the user interactions implemented for the camera for Project 1:

- The umbra angle of the spotlight (outside of which no spotlight illumination is possible) should be controlled by clicking (without shift) and vertical dragging in the left 1/5th of the window.
- The fall-off function for the spotlight should obey Eq. (7.15) from Real-Time Rendering, and the r_{start} and r_{end} distances should be controlled by shift-clicking and vertical dragging in the left 1/5th of the window, perhaps by using this to control the size of the interval $r_{\text{end}} - r_{\text{start}}$ which you center around the world-space origin.
- The orientation of the spotlight texture relative to the central spotlight direction should be controlled by the arrow keys (in mode "l"), rotating the light texture by 3 degrees with each arrow key press.
- The distance between the spotlight and the world-space origin should be controlled by shift-clicking and horizontal dragging in the bottom 1/5th of the window.

You can conclude that the parameters that define the spotlight are very much like those of a camera, so it makes sense to use a `camera_t` struct to represent this information. In these terms, the umbra angle is like the spotlight's FOV, the r_{start} and r_{end} distances are similar to near and far clipping plane distances, the aspect ratio of the spotlight is always 1.0, the "at" is always the world-space origin, and the projection is always with perspective (not orthographic). The light's "up" vector is not fixed, and is adjusted with the arrow keys.

The functions you developed for Project 1 to create the view and projection transforms from a camera specification will be useful here, to determine the texture coordinates (in the light's texture) of a given fragment. The spotlight provides the per-fragment light direction and light color that you can use in the context of Phong lighting (as in Project 2). Outside of the spotlight illumination, the only visibility will come from the ambient term controlled by K_a.

- Permit toggling between the simple directional light (from previous projects) versus the textured spotlight. The initial (default) state of the spotlight should produce an informative rendering of the scene.
- Permit toggling between the supplied `textimg/spot-circle.png`, `textimg/uhic-rgb.png`, and any other textures you find interesting.

3.4 Shadow Mapping

Allow your textured spotlight to cast shadows on your scene, using shadow mapping. You will have to render the scene from the viewpoint of the spotlight to generate the depth map that is queried to determine whether a given fragment is in shadow. You don't have to implement shadow map filtering to soften the shadow edge. Make sure that you set up your scene so that it is convenient to see the spotlight cast a shadow of the spinning object onto another surface (perhaps a large textured square).

- Toggle whether shadow mapping is used.
- Permit switching between shadowmap resolutions 128^2, 256^2, and 512^2.
- Permit adjusting the bias (see Figure 9.17 in Real-Time Rendering) that you use in your depth comparison, but be sure to initialize it to a sensible default that works for your scene.

4 Style Considerations

The correctness portion of the grade (70%) will be determined by how accurately and completely you implement the functionality described above. For the style portion (30%), refer to the style considerations from Project 1.

5 Submission

This project will be submitted via `svn` the same as Project 1. Be sure to hand-in **all files**, including shaders (via `svn add` and `svn commit`), required to run your project in all its functionality!

For those of you working in pairs, please note the following:

1. You only need to commit your code in one person's repository.
2. IMPORTANT: **Both** repositories must include a modified `proj2.c` file that clearly states in a comment at the top:
   (a) The Names and CNetIDs of **both** students in the pair
   (b) The Name and CNetID of the one student who's repository should be graded

Remember the late policy: late programming projects will be docked 12% every 6.0 hours, in increments of 6.0 hours, based on the clock on the Pheonixforge server. This is linear, not an exponential process; after 48 hours the project will receive a zero.