CMSC 23700 **Winter 2012**	**Introduction to Computer Graphics**	**Project 2** **Feb 2**

Shading and Texturing

Due: Friday, Feb 17 at 6:00pm

(v1.0 Feb 2: initial version)

1 Summary

You can complete this project in pairs (a group of two people) or individually.

The purpose of this project to is to explore different ways of specifying surface appearance using shading and textures. Your starting point will be your code from Project 1, which will be used for its interactive camera and lighting control. The code will again use the GLFW library and an expanded version of the `spot` library. New with this project, however, is writing and debugging your own vertex and fragment shaders, and using AntTweakBar to help adjust the many parameters involved.

As with the previous project, the Maclab macs will be the reference platform for development, debugging, and grading.

2 Software

Your code should still implement all the keyboard and mouse interactions detailed in the Project 1 description, with one exception: the scene rendering and printed description triggered by typing "1", "2", and "3" can be removed; these will not be used as such in this project. However, you may want to implement something similar in order to switch between different scenes and rendering parameters, so that the grader will have an easier time of assessing your implementation. See "Suggested Process" section below.

We're still using GLFW. The documentation at http://www.glfw.org/documentation.html is informative. We are using only a "core" OpenGL 3.2 context.

The `spot` library has been expanded from the last project to provide more helper functions. 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.

A skeleton of the Project 2 code is provided to you in `proj2.c`. Like last time, it contains many "`YOUR CODE HERE`" sections; these indicate places were it is especially likely that you will add new code. Still, feel free to modify the rest of `proj2.c` too, as well as re-organize its functionality into different source files as you see fit (updating your Makefile appropriately).

Note that a useful keyboard callback has been added; when you type "d" the currently rendered screen is saved to a PNG file in the current working directory. Some care is taken to ensure that a new filename is used each time. If nothing else, this may be useful when trying to document for the TA and instructor a problem you are having.

The provided skeleton also includes some example usage of AntTweakBar, a minimalist but effective

way of providing some interactive adjustment of various parameters. You will be adding controls in the "tweak bar" for parameters in your renderer, so that you can explore their effects.

3 What You Implement

Besides the new functionality described below, one small change from Project 1 behavior is that for Project 2, the light direction should move *with* the camera in mode "v". The effect should be that during view rotations in mode "v", the light should appear to be fixed relative to the viewer (instead of fixed relative to the world). Rotations in mode "l" still rotate the light direction as before.

3.1 Shading

For this part, you will re-use scene geometry that you created for the last project (in which the objects have a single fixed per-object color), but you will implement Phong lighting with Gouraud shading and Phong shading. You can implement a simplified form of the lighting which determines the pixel color $\mathbf{C}$ with

$$\mathbf{C}_{\text{phong}} = K_a\mathbf{C}_{\text{mat}} \; + \; K_d\mathbf{C}_{\text{mat}} * \mathbf{C}_{\text{light}}\max(0, \mathbf{n}\cdot\boldsymbol{\ell}) \; + \; K_s\mathbf{C}_{\text{light}}(\max(0, \mathbf{n}\cdot\mathbf{h}))^{\text{shexp}}. \qquad (1)$$

There is a single light at direction $\boldsymbol{\ell}$ with RGB color $\mathbf{C}_{\text{light}}$, and the ambient light color is white $(1, 1, 1)$. At the point being rendered, the material color is $\mathbf{C}_{\text{mat}}$, $\mathbf{n}$ is the surface normal and $\mathbf{h} = \frac{\mathbf{e}+\boldsymbol{\ell}}{|\mathbf{e}+\boldsymbol{\ell}|}$ is the halfway vector between the directions towards eye $\mathbf{e}$ and light. Per-component multiplication of material $\mathbf{C}_{\text{mat}}$ and light $\mathbf{C}_{\text{light}}$ color is with "$*$" (for which the RTR book uses "$\otimes$"), and the remaining scalar parameters determine the contributions of the ambient (K_a), diffuse (K_d), and specular (K_s) terms, with shininess exponent shexp. It is typical to have $K_a + K_d + K_s \approx 1$ but this is not a hard constraint; some extra specular term may help bring out the highlights. In Gouraud shading, $\mathbf{C}_{\text{phong}}$ is computed at vertices, and then (this color) is interpolated across triangle faces. In Phong shading, $\mathbf{n}$ and $\mathbf{h}$ are interpolated across triangle faces, and $\mathbf{C}_{\text{phong}}$ is computed per-fragment.

In your program, permit interactive adjustment of the light position (as described above), and, via the "tweak bar", toggling whether the rendering should be via Gouraud versus Phong shading, and varying light color, K_a, K_d, K_s, and shexp. Permit shexp to go up to about 400; you may find it more useful to interactively control a logarithm of shexp rather than shexp itself.

3.2 Texture-mapping (Color)

3.2.1 Vertex coloring by hand vs. Fragment coloring by texture

Analogous to how Gouraud shading interpolates per-vertex colors, you can roughly approximate the effect of texture mapping by assigning per-vertex color based on per-vertex texture coordinates, and then interpolate those colors.

Generate a sphere with `sph = spotGeomNewSphere()`. Generate and fill an image buffer with `img = spotImageNew()` and `spotImageLoadPNG(img, "textimg/uchic-rgb.png")`. Traverse the vertices, reading off the (U, V) texture coordinates in `sph->tex2`, and using those to index into the `img->data.uc` image data array (using only nearest-neighbor interpolation) to obtain and set per-vertex color in `sph->rgb`. Render this with Gouraud shading of diffuse (Lambertian) lighting (*i.e.*, $K_a = 0$, $K_d = 1$, $K_s = 0$).

Then, write a fragment shader that indexes into the same image data to assign a per-fragment color based on interpolated (across triangle faces) texture coordinates, and render this also with Gouraud shading of diffuse lighting.

Permit toggling (via a control in the tweak bar) back and forth between the per-vertex coloring and true texture mapping. The orientation and size of the image as it appears on the surface should be the same, for both per-vertex coloring and true texture mapping, but the resolution of the image details will be very different. With this working, you'll have first-hand knowledge of exactly how OpenGL maps texture coordinates to image pixel coordinates, and also some appreciation of the value of texture-mapping to depict small details across a comparatively simple surface.

3.2.2 Texture interpolation

As you have probably noticed by now, there is a jagged seam on the texture-mapped sphere, caused by the interpolation of disparate U texture coordinates from the vertices of the triangles that straddle the line $U = 0 = 2\pi$. Figure out how to make the seam go away. See if you can do this without relying on a second texture image, and without changing anything about the per-vertex attributes of the `spotGeom`. Be able to toggle whether or not the seam is fixed.

3.2.3 Texturing filtering

Generate a square with `sqr = spotGeomNewSquare()`, and texture this with one of the simple textures supplied (such as `"textimg/check-rgb.png"`), and adjust your fragment shader so that many copies of the texture image are repeated onto the square, using `GL_REPEAT` for the texture wrapping (corresponder) function. By controlling this and the camera view you should be able to make an image that shows the texture effectively repeating *ad infinitum* off into the horizon. Render this with only ambient light (*i.e.*, $K_a = 1$, $K_d = 0$, $K_s = 0$).

Permit switching between the following filtering operations:

1. `GL_NEAREST` for both min- and mag-nification, no mipmap
2. `GL_LINEAR` for both min- and mag-nification, no mipmap
3. `GL_NEAREST` for magnification, and `GL_NEAREST_MIPMAP_NEAREST` for minification (with mipmap)
4. `GL_LINEAR` for magnification, and `GL_LINEAR_MIPMAP_LINEAR` for minification (with mipmap)

3.3 Bump-mapping

Implement bump mapping, using textures for which there is a (blue-ish) normal map image supplied in "`textimg`" for a corresponding color texture image. Allow toggling of whether bump mapping is applied, and support rendering the bump mapping with Phong shading (with the same interactive shading controls as in part 3.1 above), so that you can watch how the specular highlights move over the bumps. This will help you debug whether the highlights on the bumps are in the right locations relative to the light position (as indicated by the diffuse component of a non-bump-mapped surface).

3.4 Parallax-mapping

Implement the parallax mapping, using textures for which there is both a (blue-ish) normal map image and a height-field image supplied in "`textimg`" for a corresponding color texture image. Use the equation (6.8) in RTR by Welsh that limits the amount of parallax-induced offset. Allow smooth changes in the amount of parallax-mapping, from none (plain bump-mapping), to some maximal amount.

4 Suggested Process

As with last time, you will need to start early in order for this to be completed on time. Budget your time, and ask questions when you hit barriers to progress. Here is a list of the basic things to implement

1. Phong vs Gouraud shading of Phong lighting, ability to adjust Phong lighting parameters. Also: manual per-vertex "texturing" vs real texture map.
2. Demo of different filtering options.
3. Fixing parameterization of texture on sphere (no more seam).
4. Bump mapping.
5. Parallax mapping.

If it is easier for you, start by write separate programs for each of these, or implement some command-line parsing (of `argc` and `argv`) that enables running one program in different ways. Then, if you have time, you can reintegrate these into a single program, for which typing "1", "2", etc, will trigger displays (with geometry and parameters) relevant to demonstrating mastery of the material suggested by the numbered points above.

In any case, we ask that you include a text file or some other write-up that describes how to use your code to assess the implementation of the listed tasks.

5 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.

6 Submission

This project will be submitted via `svn` the same as Project 1. 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

Be sure to hand-in all files (including shaders) required to run your project in all its functionality. Also hand in some description of how to run your software.

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 72 hours the project will receive a zero.