CS 4620 Programming Assignment 2A
PA2A: Shading and Lighting

out: Monday 15 October 2012
due: Wednesday 24 October 2012

1 Introduction

In this assignment, you will implement several shaders in GLSL and integrate them into the modeler environment from PA1. There are three main components to the assignment:

0. Port over the features you implemented in PA1, fixing them if necessary.
1. Implement a suite of visual effects using shaders (normal shader, per-pixel Blinn-Phong shading, "toon" shader).
2. Use the vertex shader to implement a non-linear transformation (making a flower bend to follow a light source).

For part 1 above, you will use your shaders in an extension of the modeler interface from PA1 that allows the user to select what shader to apply to each object in the scene graph. For part 2, a separate interface allows you to interact with the flower shader (Section 3.2).

2 Interface Overview

The interface for Problem 1 extends the modeler interface from PA1 by adding a drop-down box for selecting what shader is used to rasterize each object in the scene. Two of the options work out-of-the-box: "GL Fixed-Function Blinn-Phong" renders objects using the same fixed-function lighting system used in PA1, and "Green Shader" uses a simple example shader that colors the object green. We include some scene files that test the remaining shaders (normal, Blinn-Phong, and toon) you will implement in Problem 1.

3 Requirements

The assignment is broken into three parts: making sure the functionality of PA1 works, implementing the suite of shaders in the modeler interface, and implementing the "flower" shader.

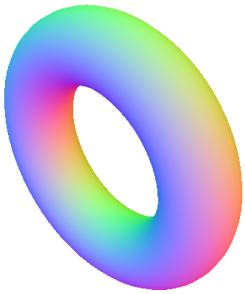

Figure 1: A torus rendered with the normal shader.

3.0 Problem 0: Functionality from PA1

Because the interface for testing the shaders extends the modeler interface from PA1, you must copy the functionality you implemented for PA1 into the PA2A code base. All the locations where you supplied code for PA1 are marked with `TODO PA1` in the source code. Note that you cannot simply replace entire `.java` files in the PA2A code base with their implemented versions from PA1; some of the classes contain new methods that have been added for PA2A.

We expect the functionality from PA1 be in working condition; a small part of your grade will go to including working implementations of the features from PA1. As a heads-up, this will be particularly important for PA2B, which will require you to add texture coordinates to your mesh generation code; if you need help fixing your PA1 code, you are encouraged to attend office hours.

3.1 Problem 1: A Suite of Shaders

In this problem, you will add three basic shading effects to the modeler interface of `ProblemA1.java`.

3.1.1 Normal Shader

Files: `normal_shader.vp`, `normal_shader.fp`

The normal shader visualizes the normal vectors assigned to a mesh by mapping the three components of a normal vector to an RGB color. For a (normalized) normal vector $\mathbf{n}$, we ask you to implement the formula

$$(r, g, b) = \left(\frac{\mathbf{n}.x + 1}{2}, \frac{\mathbf{n}.y + 1}{2}, \frac{\mathbf{n}.z + 1}{2} \right).$$

Note that this produces color values in the range $[0, 1]$, which in GLSL corresponds to the full range of brightnesses that can be displayed.

3.1.2 Per-pixel Blinn-Phong Shader

Files: `blinn_phong.vp`, `blinn_phong.fp`

In PA1, the modeler application rendered scenes using OpenGL's "fixed-function" lighting system, which uses Gouraud interpolation and a version of Blinn-Phong shading to set fragment colors. For

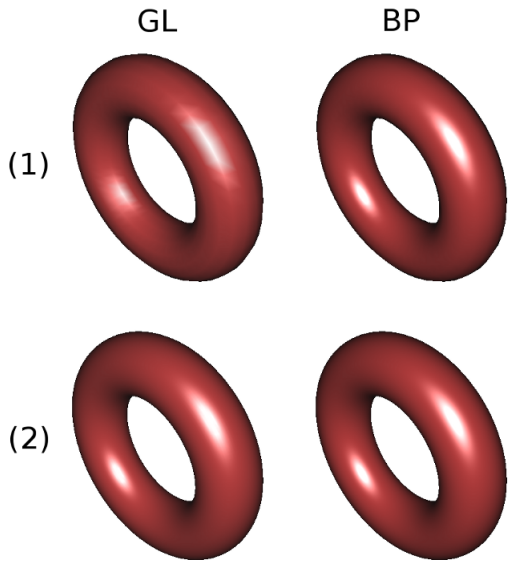

Figure 2: Comparison of GL fixed-function shading and per-pixel Blinn-Phong. Row (1) shows a lower-tessellation mesh, and (2) shows a highly tessellated mesh. On highly-tessellated meshes, the two shading models should be indistinguishable in your implementation.

this shader, you goal is to implement a *per-pixel* shader that implements the same lighting equation as OpenGL:

$$L_a = k_a I_a \qquad \text{(ambient)}$$
$$L_d = k_d I_d \max(\mathbf{n} \cdot \mathbf{l}, 0) \qquad \text{(diffuse)}$$
$$L_s = k_s I_s \max(\mathbf{n} \cdot \mathbf{h}, 0)^n \qquad \text{(specular)}$$

$$L = \begin{cases} L_a + L_d + L_s & \text{if } \mathbf{n} \cdot \mathbf{l} > 0 \\ L_a & \text{otherwise} \end{cases} \qquad \text{(final output)}$$

where $\mathbf{n}, \mathbf{h}, \mathbf{l}$ are defined as discussed in the lecture.

Note that OpenGL allows the user to supply separate intensity values I_a, I_d, I_s for the ambient, diffuse, and specular components of the light source. These intensities can be modified in the interface by clicking on the light node in the hierarchy. The final equation for L has two cases because if $\mathbf{n} \cdot \mathbf{l} < 0$, the light source is behind the surface, so neither the diffuse nor the specular term should contribute to the resulting color.

The information about the light source is available in the shader through the `gl_LightSource[0]` uniform (which is automatically provided to all vertex and fragment shaders). Its member variables

`gl_LightSource[0].position`, `gl_LightSource[0].ambient`,
`gl_LightSource[0].diffuse`, `gl_LightSource[0].specular`

give the eye-space coordinates of the light and its three intensity values. The uniforms `ambient`, `diffuse`, `specular`, and `shininess` contain the k_a, k_d, k_s, and n of the surface being illuminated.

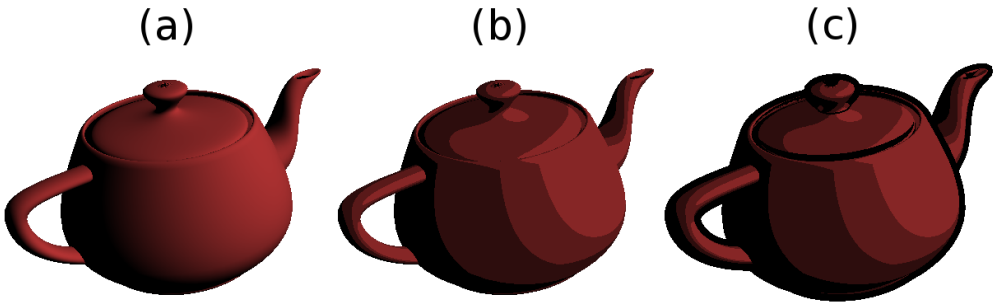

Figure 3: Building up the toon shader. Image (a) shows the teapot under diffuse shading, (b) shows the result of the quantize shader, and (c) is the final "toon" result with the outline from the displace shader.

Implemented correctly, your shader will match the result produced by fixed-function lighting on highly-tesselated meshes (see Figure 2).

3.1.3 Toon Shader

Finally, you will implement a "toon" shader that renders objects in a cartoon-like style (see Figure 3). The effect has two components: rendering the object with quantized colors (to imitate cell shading), and drawing an outline around the object using a displacement shader.

Quantization

Files: `toon_quantize.vp`, `toon_quantize.fp`

The quantized shader renders a modified version of the diffuse color L_d above (ambient and specular are ignored) by quantizing the cosine of the angle between the surface normal and the light vector. To make sure the boundaries between the color regions form smooth curves, perform this calculation per-pixel as you did in the Blinn-Phong shader. We ask you to quantize the cosine value according to the following pseudocode:

if $intensity \leq 0$ **then**
  $intensity \leftarrow 0$
else if $intensity \leq 0.5$ **then**
  $intensity \leftarrow 0.25$
else if $intensity \leq 0.75$ **then**
  $intensity \leftarrow 0.5$
else
  $intensity \leftarrow 0.75$
end if

Outlining

Figure 4: The flower mesh bending nonlinearly to follow a light. The un-bent mesh is second from the left.

Files: `toon_displace.vp`, `toon_displace.fp`
Classes: `MeshNode`

The outlining shader renders a version of the mesh that is displaced according to its surface normals. Each vertex of the mesh must be displaced a distance `displaceScale` in *eye* space in the direction of the normal at the vertex. The result is a "thicker" version of the object.

Remember that what we want is to draw an outline around the object. How does this displacement shader help us? OpenGL allows the user to request that only triangles facing *away* from the camera are drawn (front-face culling). If we draw the object with the displacement shader while front-face culling is enabled, the back half of the thickened mesh will fill in around the rendering of the object produced with the quantization shader, creating an outline. In addition to implementing the GLSL shaders, you must modify the indicated part of `draw` in `cs4620.pa2.scene.MeshNode` to render the mesh properly to produce the outlining effect.

Note: This method of producing outlines will sometimes produce bad results. For example, the outlines on the cube model will look strange from some angles; this is because sharp changes in the normal vectors of neighboring triangles will produce "gaps" in the displaced mesh. One can produce "correct" outlining by rendering the object and then performing a secondary edge-detection pass on the resulting depth information, but that is beyond the scope of this assignment.

3.2 Problem 2: Flower Shader

Files: `flower.vp`, `flower.fp`
Classes: `FlowerMaterial`

In this problem, you will implement a shader that transforms a flower mesh to make the flower bend towards the light source in the scene (Figure 4). In your shader program, the vertex shader will transform the mesh vertices and normals, and the fragment shader will render the bent flower

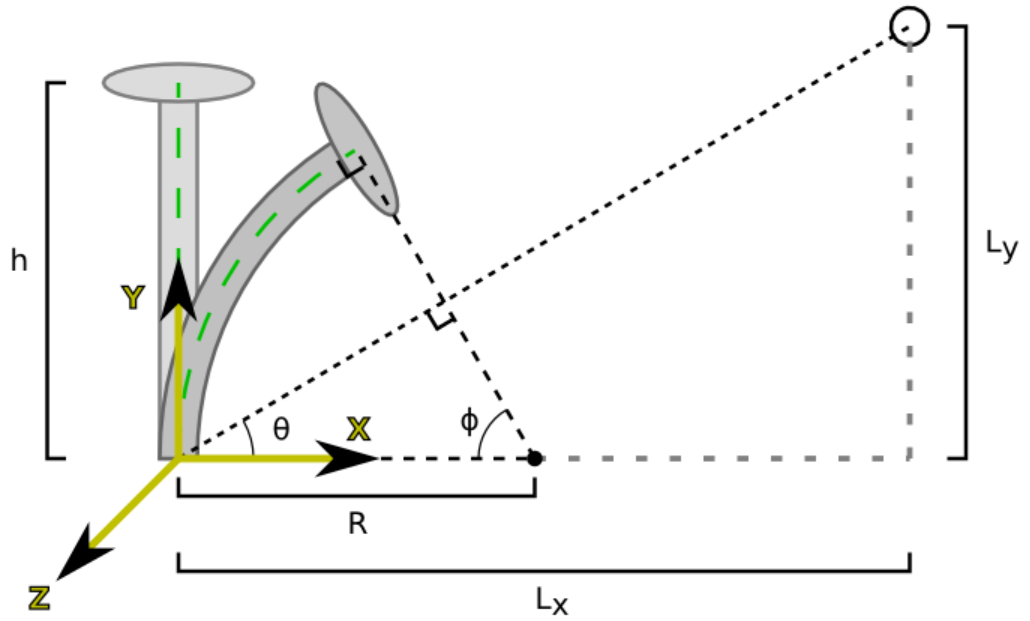

Figure 5: Geometry of the flower-bending method.

using the same per-pixel Blinn-Phong shading you implemented in Problem 1. To test the shader, we provide the program `ProblemA2.java` that shows the flower, a light source, and a slider that controls the motion of the light source in the scene.

3.2.1 The Transformation

Figure 5 shows the flower mesh before and after it is bent to point at the light. For now, assume that the light source lies in the x-y plane. Before applying the transformation, the flower mesh is located at the origin and extends to a height h along the y axis. The bending transformation takes the line running up the center of the flower stalk (the green line in Figure 5) and maps it to an arc of a circle, such that:

- The base of the bent stalk remains at the origin.
- The length of the bent stalk equals the length h of the stalk in the un-modified mesh.
- The top of the bent stalk points in a direction parallel to the vector from the origin to the light source.

This fully defines how to transform the center of the flower stalk. Finally, we can use the measures in Figure 5 to define how to move an arbitrary point $\mathbf{p}$ on the mesh to its transformed position $\mathbf{p}'$ (Figure 6). First of all, since all of the bending happens along the x-y plane, the z coordinates of vertices in the mesh are unchanged by the transformation. To get $\mathbf{p}'_x$ and $\mathbf{p}'_y$, notice that vertical lines in the untransformed flower of Figure 5 are mapped to arcs of circles centered at point $(R, 0, z)$ with

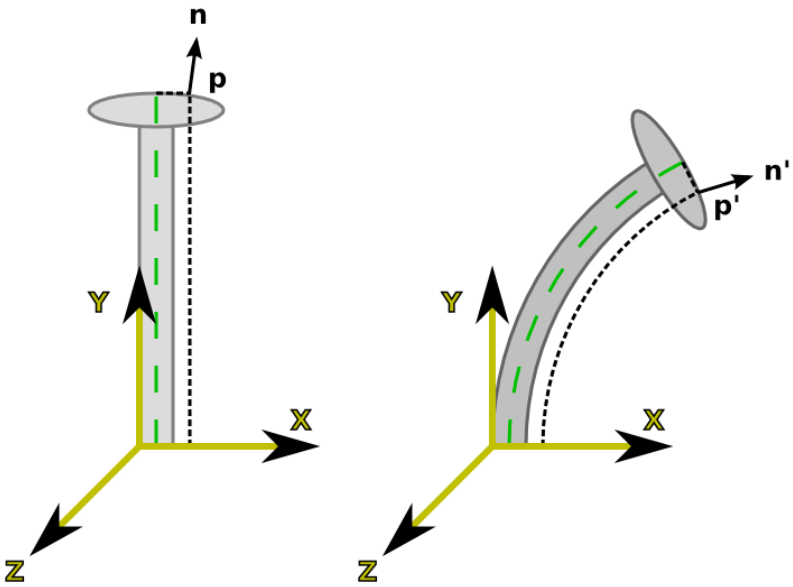

Figure 6: Transforming points on the flower.

different radii. In this mapping, $\mathbf{p}_x$ determines the radius of the circle containing $\mathbf{p}'$: for example, if $\mathbf{p}_x = 0$, the circle containing $\mathbf{p}'$ has radius R. Then, the height $\mathbf{p}_y$ of the original point determines where $\mathbf{p}'$ lies on this circle: for example, if $\mathbf{p}_y = 0$, then $\mathbf{p}'$ is at angle 0 above the x axis, and if $\mathbf{p}_y = h$, then $\mathbf{p}'$ is at angle ϕ above the x axis.

Similarly, the normal $\mathbf{n}$ at $\mathbf{p}$ must be transformed to produce a corresponding normal $\mathbf{n}'$ for the bent flower. To determine this transformation, notice that the small region around $\mathbf{p}'$ in the bent flower is rotated compared to the corresponding region around $\mathbf{p}$; this rotation is what must be applied to the original normal $\mathbf{n}$.

3.2.2 The Coordinate Frame

In Figure 5 and the section above, we simplified our math by assuming the flower and the light source both lie in the x-y plane. In general, the light source might not be on the x-y plane in the coordinate frame of the flower (the black axes in Figure 7). To fix this, define a new coordinate frame (the yellow axes in Figures 5 and 7) in which the flower center and the light source do fall on the x-y plane.

The matrices `objToFrame` and `frameToObj`, which map from the object's frame to the yellow light-source-aligned frame and vice versa, are already calculated by the `FlowerMaterial` class and provided to `flower.vp` as uniforms.

3.2.3 Putting It All Together

In total, the work done by your shaders will look like this:

- Map the flower (vertices and normals) into the frame aligned with the light source.

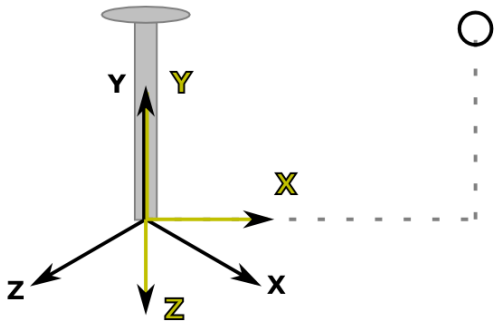

Figure 7: Defining a convenient frame for the flower problem. The axes of the object's local frame are in black, and the light-aligned frame is in yellow.

- Apply the transformation shown in Figure 5.
- Map the transformed geometry back to object space.
- Perform Blinn-Phong shading using the object-space, transformed geometry. This part should do the same work as your implementation of Blinn-Phong from Problem 1.

4 Implementation Notes

All the shaders you will write are loaded and compiled by classes we have provided. With the exception of the `FlowerMaterial` class in Problem 2, you will not need to modify these classes themselves; your work will go in the `.vp` and `.fp` shader source files in `src/cs4620/shaders`.

Aside from the normal shader, all the shaders you will implement use the ambient/diffuse/specular material properties of the object they are rasterizing. For these shaders, the modeler interface sets the material properties of the object being rendered as uniforms in the shaders.

Note: When you first run the shaders that have uniform variables (e.g. Blinn-Phong, Toon), the program will produce a bunch of warnings that the shaders are "ignoring" the uniforms. This is because the skeleton code we've given you for these shaders does not yet yet do anything with the uniform values it receives, and GLSL silently removes unused uniforms from your shaders. For example, once you modify `blinn_phong.vp` and `blinn_phong.fp` so that at least one of them uses `shininess` to calculate the output color, the `shininess` warning will disappear.

Below are some hints and suggestions for each problem:

1.
   - Normal shader:

     Hint: Look at the cs4620demos Eclipse project we released earlier in the term. One of the `Demo**.java` files solves a similar problem.
   - Blinn-Phong shader:

     Make sure you understand what `varying` variables do in GLSL, and when you should use them.

If you add multiple light sources to the scene in `ProblemA1.java`, the uniform array `gl_LightSource` contains information about multiple light sources. We only ask you to implement Blinn-Phong for a single light source, though, so your shader only needs to look at `gl_LightSource[0]`.

2. Flower shader:

   Note: When the flower is not bent (the light is directly above it), the mesh is rendered using your Blinn-Phong shader from Problem 1.

   First, we recommend using figures 5 and 6 to come up with explicit formulas for $\mathbf{p}'_x$ and $\mathbf{p}'_y$ based on $\mathbf{p}_x, \mathbf{p}_y$, and the values marked in Figure 5.

   To implement these equations in your shader, you will need to use `uniform` variables to tell the shader something about where the light source is located. Calculate your uniform values for the shader in the `setUniforms` method of the `FlowerMaterial` class, and send them to the shader program in the `use` method. You could just send L_x and L_y to your shader, but your shader code will be simpler if you can isolate parts of the equations for $\mathbf{p}'_x$ and $\mathbf{p}'_y$ that only depend on the light position, calculate these once in Java, and send them as uniforms to your shader.

Also, we have marked all the functions or parts of the functions you need to complete in with TODO in the source code. To see all these TODOs in Eclipse, select Search menu, then File Search and type TODO.

5 What to Submit

Submit a zip file containing your solution organized the same way as the code on CMS. Include a readme in your zip that contains:

- You and your partner's names and NetIDs.
- Any problems with your solution.
- Anything else you want us to know.