CS 4621 Practicum Programming Assignment 1
PPA1: Modeling and Shading

out: Tuesday 11 October 2011
due: Part (A): Tuesday 25 October 2011, Part (B): Tuesday 1 November 2011

1 Introduction

In this assignment, you will familiarize yourself with the basics of OpenGL and the GLSL language. The assignment consists of two parts.

In Part (A), you will work on a 3D modeling and shading system that uses simple primitives and curved surfaces organized in a transformation hierarchy. Initially, the modeler can show 3D scenes in orthographic and perspective views and has tools for editing shapes, transformations, and lights. It also provides camera control, a tree-style hierarchy view, type-in transformations and property editors. You will complete the modeler by writing the code that constructs triangle meshes for some of the shapes in the scene, and the code that performs mouse-based manipulation of transformations.

In Part (B), you will write a number of useful vertex and fragment shaders in GLSL. We will provide you with a number of test programs that load the shaders and apply them to meshes you constructed in Part (A).

We have split the assignment into two parts because it is long. Part (A), consisting of Problem 1 to Problem 3, is due in 2 weeks. Part (B), consisting of Problem 4, is due a week after that.

2 Modeler Overview

This section gives an overview of how the modeler, which is used in Part (A), works; more details are in the following sections. There is a brief user guide on the assignment's web page, which details the operation from the user's point of view.

The modeler's central data structure is a tree. Each node in the tree is either (1) an individual shape in the scene (a cube, sphere, cylinder, cone, torus, or custom triangle mesh), (2) a point light source, or (3) a transformation node that contains nothing in itself. Every node can have children. Moreover, each node has a 3D affine transformation associated with it, and this transformation affects the world-space position of all the nodes below it. This means that the modeling transformation for any particular object is the product of all the transformations along the path from the root to that object's node. The root node is always a transformation node.

Many lights can appear in the hierarchy, but only 8 of them will be functional because of OpenGL's limit on lighting. You may assume that in all of the test scenes we will have no more than 8 lights.

The main window displays a tree control in the upper left pane, which you can use to select nodes in the hierarchy; you can also select nodes by clicking on the objects in the viewports. You can use menu commands to create new nodes, which will appear as children of the selected node. You can change the selected node's properties in the lower left pane. There are also three menu commands to manipulate the hierarchy:

- Group creates a new transformation with the selected nodes as children;
- Reparent moves the selected nodes so that they are children of another node; and
- Delete removes the selected nodes from the tree.

The right pane is split into four viewports that view the scene with three perpendicular orthographic views and one perspective view. Each viewport has a camera associated with it, and by clicking and dragging in a viewport with various modifier keys you can move the associated camera. The orthographic cameras always maintain their view directions, but the perspective camera is fully adjustable.

You can change the transformation applied to objects by selecting a node and then choosing a manipulator from the Edit menu. The manipulator, a graphical tool for changing the transformation's parameters, then appears in the viewports. Initially, no manipulators function. It is your job to implement them such that, by dragging the parts of the manipulator around in those views, you can alter the selected transformation.

The available shape types are cubes, spheres cylinders, cones, tori, and custom triangle meshes. These objects are defined only in a canonical position and size, and all variations are handled by transformations. For instance, a sphere has no concept of a center and a radius; it is always a unit sphere centered at the origin, and the center and radius are adjusted by translation and scaling transformations that apply to the sphere. Details of each object's specification are available in the next section.

The program has a standard event-based structure, where an event generated by the user causes a change to the data structures and then all the viewports are redrawn. Drawing in the viewports is done using the OpenGL 3D graphics API; all other drawing happens using the usual Java 2D graphics. When the scene is drawn, the scene tree of transformations is traversed. If the node is a shape node, it is asked to supply OpenGL with a triangle mesh in its own coordinates, which is then transformed into world coordinates as the mesh is drawn.

3 Requirements

There are several core pieces of the program that need to be implemented before the modeler becomes a functional modeling program. The general areas are mesh drawing, mesh generation, manipulators, and shaders.

3.1 Problem 1: Mesh Drawing

To give you some practice using OpenGL, we leave it up to you to implement the drawing of generated meshes. Except for the drawing part, three types of shapes have already been fully implemented: cubes, tori, and custom triangle meshes which are loaded from files. Make sure that you draw these three types of meshes correctly before attempting the next problem.

3.2 Problem 2: Mesh Generation

You must write code to generate shared-vertex triangle meshes to approximate cylinders, spheres, and cones. In addition to the vertex positions and triangles connecting them, these meshes contain vertex normals for smooth shading.

The particular requirements for the individual shapes are:

1. *Sphere.* The sphere is a unit sphere centered at the origin. The normals should make the whole sphere look smooth.
2. *Cylinder.* The cylinder has radius 1 and height 2, and is centered at the origin. Its axis of symmetry is y axis. (So the bottom cap is at $y = -1$ and the top cap is at $y = 1$.) The side should be constructed with one row of triangles, and the end faces can be tessellated in any reasonable way you like ("reasonable" does not include generating degenerate triangles). The normals should be set up so that the sharp edges of the cylinder appear sharp, the flat ends appear sharp, and the side appears smooth.
3. *Cone.* The cone is a truncated cone, centered at the origin, aligned with the y axis. The total height is 1. The top radius is 0.5 and the bottom radius is 1. The cone should be tessellated in the same way the cylinder is.

Every mesh you generate must satisfy the following constraints:

- Every mesh vertex must coincide with a point on the shape. (On the other hand, a shape point may have multiple vertices coincide with it.)
- Every mesh vertex has only one normal vector associated with it.
- A vertex's normal must correspond to a normal associated with the shape point the vertex coincides with.

  Notice that a shape point may be associated with multiple normals. For example, consider a point on the edge of the cap of the cylinder. It has two normals associated with it: one is the cap's normal and the other is the side's normal. In this case, because a mesh vertex can have only one normal, there must be at least two mesh vertices for this shape point.
- The density of the meshes is determined by a global flatness tolerance ϵ that the user can adjust to trade off accuracy for speed. The mesh must be generated such that, for every triangle edge that approximate *curves* (as opposed to straight lines), the edge length is at most ϵ.

  Note that this constraint applies to only edges which approximate curves. If a triangle edge aligns with a straight line, there is no limit on how long it can be.

  There is also an exception for edges on sides of cylinders and cones. As we said, the sides must be constructed with only one row of triangles, it is fine that some edges on the sides may be longer than ϵ. However, edges on the circular caps must follow the contraints.
- Of course, the mesh must look correct. It must contain no holes and render correctly under Phong lighting.

3.3 Problem 3: Manipulators

The modeler provides a mean for editing transformations by typing in numbers. This can be useful if you want to apply a particular transformation exactly, but to get things where you want them it is much easier to position them interactively.

A widely used approach to transforming objects in 3D is by the use of *manipulators*. Manipulators are user interface elements that appear in the 3D scene to give the user direct visual cues about the operations that can be done. Transformations are edited by clicking on parts of the manipulator and dragging. For instance, the translation manipulator displays the three coordinate axes, and the user moves the selected object by clicking on one axis and dragging in the direction of desired motion. By clicking on the center of the manipulator one can drag the objects around freely in the viewport, resulting in a translation parallel to the view plane.

It is important to remember that a manipulator operates on a transformation, not directly on an object itself. This transformation applies to all objects below it in the tree.

Transformations in the modeler are represented as a nonuniform scale followed by x, y, and z rotations and then a translation; each manipulator affects one of these three pieces of the transformation. (In other words, the transformation matrix is $TR_zR_yR_xS$.) When the user adjusts one axis of a manipulator, the result is a change in exactly one number in the transformation (the x, y, or z component of the translation, scale, or rotation).

When manipulators are drawn on the screen they appear in 3D at the position in world space of the origin of the coordinate frame of the transformation being edited. The arrows of the scale and translation manipulators point exactly along the axes that correspond to x, y, and z adjustments to the scale or translation component. We call these the *manipulation axes* of the manipulator. The circles of the rotation manipulator are perpendicular to the three axes of rotation, unless the transformation being manipulated is below a nonuniform scale in the tree.

The modeler provides infrastructure for drawing the manipulators and detecting when they have been clicked on. It is your job to map the user's mouse motions into changes to the selected transformation. You should implement the manipulators for translation, rotation, and scaling based on the requirements below. The general principle is that when the user drags the mouse in a direction that the manipulator handle can move, the handle should move with the mouse, staying glued to the mouse pointer. (There are exceptions where it is difficult or nonsensical to have the transformation follow the mouse.)

1. *Translation.* The translation manipulator displays three arrows that represent the directions of motion that will result from an x, y, or z translation in the coordinates of the selected transform. If the user clicks and drags exactly in the direction of the axis, the resulting translation should exactly follow the mouse. When the drag is not parallel to the selected axis, the translation should follow the mouse as much as possible while still operating along the selected axis. If the user clicks the center of the manipulator, you should apply a translation parallel to the view plane so that the origin of the manipulator moves the same distance as the mouse—that is, if you move the mouse to the right 5 pixels and up 2 pixels, the manipulator should move to the right 5 pixels and up 2 pixels, carrying the affected objects with it. This should be true no matter what transformations are above the selected one.

2. *Rotation.* The rotation manipulator displays three circles on a sphere surrounding the origin of the transformation's coordinates. Each circle is perpendicular to one of the rotation axes,

and clicking on that circle and dragging performs a rotation around that axis. Because getting the transformation to follow the mouse is complicated for this manipulator, just map vertical mouse motion directly to the rotation angle.

3. *Scaling.* The scaling manipulator shows the three scaling axes by drawing three lines with small boxes at the ends. If the user clicks on one of these three axes and drags in the direction of the axis, a scale should be applied such that the amount of scaling change in the axis being manipulated is the distance in world space along the axis that the mouse moves. (See the next section for more details.) If the user clicks on the center cube and drags, a uniform scale should be applied. The uniform scale need not follow the mouse; you can simply map vertical or horizontal mouse motion directly to scaling.

All manipulations should have the property that dragging the mouse back to the starting point and releasing causes no change.

3.4 Problem 4: Shaders

In this problem we ask you to implement a few basic shaders.

I. **Normal Shader**

Each point on an object will have its color determined by its normal.

This is a fairly simple shader, where you have to color each vertex based on its normal vector and let the fragment shader interpolate those colors for you. In order to map a normalized vector (**n**) into a valid RGB triplet (values between $[0, 1]$), we ask you to implement the following formula $rgb = [\frac{\mathbf{n}.x+1}{2}, \frac{\mathbf{n}.y+1}{2}, \frac{\mathbf{n}.z+1]}{2}]$.

As mentioned above, this shader does not depend on the light in the scene, so you should always render the full color intensity (like in flat shading), but based on the given formula.

II. **Toon Shader**

Toon shading is a non-photorealistic rendering technique, that produces a cartoon-like, hand-drawn appearance.

Two main features create the effect: quantization and outlining. Quantization involves rendering objects surface in a piece-wise constant manner, and outlining involves drawing the edges of the object as thick, black line segments.

(A) Quantization.

To achieve the step-wise colorization, you have to compute intensity values per-fragment, based on the cosine of the angle between the surface normal (interpolated from the vertex shader to the fragment shader) and the direction to the light. Then you have to quantize the intensity, which is originally a value between $[-1, 1]$, into some small number of fixed values. In pseudo code, the quantization step should look like that:

```
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
```

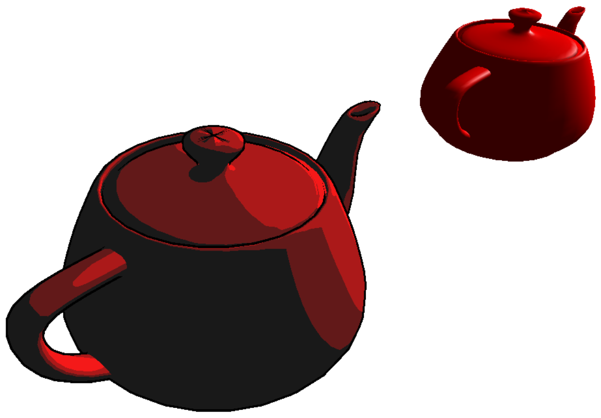

Figure 1: Toon shading vs Phong shading

That way instead of having smoothly varying intensity values along the surface of the object, you will get a few piece-wise constant regions, which will give you the feeling of a cartoon model. Note that the quantized intensity values only specify how the base color should be scaled, but the color itself is given in the OpenGL program, using glColor3f.

(B) Outlining.

The outline effect is created by rendering the edges of objects using thick, black segments.

There are different ways to achieve this effect, but they can be divided into two basic categories: Image Space Methods and Object Space Methods. Methods that work in “image space” operate on the pixel in the framebuffer as a second-pass rendering technique, which has the advantage of scaling extremely well, but is harder to implement. Object space methods operate on individual objects, where the goal is to find the edges between front-facing and back-facing polygons and render them thicker. Those edges represent the boundary between what the camera sees and what the camera doesn’t see.

Object space outlining can be implemented in a straight-forward manner using the OpenGL fixed functionality. The idea is to render the object twice. First, the object is drawn using the toon shader and after that the same object is drawn in wire-frame mode, with thicker, black lines on top of the first one.

As you want to draw two objects in virtually the same position, this might cause artifacts due to the well known phenomenon, called **Z-fighting**. If you don’t take special care, pixels will be rendered with fragments coming from the solid model or the wire-frame model arbitrarily, in a manner determined by the precision of the Z-buffer. What you really want to achieve is to displace the vertices of the wire-frame model just a little bit in the direction of their normals, so that it will "fatten" the solid model. In order to do that, you have to read how to use the `glPolygonOffset` function.

If you implement the above, you may notice that the front edges of the wire-frame model (the ones facing the camera) might become visible, but the whole point of using the wire-frame model is to emphasize only the edges that are part of the object's silhouette. In order to produce this effect, all you have to do is render the wire-frame model with front-face culling enabled, rather than the usual back-face culling.

That way, the intersection of the solid model with the offset back faces will "extract" the outline edges of our model.

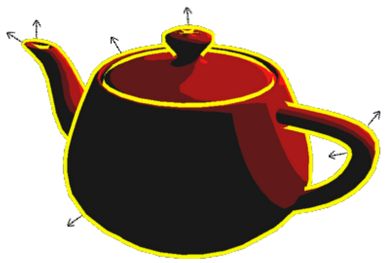

Figure 2: The outline edges (rendered yellow just for clarity) are displaced in the direction of the normals, in order to resolve Z-fighting artifacts.

III. **Displacement Shader**.

Displacement shaders are a family of **vertex shaders**, that modify the actual 3D position of the vertices in some procedural way.

In this problem, we ask you to implement a simple "Sine Shader", that will change the eye-space coordinates of each vertex ($\mathbf{v}$), using the following product of sine and cosine functions:

$$\mathbf{v}.y = \mathbf{v}.y + sin(time + \mathbf{v}.x) * cos(time + \mathbf{v}.z);$$

where "time" is an ever increasing uniform floating point variable, that should be updated from the OpenGL application in order to create the illusion of a simple animation. You are free to play with the update step (the value that increments the time uniform), but something like 0.01 might work well for you.

Your displacement **fragment shader** should implement the simple Lambertian lighting model (using interpolated normals for each fragment), when the directional light in the scene is enabled. Otherwise, it should render the colors with full intensity, as if the scale term in the Lambertian model was 1.

4 Implementation Notes

This section discusses how to implement the requirements; the suggestions here are just suggestions.

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.

4.1 Mesh drawing

To draw the mesh using OpenGL, fill out the `render` method of class `TriangleMesh`. Be sure to send not only vertex positions, but also the corresponding normal. We have provided the program `Problem1` under the package `cs4621.ppa1.p1` for you to look at your rendering results.

All meshes we will be rendering derive from the `TriangleMesh` class. Each instance of `TriangleMesh` stores its vertex positions in the `vertices` array, vertex normals in the `normals` array, and lists of indices of vertices that form triangles in the `triangles` array.

Data are packed into arrays so that coordinates of an item are consecutive. Vertex positions and normals are packed as:

Array indices	0	1	2	3	4	5	6	7	8	$\cdots$
Data	x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	$\cdots$

where (x_i, y_i, z_i) is the coordinate of the ith vertex's position or normal. Indices of triangle vertices are packed as:

Array indices	0	1	2	3	4	5	6	7	8	$\cdots$
Data	$i_{0,0}$	$i_{0,1}$	$i_{0,2}$	$i_{1,0}$	$i_{1,1}$	$i_{1,2}$	$i_{2,0}$	$i_{2,1}$	$i_{2,2}$	$\cdots$

where $i_{j,k}$ is the index of the kth vertex of the jth triangle. The code for cubes and custom triangle meshes follow these conventions, and we expect your code to do the same.

4.2 Mesh generation

For each shape, you need to implement the `buildMesh` method to construct a triangle mesh that approximates the shape. The shape classes are located in the `cs4621.ppa1.shape` package. The `buildMesh` method should replace the `vertices`, `normals`, and `triangles` field with arrays containing appropriate data for each shape. The flatness tolerance is given as an argument to `buildMesh`.

There are two things to note when building meshes. First, you will need to be very careful not to replicate vertices most of the time. If you replicate vertices for different triangles, the normals will not be smoothed properly when rendering. However, there are clearly times when you want to create sharp edges. To create a sharp edge in the mesh you must replicate the vertices and supply a different normal to each copy.

Second, if you divide a circle of radius r into k equivalent sectors, then the length of the base of the sector is $\frac{2\pi r}{k}$. Hence, if you want to approximate a circular vector with a triangle and want the triangle's base to be to be shorter than ϵ, then you should divide the circle into at least $\frac{2\pi r}{\epsilon}$ sectors.

You can build your mesh as a (u, v) grid for all the surfaces of revolution (cylinders, spheres). The diagram in Figure 3 illustrates this idea. The gridlines can be evenly spaced in u, with the number of steps in u determined directly from the global flatness. For the sphere, regular subdivision in v is fine; for the cylinder only a one-row grid is needed.

Like the first problem, we have provided the program `cs4621.ppa1.p2.Problem2` for you to look at your rendering results.

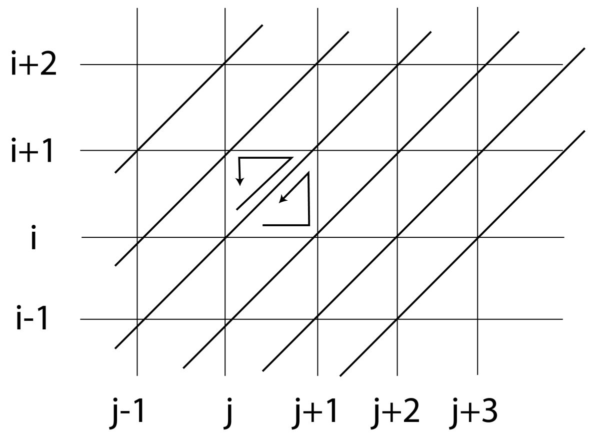

Figure 3: Diagram for surface triangulation.

4.3 Manipulators

The manipulators are all subclasses of `Manip`, and the main action is in the `dragged` method. This method has arguments `mousePosition`, which is the coordinate of the point where the mouse is, and `mouseDelta`, which is the offset from the previous point that was reported. The `Manip` class also stores the point where the user first clicked in the field `pickedMousePoint`. These coordinates are the xy-coordinate in *clip space*. The range of each component is from -1 to 1, inclusive.

One good way of setting up the follows-mouse constraint for manipulation along axes is as follows. If the user clicks on a point on a manipulation axis, then drags to another point on the axes, this means the eye rays corresponding to the initial and final mouse positions both intersect the axis. By computing these intersection points you can figure out what the transformation is that takes one to the other. If the user's clicks aren't exactly on the the axis, this means the rays don't exactly intersect the axis, but we can get a reasonable result if we just use the point of closest approach (the "pseudo-intersection" point) instead of the exact intersection. These computations are pretty easy to do using parametric lines, as we did in ray tracing. If you do it this way your work breaks down as:

1. Write a method to compute the parametric line (the viewing ray) in world space corresponding to a point in the image.
2. Write a method to compute the parametric line in world space that describes the manipulation axis. You had to do this anyway to position the manipulator in space.
3. Write a method to compute the pseudo-intersection of two parametric lines, returning the two t values for the closest pair of points on the two lines.
4. Do the manipulation by using this machinery to compute the initial and final t values on the manipulation axis; then updating the transformation is a simple matter.

You may find that your manipulator behaves a little strangely if the vanishing point of the manipulation axis is in the image and the user drags past it. This is OK.

The modeler is the program `Problem3` under the package `cs4621.ppa1.p3`.

4.4 Shaders

Like in the other problems, we have provided a program `Problem4.java` under the package `cs4621.ppa1.p4`, which you can use to test your shaders implementation. Here are some details of how the program and the shaders interact:

- You start with the well known four/one view windows.
- For simplicity, there will be only one, directional light in the scene, and you have to set it up. You should use the *GL_LIGHT0* light unit, where the direction should be stored inside the *GL_POSITION* attribute. This means that from your GLSL shaders, the light direction can be accessed through "*gl_LightSource[0].position*".

  The light will have impact on two of the three shaders (Toon and Displacement), and you should be able to turn it on and off. You have to write this logic of enabled/disabled light in those two shaders. The Normal shader should be invariant to the light in the scene, and thus to the state of the "Lighting" check box.
- From the upper-left combo box you can select between different primitives to render. Those include all objects that you have completed in Problem2, and we have also added the famous Utah teapot. You don't have to write code for the teapot model, it's there ready to use. This means that you can actually start working on the shaders assignment before completing any of the others, but you will be able to see the results only for the teapot model.
- From the upper-right combo box you can select which shader (Toon, Normal, Displacement) will be active (only one of them can be active at a time).
- On the right side you can still use the slider to control the tesselatlion level of the primitives that you have finished in Problem 2 (this slider does not affect the tessellation of the teapot model).

In the same file (`Problem4.java`) is also the first change that you have to make: inside the function `initLight()` you have to setup the directional light in the scene, which will be the same for all shaders.

All GLSL vertex and fragment shaders source files are located under `cs4621.glsl_shaders`. You can see that all of the required shaders are empty, with the exception that we put the one line of code `gl_Position = vec4(0, 0, 0, 0)` inside the vertex shaders, so that the "empty" programs can compile. You should remember, that one of your tasks as a vertex shader programmer is to assign the vertex screen coordinates to the `gl_Position` variable.

For your convenience, we have wrapped up the shaders in classes, so that we can expose through the interface of a base class the minimum functionality that you have to implement for this assignment.

You can find those classes in `cs4621.ppa1.shader`, where `BaseShader.java` defines the common interface and implements common functionality for all shaders (you don't have to do any changes here). We have derived a class for each of the required shaders (`ToonShader.java`, `NormalShader.java`, `DisplacementShader.java`), and part of your assignment is to implement the `render()` functions in all of the 3 shaders. Those functions should setup the OpenGL side of the rendering, before using the GLSL code.

- Normal Shader
  1. Implement `render()` in `NormalShader.java`
     Render the given mesh, using the Normal shader.
  2. Implement `normal_shader.vs`
     Set the color of each vertex based on its normal, and remember to encode the range from $[-1, 1]$ to the range of colors $[0, 1]$.
  3. Implement `normal_shader.fs`
     Color each fragment, using the interpolated colors from the vertices.

- Toon Shader
  1. Implement `render()` in `ToonShader.java`
     First, you have to render the given object using the toon shader. Then, you have to render it for a second time in wire-frame mode using the fixed OpenGL functionality. Remember to use `glPolygonOffset` to avoid Z-fighting artifacts.
  2. Implement `normal_shader.vs`
     Remember to "prepare" the normals (in eye space) for interpolation by the fragment shader.
  3. Implement `normal_shader.fs`
     If `enable_lights = 1`, you have to calculate the intensity based on the cosine of the angle between the (interpolated) normal and the direction to the light source. You should use the quantized intensity to scale the incoming fragment color (`gl_Color`). If `enable_lights = 0`, you should use `1` as the scale factor.

- Displacement Shader
  1. Implement `render()` in `DisplacementShader.java`
     Keep track of an ever increasing time variable and update the `time` uniform in the shader. You can modify the shader's uniform through the `timeUniform` object, which has been created for you.
     Finally, render the given mesh, using the Displacement shader.
  2. Implement `normal_shader.vs`
     Implement the sin/cos displacement function described above (in eye space), and remember to "prepare" the normals (again in eye space) for interpolation by the fragment shader.
  3. Implement `normal_shader.fs`
     Use the simple Lambertian model (with the interpolated normals), whenever the uniform `enable_lights = 1`, otherwise render colors with full intensity.

Note about compilation of the shaders: keep an eye on the console as you run the application, if you have any compilation or link errors in your shaders, you will see more information about it between the following tags:

******* **GLSL Begin Error** *******

// Error message goes here

******* **GLSL End Error** *******

The program will continue running, but you won't be able to use the shaders that did not compile.

5 Extra Credit

We recommend talking to us about your proposed extra credit first so we can steer you towards interesting mappings and make sure we agree that it would be worth extra credit.

Some ideas: trackball rotation, more complicated manipulator behavior, more shapes, using textures to eliminate the if-else sequence in toon shading quantization, and more complex shading.

Let us re-emphasize that, as always, extra credit is only for programs that correctly implement the basic requirements. And you should talk to us about it before implementing it since it might not be something we will give extra credit for unless we deem it is indeed interesting.